Графи. Дървета. Обхождане на графи.

Дефиниция: Нека $V = \{v_1, v_2, v_3, \ldots, v_n\}$ е крайно множество, елементите на което наричаме върхове, а $E = \{e_1, e_2, e_3, \ldots e_m\}$ е крайно множество, елементите на което наричаме ребра. Функцията $f_G : E \to V \times V$, съпоставяща на всяко ребро наредена двойка от върхове, наричаме ***краен ориентиран мултиграф***. Казваме още, че е зададен краен ориентиран мултиграф G с върхове V и ребра E и свързваща функция f_G и го означаваме с G(V, E, f_G).

Графите представяме в ранината, като всеки връх $v_i \in V$ представяме с точка, а реброто $e \in E$, такова че $f_G(e) = (v_i, v_j)$ изобразяваме с линия, започваща във v_i и завършваща със срелка във v_j (за да се посочи кой е вторият елемент на наредената двойка). Ребрата $f_G(e) = (v_i, v_i)$ наричаме *примки*. Името на реброто записваме до линията.

Дефиниция: Нека G(V, E, f_G) е краен ориентиран мултиграф и функцията f_G е еднозначна, т.е. f(e_i) ≠ f(e_j), i ≠ j. Тогава G(V, E, f_G) наричаме ***краен ориентиран граф***.

Дефиниция: Нека G(V, E) е краен ориентиран граф, такъв че релацията $E \subseteq V \times V$ е рефлексивна и симетрична. В такъв случай G(V, E) наричаме ***краен неориентиран граф*** или просто ***граф***.

Забележка: Крайният неориентиран граф G(V, E) можем да превърнем в ***краен неориентиран мултиграф***, ако позволим повече от едно неориентирано ребро да свързва два върха от V, т.е. ако вместо множество $E \subseteq V \times V$ вземем мултимножество от елементите на $V \times V$.

Дефиниция: Последователността от върхове $v_{i_0}, v_{i_1}, \ldots, v_{i_l}$ на крайния ориентиран мултиграф наричаме ***маршрут***,ако ∀j = 0, 1, 2,…, l-1, ∃ $e \in E$ такова, че $f_G(e) = (v_{i_j}, v_{i_{j+1}})$. Числото l наричаме дължина на този маршрут. В случая, когато $v_{i_0} = v_{i_l}$ маршрута наричаме ***контур***.

Дефиниция: Последователността от върхове $v_{i_0}, v_{i_1}, \ldots, v_{i_l}$ на крайният неориентиран граф G(V, E) наричаме ***път*** в G от v_{i_0} до v_{i_l}, ако $(v_{i_j}, v_{i_{j+1}} \in E)$, ∀j и $v_{i_{j-1}} \neq v_{i_{j+1}}$. Броят l на ребрата наричаме *дължина* на пътя. Ще считаме че съществува тривиален път с дължина 0 от всеки връх v_i до v_i (така отчитаме наличието на примки в неориентирания граф, но им даваме тежест 0 при образуването на дължината на пътя). В случая, когато $v_{i_0} = v_{i_l}$ пътят наричаме ***цикъл***.

За всеки краен неориентиран граф G(V, E) дефинираме релацията $P_G \subseteq V \times V$, така (v_i, v_j)$\in P_G$ тогава и само тогава когато съществува път в G от v_i до v_j.

***Теорема*:** Релацията P_G е релация на еквивалентност.

Дефиниция: Върховете от всеки клас на еквивалентност $V' \subseteq V$ на релацията P_G индуцират в G(V, E) по един подграф, който наричаме ***свързана компонента*** на G. Графът G(V, E) се нарича ***свързан***, ако $\forall\, v_i, v_j \in V$, $\exists$ път в G от v_i до v_j, т.е. G има точно една свързана компонента.

Забележка: Дефинията е приложима и за ориентирания случай, както и за мултиграф, със замяна на понятието път с маршрут.

Дървета

Дефиниция: Свързан граф D(V, E) без цикли наричаме ***дърво***. Несвързан граф без цикли – гора.

Дефиниция:

а) Графът D({r}, {}) наричаме *дърво с корен* r (***кореново дърво***). Единственият връх r е и единствен *лист* на това дърво.

б) Нека D(V, E) е дърво с корен r $\in V$ и листа l_1, l_2, ..., l_r. Нека $u \in V$ и $w \notin V$. Тогава $D'(V', E') = D'(V \cup \{w\}, E \cup \{(u, w)\}$ е също дърво с корен r. Ако $u = l_i$, $1 \le i \le r$, тогава листа на D' са l_1, l_2, ..., l_{i-1}, w, l_{i+1}, ..., l_r. В противен случай листа на D' са l_1, l_2, ..., l_r, w

в) Няма други коренови дървета

Забележка: Операцията която приложихме в индуктивната стъпка на горната дефиниция, наричаме *присъединяване на връх*. Кореновите дървета са неориентирани графи, но от дефиницията получаваме **неявна ориентация** на всяко ребро – от върха, към който присъединяваме (баща) към присъединения връх (син). Затова, при записване на ребрата на кореново дърво ще поставяме на първо място в наредената двойка бащата, а на второ – сина.

Теорема: **Всяко дърво с корен е дърво**

Доказателство: 1. Всяко кореново дърво е свързан граф. а) D({r}, {}) е очевидно свързан. б)Нека D(V, E) е свързан. в) $D'(V', E') = D'(V \cup \{w\}, E \cup \{(u, w)\}$,

също е свързан, защото между всеки два върха от *V* има път (от ИП), а от *w* до всеки друг връх има път с първо ребро (*w,u*).

2 Кореновото дърво няма цикли. а) D({r}, {}) очевидно няма цикли. б) Нека D(V , E) няма цикли. в) Допускаме, че D(V ', E ') има цикли. Тогава (u,w) участва в цикъл. Но d(w) = 1 $\Rightarrow$ противоречие. Сл. и D(V ', E ') няма цикли.

Теорема: Нека D(V , E) е дърво с корен r, тогава $|V|=|E|+1$

Доказателство: а) За D({r}, {}) – очевидно, понеже $|V|=1$, $|E|=0$. б) Допускаме че за кореновото дърво D(V , E) е в сила $|V|=|E|+1$. в) За кореновото дърво $D'(V',E')$, $V'=V\cup\{w\}$, $E'=E\cup\{(r,w)\}$ $\Rightarrow|V'|=|V|+1$, $|E'|=|E|+1$ $\Rightarrow|V'|-|E'|=|V|-|E|=1\Rightarrow|V'|=|E'|+1$

Дефиниция: Нека G(V , E) е граф, а D (V , E'), $E'\subseteq E$ е дърво. Тогава D наричаме ***покриващо дърво*** на G..

Обхождане на графи

Различните задачи върху графи могат да изискват специфичен алгоритъм за обхождане на графа. Съществуват обаче техники, които водят до построяване на сходни алгоритми за задачи, които на практика съществено се различават. Такива техники ще наричаме *алгоритмични схеми*. Под *обхождане* на граф ще разбираме процедура, която систематически, по определени правила „посещава” (разглежда) всички върхове на графа.

Обхождане в ширина: Съществено за тази алгоритмична схема е понятието *ниво на обхождане*, което представлява подмножеството на множеството от върхове на свързания граф G(V , E). В резултат на обхождането в ширина, V се разбива на нива $L_0, L_1, ..., L_k$ по следният начин)

1) Избираме начален връх i_0 на обхождането. $L_0=\{i_0\}$ и обявяваме върха i_0 за „обходен”, и $i=0$.

2) Ако $L_0\cup L_1\cup ...\cup L_i=V$ прекратяваме обхождането, иначе преминаваме към стъпка 3).

3) Нека сме обходили върховете от нивата $L_0, L_1, ..., L_i$.

   $L_{i+1}=\{v\,|\,v$ е „необходен” и $\exists w\in L_i, (v,w)\in E\}$.

   Обявяваме всички върхове $v\in L_{i+1}$ за „обходени”, $i=i+1$ и преминаваме към стъпка 2).

Забележка: Под „обхождане” на връх, разбираме някакви специфични действия върху разглеждания връх, зависещи от конкретната задача.

За илюстрация на алгоритмичната схема „обхождане в ширина” ще построим следният алгоритъм:

Алгоритъм за построяване на покриващо дърво (в ширина):

Нека е даден свързан граф G(V, E). Целта е да построим покриващо дърво D(V, E') на G.

Алгоритъм:

1) Коренът на покриващото дърво r ще изберем за начален връх на обхождането. Затова L_0={r}. Образуваме $D_0(V_0, E_0), V_0 = L_0, E_0 = \emptyset$, и $i = 0$.

2) Ако $V_i = V$ - край и резултатът е D_i, иначе продължваме със стъпка 3).

3) Нека $D_i(V_i, E_i)$ е дървото построена след i-тата стъпка и L_i са върховете от i-то ниво. $L_{i+1} = \{v \mid v \notin V_i, \exists w \in L_i, (w,v) \in E\}$. Образуваме дървото $D_{i+1}(V_{i+1}, E_{i+1}), V_{i+1} = V_i \cup L_{i+1}, E_{i+1} = E_i \cup \{(w,v) \mid w \in E_i, v \in E_{i+1}\}$, $i = i+1$ и преминаваме към стъпка 2).

Обхождане в дълбочина: За тази схема основни понятия са *текущ връх* t и *баща на върха* t – p(t).

1) Избираме i_0 за начален връх, $t = i_0$, а p(t) е неопределен.

2) Търсим „необходен” връх, който е съседен на t.

а) Ако има такъв връх v, тогава („стъпка напред”): p(v)=t, t=v и обявяваме v за обходен. Преминаваме към стъпка 2).

б) Няма такъв връх:

- Ако $t \neq i_0$ (началният връх) („стъпка назад”) t = p(t) и преминаваме към стъпка 2).

- Ако $t = i_0$ (началният връх) – край на обхождането.

Схемата за обхождане в дълбочина ни предписва да „опитваме” стъпки напред (в дълбочина) докато това е възможно, и при невъзможност да направим стъпка назад и отново да опитваме стъпка напред. Отново ще решим същата задача (построяване на покриващо дърво с корен r на свързан граф), но с алгоритмичната схема обхождане в дълбочина. Под „обхождане” на връх е естествено да разбираме включване на този връх в дървото.

Алгоритъм за построяване на покриващо дърво (в дълбочина):

Нека ни е даден свързан граф G(V, E). Целта е да построим покриващо дърво D(V, E') на G.

Алгоритъм:

1) Коренът на покриващото дърво r ще изберем за начален връх на обхождането. Образуваме $D_0(V_0, E_0), V_0 = \{r\}$, $E_0 = \emptyset$, $t = r$, $i = 0$ и p(t) е неопределен

2) Нека е построено дървото $D_i(V_i, E_i)$ Търсим $v \notin V_i$, такъв че $(t, v) \in E$:

а) Ако има такъв v, строим $D_{i+1}(V_{i+1}, E_{i+1}), V_{i+1} = V_i \cup \{v\}, E_{i+1} = E_i \cup \{(t, v)\}$, p($v$) = t, t = v, $i = i+1$ и преминаваме към стъпка 2).

б) Иначе (няма такъв връх).

б.1) Ако $t \neq r$, t = p(t) и преминаваме към стъпка 2).

б.2) Иначе – край. Търсеното дърво е $D_i(V_i, E_i)$

Ойлерови обхождания: Път в свързан мултиграф G(V, E), който минава еднократно през всяко ребро на мултиграфа, наричаме *Ойлеров път*. Ако Ойлеровият път има начало и край, които съвпадат то тогава той се нарича *Ойлеров цикъл*. Мултиграф, ребрата, на който образуват Ойлеров цикъл наричаме Ойлеров граф.

Теорема (Л. Ойлер): Свързаният мултиграф G(V, E) е Ойлеров $\Leftrightarrow$ всеки връх на G е с четна степен.

Доказателство:

$\Rightarrow$) Нека ребрата на G(V, E) образуват Ойлеров цикъл, тогава всеки връх има четна степен, понеже при обхождане на мултиграфа по Ойлеровия цикъл, на всяко ребро, „влизащо" във v_i съответства „излизащо" такова (различно от първото, според дефиницията за път), а цикълът съдържа всички ребра точно по един път.

$\Leftarrow$) Нека мултиграфът G(V, E) е свързан и такъв, че всеки връх има четна степен. Ще опишем процедура която построява Ойлеров цикъл като използва всички ребра от графа точно по един път.

1) Нека r е произволен връх в мултиграфа. Обявяваме го за текущ t.

2) Докато е възможно, строим път по следното правило, от върха t, в който се намираме, преминаваме към някой съседен връх v по

„необходено” ребро. Използваното ребро обявяваме за „обходено” а върхът v за текущ($t = v$).

Нека да допуснем че когато правилото от тази стъпка не е приложимо то $t \neq r$. Тогава ще се окаже че степента на t е нечетна (ние сме във връх от който не можем да продължим), тъй като когато сме влезли за последно във t сме обявили реброто за „обходено, а при предишни пъти когато сме минавали през този връх сме отбелязвали като обходени по 2 негови ребра – противоречие с четността на степените на върховете. Следователно когато 2) приключи то $t = r$ и всички на този етап обходени ребра образуват цикъл C.

3) Ако цикълът C съдържа всички ребра – край на обхождането. Построен е Ойлеровият цикъл C.

4) Вече имаме цикъл с начало и край r, но имаме оставащи ребра, които образуват цикли. Трябва да включим тези цикли в нашият цикъл C)

Ако премахнем всички „посетени” ребра от мултиграфа G(V, E), то отново в графа G(V, E) всички върхове са с четна степен. Търсим такъв връх r' в намерения вече сикъл C, от който излиза поне едно необходено ребро. Ако допуснем че няма такъв връх, и не всички ребра са обходени, ще получим противоречие със свързаността на мултиграфа. Преминаваме към стъпка 2) с начален връх $r = r'$ и построяваме нов цикъл C’. Разширяваме C с C’ във върха r' и преминаваме към стъпка 3).

Следствие: Свързаният мултиграф G(V, E) има Ойлеров път $\Leftrightarrow$ когато точно два върха са с нечетна степен.

3. Булеви функции. Пълнота. Съкратена ДНФ на БФ.

Дефиниция: Функциите $F_2 = \{f : J_2^n \to J_2 \mid n = 1,2,...\}$ наричаме **булеви (двоични)**. Булевите функции на n променливи означаваме с F_2^n

Нека F = { f_0, f_1, …} $\subseteq$ **F_2**. Нека X = { f, x, 0, 1, (,), <запетая> }.

По-нататък ще записваме думите fα, xβ, където α, β ∈ { 0, 1}$^+$ като f_i, x_j, където α е двоичното представяне на числото i, β е двоичното представяне на числото j.

Дефиниция: Дефинираме индуктивно понятието **формула над множеството от функции** F:

База: За всяка функция $f_i \in$ F на n променливи, думата $f_i (x_1, x_2, …, x_n) \in$ X* е формула над F.

Предположение: Нека $f_i \in$ F е функция на n променливи и $\varphi_1, \varphi_2, …, \varphi_n \in$ X* са формули над F или променливи, т.е. от вида x_k.

Стъпка: Тогава думата $f_i (\varphi_1, \varphi_2, …, \varphi_n) \in$ X* е формула над F.

БФ с една променлива са 4 и са представени в следната таблица:

x	f_0	f_1	f_2	f_3
0	0	0	1	1
1	0	1	0	1

Имената на функциите са както следва:

- f_0 (x) – константата нула; означаваме я с $\tilde{0}$
- f_3 (x) – константата единица; означаваме я с $\tilde{1}$
- f_1 (x) = x – идентитет
- f_2 (x) = $\overline{x}$ - отрицание на x

БФ с две променливи са 16 и са представени в следната таблица:

x	y	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Имената на функциите са както следва:

- f_0 (x, y) – константата нула, означаваме я с $\tilde{0}$. Приемаме същото означение, тъй като разликата е само в броя на променливите, а самата функция не зависи от тези променливи
- f_{15} (x, y) – константата единица, означаваме я с $\tilde{1}$
- f_3 (x, y) = x – идентитетът (не зависи от y)
- f_5 (x, y) = y – идентитетът (не зависи от x)
- f_{12} (x, y) = $\bar{x}$ - отрицанието на x (не зависи от y)
- f_{10} (x, y) = $\bar{y}$ - отрицанието на y (не зависи от x)

Следващите функции от F_2^2 съществено зависят и от двете променливи; ще ги означаваме както следва:

- f_1 (x, y) = $x \wedge y$ = xy – конюнкция на x и y; функцията можем да разглеждаме като умножение по модул 2;
- f_7 (x, y) = $x \vee y$ – дизюнкция на x и y;
- f_6 (x, y) = $x \oplus y$ – събиране по модул 2;
- f_9 (x, y) = $x \equiv y$ – еквивалентност на x и y;
- f_{13} (x, y) = $x \rightarrow y$ – импликация от x към y;
- f_{11} (x, y) = $y \rightarrow x$ – импликация от y към x;
- f_{14} (x, y) = $x|y$ – функция на Шефер;
- f_8 (x, y) = $x \downarrow y$ – функция (стрелка) на Пирс;

Свойства:

Комутативност - xy = yx, $x \vee y = y \vee x$, $x \oplus y = y \oplus x$

Асоциативност - (xy)z = x(yz), $(x \vee y) \vee z = x \vee (y \vee z)$, $(x \oplus y) \oplus z = x \oplus (y \oplus z)$

Дистрибутивност - $x (y \vee z) = xy \vee xz$, $x \vee yz = (x \vee y)(x \vee z)$, $x (y \oplus z) = xy \oplus xz$

Идемпотентност - $x \vee x = x$, xx = x, $x \oplus x = \tilde{0}$

Свойства на отрицанието - $x\bar{x} = \tilde{0}$, $x \vee \bar{x} = \tilde{1}$, $x \oplus x = \tilde{1}$

Свойства на константите - $x\tilde{0} = \tilde{0}$, $x\tilde{1} = x$, $x \vee \tilde{0} = x$, $x \vee \tilde{1} = \tilde{1}$, $x \oplus \tilde{0} = x$, $x \oplus \tilde{1} = \bar{x}$

Закон за двойното отрицание - $\bar{\bar{x}} = x$

Закони на Де Морган - $\overline{x \vee y} = \bar{x} \wedge \bar{y}$, $\overline{x \wedge y} = \bar{x} \vee \bar{y}$

Всяко едно от тези свойства може да се провери като директно сравним стълбовете на функциите отговарящи на двете формули. Ще използваме тези свойства за да покажем още две:

Слепване - нека $f \in F_2$; Тогава е в сила: $f\,x \vee f\,\bar{x} = f\,(x \vee \bar{x}) = f\,\tilde{1} = f$

Поглъщане - нека $f \in F_2$; Тогава е в сила: $fg \vee f = fg \vee f\tilde{1} = f(g \vee \tilde{1}) = f\tilde{1} = f$

Дефиниция: С [F] ще означаваме множеството от всички двоични функции, съпоставени на формулите над F и ще го наричаме **затваряне** на F (относно суперпозицията).

Дефиниция: Множеството от функции $F \subseteq \mathbf{F_2}$ е **пълно** в $\mathbf{F_2}$, ако $[F] = \mathbf{F_2}$.

Дефиниция: Двоичната функция $f(x, \sigma) = x^\sigma$ дефинираме така: $x^\sigma = x$, ако $\sigma = 1$ и $x^\sigma = \bar{x}$, ако $\sigma = 0$.

Лема1: $x^\sigma = 1 \Leftrightarrow x = \sigma$.

Доказателство: Достатъчно е да пресметнем стълба на x^σ и да видим, че $x^\sigma = x \equiv \sigma$, което доказва твърдението.

Дефиниция: Формули от вида $x_{i_1}^{\sigma_1} x_{i_2}^{\sigma_2} \ldots x_{i_k}^{\sigma_k}$, където $l_j \neq i_s$ при $j \neq s$, $\sigma_j \in \{0, 1\}$, наричаме **елементарни конюнкции**.

Теорема (Разбиване на БФ по част от променливите): Нека са избрани i, $1 \leq i \leq n$ от променливите на функцията $f(x_1, \ldots, x_n) \in \mathbf{F_2}$. Без ограничение на общността, нека това са първите i променливи. Тогава

$$f(x_1, x_2, \ldots, x_n) = \bigvee_{\forall \sigma_1 \sigma_2 \ldots \sigma_i} x_1^{\sigma_1} x_2^{\sigma_2} \ldots x_i^{\sigma_i} f(\sigma_1, \sigma_2, \ldots, \sigma_i, x_{i+1}, \ldots)$$

Доказателство: Нека $g(x_1, \ldots, x_n)$ е функцията определена от дясната част на равенството. Да пресметнем стойностите на функциите f и g за произволен вектор $(a_1, \ldots, a_n) \in J_2^n$. Вляво получаваме $f(a_1, \ldots, a_n)$. От Лема1 следва, че от всички 2^i елементарни конюнкции $x_1^{\sigma_1} x_2^{\sigma_2} \ldots x_i$, участващи в дясната част, само една има значение 1 – тази при която $\sigma_j = a_j$, j = 1,2,...,i. Останалите елементарни конюнкции имат стойност 0 и анулират съответните членове на многократната дизюнкция. Така за стойността на дясната част получаваме:

$$g(a_1, a_2, \ldots, a_n) = a_1^{a_1} a_2^{a_2} \ldots a_i^{a_i} f(a_1, a_2, \ldots, a_i, a_{i+1}, \ldots, a_n) \vee \tilde{0} =$$
$$= \tilde{1} f(a_1, a_2, \ldots, a_i, a_{i+1}, \ldots, a_n) = f(a_1, a_2, \ldots, a_n)$$

Следователно функциите от двете части на равенството съвпадат.

Теорема (Бул): Множеството $\{ x \vee y, xy, \bar{x} \}$ е пълно.

Доказателство: Ако $f = \tilde{0}$, можем да представим $f = x\bar{x}$ и тогава $f \in [\{ x \vee y, xy, \bar{x} \}]$.

Нека $f \neq \tilde{0}$. Тогава разлагаме $f(x_1, x_2, \ldots, x_n)$ по всичките n променливи и получаваме

$$f(x_1, x_2, \ldots, x_n) = \bigvee_{\forall \sigma_1 \sigma_2 \ldots \sigma_n} x_1^{\sigma_1} x_2^{\sigma_2} \ldots x_n^{\sigma_n} f(\sigma_1, \sigma_2, \ldots, \sigma_n)$$

Ако $f(\sigma_1,\sigma_2,\ldots,\sigma_n)=0$, съответният член в дясната част се анулира и може да не участва във формулата. Ако $f(\sigma_1,\sigma_2,\ldots,\sigma_n)=1$, то $x_1^{\sigma_1}x_2^{\sigma_2}\ldots x_n^{\sigma_n}f(\sigma_1,\sigma_2,\ldots,\sigma_n)=x_1^{\sigma_1}x_2^{\sigma_2}\ldots x_n^{\sigma_n}$. Така получаваме: $f(x_1, x_2, \ldots, x_n) = \bigvee_{\substack{\sigma_1\sigma_2\ldots\sigma_n\in B_2^n \\ f(\sigma_1,\sigma_2,\ldots,\sigma_n)=1}} x_1^{\sigma_1}x_2^{\sigma_2}\ldots x_n^{\sigma_n}$, което е формула над $\{ x \vee y, xy, \overline{x} \}$.

В означенията на доказателството, когато $f \neq \tilde{0}$, формулата се нарича **съвършена дизюнктивна нормална форма** на f.

Теорема на Пост-Яблонски за пълнота на множество БФ

Нека $F \subseteq \mathbf{F_2}$ **.** Множеството F е пълно ⇔ F не е подмножество на нито едно от затворените множества $\mathbf{T_0}$, $\mathbf{T_1}$, **S**, **M**, **L**, където

$\mathbf{T_0}$ **=** множ. от функции $f(x_1, \ldots, x_n) \in \mathbf{F_2}$**,** които запазват нулата. f(0,0,…0) = 0

$\mathbf{T_1}$ = множ. от функции $f(x_1, \ldots, x_n) \in \mathbf{F_2}$**,** които запазват единицата. f(1,1,…1) = 1

S = множ. на самодвойствените функции { $f(\alpha)$: за всяко α => $f(\alpha) = f^*(\alpha)$ } (*идентитет и отрицание*)

M = множ. на монотонните функции {$f(\alpha)$: за всяко $\alpha <= \beta$ (лексикографски) : $f(\alpha) <= f(\beta)$ }

L = множ. от линейните функции (представящи се с полином на Жегалкин – $a_0 \oplus a_1x_1 \oplus \ldots \oplus a_nx_n$)

3. Крайни автомати. Регулярни езици. Теорема на Клини.

$M = (K, \Sigma, \delta, s, F)$

Дефиниция: Краен детерминиран автомат е наредената петорка, където:

K

е крайно множество от състояния

Σ

е крайна азбука

$s \in K$

е началното състояние

$F \subseteq K$

е множеството от крайни състояния

$K \times \Sigma$

е функцията на преходите. Това е функция $\delta : \rightarrow$

$\delta(q, a) \in K$

Функцията на преходите определя това как се сменя текущото състояние според това каква е текущата буква в думата. Например, ако се намираме в състояние и текущата буква е , тогава следващото състояние в което трябва да се преместим е .

$(q_2, ababab)$

Дефиниция: Текуща конфигурация на автомата ще наричаме текущото състояние, в което се намираме и остатъка от думата, който не е прочетен. Както казахме по-рано това по еднозначен начин дефинира текущото състояние на изпълнение на автомата, тъй като по никакъв начин изпълнението не зависи от буквите, които са вече прочетени. Ще записваме конфигурациите с наредени двойки: , който означават състояние и остатъка от думата.

$e(q, e)) = q' (q', w')$

Дефиниция: Ако и са две конфигурации на M, тогава е изпълнено тогава и само тогава когато за някои символ и . Ако се намираме в конфигурацията , това означава, че сме прочели цялата дума и работата на автомата е приключила (тук използваме като означение на празната дума).

$(q, w)) \quad {}^{*}_{M} (q', w')$

Ще означаваме рефлексивната и транзитивно затворена релация на със , което означава, че ако , то тогава е достижимо от след последователност от преходи или дори без нито един преход. Така можем да дефинираме и кога една дума се разпознава от автомат M.

$L(M) = \{w : w \in \Sigma^*, (s,w) \bullet \overset{*}{}_{M} (q,e), q \in F\}$

Дефиниция: една дума се разпознава от автомат M, тогава и само тогава когато съществува състояние , такова че . Така казваме, че думата е от езика, който автомата разпознава, т.е. , където дефинираме.

$M = (K, \Sigma, \Delta, s, F)$

Дефиниция: Недетерминиран краен автомат е наредената петорка ,

където:

K

е крайно множество от състояния

Σ

е крайна азбука

$s \in K$

е началното състояние

$F \subseteq K$

е множеството от крайни състояния

$K \times (\Sigma \cup \{e\})$

е множество от преходите. : → (2 ?)

$\bullet_{M} p) \in \Delta$

Всяка една наредена тройка се нарича преход в M от в при входна буква . Възможно е да имаме преходи от вида , при които от преминаваме в без да четем буква от входа. Конфигурация дефинираме по същият начин както при детерминираните автомати. Релацията , се дефинира по аналогичен начин.

$L(M_1) = L(M_2)$

Дефиниция: Два автомата и са еквивалентни

Вече можем да въведем и следната теорема:

Теорема: За всеки недетерминиран автомат, съществува еквивалентен детерминиран автомат.

Доказателство:

[redacted]

Нека е КНА. Ще построим КДА еквивалентен на .Идеята е да си представим, че на всяка стъпка не се намираме в едно състояние, а в множество от състояния, което е множеството на всички състояния, в които можем да се намираме в момента за прочетената дума. Така ако имаме 5 състояния и за текущо прочетената дума можем да се намираме в състоянията и , можем да смятаме, че се намираме в състояние . Ако следващият символ от входа премества от в или , от в и от в , тогава следващото състояние за тази буква ще е .

[redacted]

Използвайки тази идея започваме конструирането. Логично, състоянията на ще бъдат . Крайните състояния на новата машина ще бъдат всички тези състояния, които съдържат в себе си крайно състояние на недетерминирания автомат. Функцията на преход е малко по сложна, поради това че при всеки един преход в недетерминираната машина е възможно да извършим и произволен брой -преходи. За да формализираме това, ще въведем следната дефиниция:

[redacted]

Дефиниция: За дадено състояние , нека , т.е. това е множеството от състояния достижими от , само чрез -преходи.

[redacted]

Множеството се пресмята лесно със следният алгоритъм:

$E(q) := \{q\}$

1. Първоначално

$E(q) := E(q) \cup \{r\}$

2. Докато има преход , такъв че и изпълни:

Този алгоритъм ще завърши след най-много броя на състоянията стъпки.

M'

Така вече можем да дефинираме формално автомата :

$K' = 2^K$

$s' = E(s)$

$F' = \{Q \subseteq K : Q \cap F \neq \emptyset\}$

$a \in \Sigma$

и за всяко и всяка буква :

$\delta(Q,a) = \bigcup \{E(q) : \exists (q,a,p), p \in K, q \in Q\}$

$\emptyset \in K' = \emptyset$

Сега остава да докажем, че така получения автомат е детерминиран и еквивалентен на . Това, че е детерминиран се вижда лесно, тъй като функцията по дефиниция има единствена стойност във всяка една точка и е дефинирана за всяко едно състояние и всяка една буква от азбуката (това че е възможно , не означава че функцията не е дефинирана, тъй като).

Сега ако докажем твърдението

$(q,w) \vdash_M^* (p,e) \Leftrightarrow (E(q),w) \vdash_{M'}^* (P,e), p \in P$

$w \in L(M') \Leftrightarrow (s,w) \vdash_M^* (f,e), f \in F \Leftrightarrow (E(s),w) \vdash_{M'}^* (Q,e), f \in Q \Leftrightarrow (s',w) \vdash_{M'}^* (Q,e), Q \in F'$

ще докажем теоремата много лесно. Това е така, защото ако вземем , то, като последната част на твърдението е дефиницията за това .

$|w|$

Ще докажем горното твърдение по индукция върху .

$w = e$
$|w| = 0$

1. За , т.е. , трябва да покажем че

$(q,e) \vdash_M^* (p,e) \Leftrightarrow (E(q),e) \vdash_{M'}^* (P,e), p \in P$

$p \in E(q)$

Първото твърдение е еквивалентно на това да кажем че ,а второто твърдение на това че , т.е. , с което твърдението е доказано.

2. Предполагаме, че твърдението е вярно за всички думи с дължина до .

k

$a \in \Sigma$

3. Трябва да докажем, че твърдението е вярно за всички думи с дължина . Нека , където ,а .

$(E(q),w) \bullet \overset{*}{}_{M} (R_1,a) \bullet {}_{M} (P,e)$ (p,e)

a.) нека , тогава съществуват две състояния и , такива че , от където следва, че можем да кажем , но тъй като , от индукционното предположение следва, че . Тъй като , то съществува , следователно от дефиницията на следва, че . Тъй като , следва че , следователно . От тук следва че ,от където .

$(q,va) \bullet \overset{*}{}_{M} (r_1,a) \bullet {}_{M} (r_2,e) \bullet \overset{*}{}_{M} (p,e)$

b.) за да докажем твърдението в обратната посока да предположим че , за някое , което съдържа и някое , за което . От дефиницията на , знаем че е обединението от всички множества , където за някое , съществува . Тъй като , съществува някакво , такова че и за някое има преход . Следователно от дефиницията на , следва че . От индукционното предположение следва че , от където .

Така теоремата е доказана.

Σ^*

Дефиниция: Нека имаме крайна азбука . Множеството от регулярни изрази във е:
$a \in \Sigma$

1. и всяка буква е регулярен израз
,

$(\alpha\beta)$

2. Ако и са регулярни изрази, то също е регулярен израз
$(\alpha \cup \beta)$

3. Ако и са регулярни изрази, то също е регулярен израз
α^*

4. Ако е регулярен израз, то също е регулярен израз
5. Няма други регулярни изрази освен тези описани в точки 1 до 4

$\Lambda(\alpha)$

Дефиниция: Ако имаме даден регулярен израз , регулярен език е множеството, където е функция дефинирана по следния начин:
$a \in \Sigma = \{a\}$

1. 1. и за всяко .
$\Lambda((\alpha\beta)) = \Lambda(\alpha)\Lambda(\beta)$

2. Ако и са регулярни изрази, тогава
$\Lambda((\alpha \cup \beta)) = \Lambda(\alpha) \cup \Lambda(\beta)$

3. Ако и са регулярни изрази, тогава
$\Lambda(\alpha^*) = \Lambda(\alpha)^*$

4. Ако е регулярен израз, тогава

Теорема: Класа от езици, които се разпознават от крайни автомати е затворен относно операциите:

1. Обединение
2. Конкатенация
3. Звезда на Клини
4. Допълнение
5. Сечение

Доказателство(тук е добра идея да се приложат и малко диаграми):

За всеки един случай ще покажем как можем да построим автомат, който разпознава езика резултат от съответната операция.

$M = (K, \Sigma, \Delta, s, F)$ $L(M_2)$

А. Обединение: Нека имаме два недереминирани крайни автомата и . Ще построим недерминиран краен автомат, който за който . Това ще направим като направим едно ново състояние и сложим преходи от това състояние към началните състояния на двата автомата (можем да приложим абсолютно същата техника за да построим обединение на произволен брой автомати). Ето как би изглеждала дефиницията формално: нека новия автомат е като,

$K = K_1 \cup K_2 \cup \{s\}$

,

$F = F_1 \cup F_2$

,

$\Delta = \Delta_1 \cup \Delta_2 \cup \{(s, e, s_1), (s, e, s_2)\}$

$L(M) = L(M_1) \cup L(M_2)$

Така при всяко едно пускане недетерминистично решава, кой от двата езика да разпознава, като по този начин разпознава и двата езика. Формално казано: ако , тогава за някое , за някое или , за някое . Тогава разпознава разпознава или разпознава , т.е. .

M_2 $M) = L(M_1) L(M_2)$

Б. Слепване: Нека пак и са два недетерминирани крайни автомати. Тогава за да построим автомат , за който , ще обединим състоянията на двата автомата (без ограничение на общността можем да считаме ще състоянията на двата автомата са непресичащи се множества) и от всяко крайно състояние на ще направим преход към началното състояние на . Доказателството е аналогично на доказателството от предишната точка.

$M_1 M_1) = L(M)^*$

В. Звезда на Клини: Нека е недетерминиран краен автомат. Искаме да построим автомат , такъв че . За целта ще използваме идея подобна на тази, която използвахме при слепването. Новия автомат ще има същите

състояние и преходи като , но ще има ново начално състояние, което е и крайно, за можем да разпознаваме празната дума. Освен това от всяко едно крайно състояние ще направим преход към (началното състояние на), както и преход от към . Така когато прочетем дума от , можем да започнем от началното състояние на с нова дума.

$\overline{M} = (K, \Sigma, \delta, s, K - F)$

Г. Допълнение: Нека е краен детерминиран автомат. Тогава допълнението на езика на този автомат е се разпознава от детерминирания краен автомат , т.е. просто правим всяко не-крайно състояние крайно и всяко крайно, не-крайно.

Д. Сечение: Имайки доказателствата до тук можем просто да си припомним, че

$L_1 \cap L_2 = \Sigma^* - ((\Sigma^* - L_1) \cup (\Sigma^* - L_2))$

Забелязваме, че дясната страна на израза използва само операции, които вече описахме, с което доказателството е завършено.

Теорема(на Клини): Един език е регулярен, тогава и само тогава, когато се разпознава от краен автомат.

Доказателство:

$\varnothing$

) Очевидно е че крайните автомати разпознават , както всеки език съставен от една буква. Освен това от предната теорема видяхме, че езиците, които се разпознават от автомати са затворени относно операциите обединение, слепване и звезда на Клини. Следователно всеки един регулярен език се разпознава от някакъв краен автомат.

$k = n\ k)\ n\ q_n\}\ F)$

) Нека е краен автомат (не задължително детерминиран). Ще конструираме регулярен език , такъв че . За целта ще представим като обединение на няколко по-прости езика. Нека и . За и , ще дефинираме да е множеството от всички думи от , които могат да накарат да отиде от състояние до състояние , без да се преминава през състояния със номера по-големи от , като и е възможно да са по-големи от . Забелязваме че при :

$R(i,j,n) = \{w \in \Sigma^* : (q_i, w) \bullet \overset{*}{}_M (q_j, e)\}$

Следователно:

$L(M) = \sim \{R(1,j,n) : q_j \in F\}$

$L(M)\ k)$

Идеята е, да докажем че всяко от множествата е регулярно, от където ще следва че също е регулярен.
$i = j \cup \{a \in \Sigma \cup \{e\} : (q_i, a, q_j) \in \Delta\}$

Ще докажем това с индукция по . За , е или ако или ако . Всяко едно от тези множества е крайно, от където следва че е регулярно.
$R(i,j,k-1)$

Предполагаме, че за , всяко едно множество е регулярно.
$R(i,j,k)$

Трябва да докажем, че е регулярно. За целта ще представим това множество с операциите обединение, слепване и звезда на Клини по следния начин:
$R(i,j,k) = R(i,j,k-1) \cup R(i,k,k-1) R(k,k,k-1)^* R(k,j,k-1)$

k_j

Идеята тук е, че всеки път от до , които не минава през състояния с номер по-голям от е един от следните типове:
$k-1$

1. Минава през състояния с номера по-малки или равни на $k-1$

2. Минава от до , после се върти от до , нула или повече пъти и накрая отива от до , като всеки един от тези подпътища не минава през състояния с номер по-голям от .
$R(i,j,k)$

От тук следва, че е регулярен, което завършва доказателството.

$i \geq 0 \in L$

Лема на покачването: Нека е регулярен език. Тогава съществува цяло число , такова че всяка дума , такава че , може да бъде представена като , като , и , за всяко .

Доказателство(тук май може да се докаже и с картинка):
w

е регулярен, значи съществува краен автомат , които разпознава . Нека е броя състояния на и нека е дума с дължина или повече. Да предположим, че първите стъпки от смятането на за са:

$(q_0, w_1 w_2 \ldots w_n) \vdash_M (q_1, w_2 \ldots w_n) \vdash_M \ldots \vdash_M (q_n, e)$

$v \; w_{j+1} \ldots w_m \; v_j$

където е началното състояние на и са първите букви от . Тъй като има състояния, а минаваме през конфигурации, то съществува състояния и , , такива че . Това е поддумата , която води от състояние обратно в същото състояние, и това не е празната дума тъй като . Но тогава тази поддума може да бъде премахната от или повторена произволен брой пъти след -тата буква на и ще продължи да разпознава новополучената дума. Следователно ще разпознава , за всяко , където и . Накрая забелязваме, че дължината на е , което по предположението по-горе е най-много , така както се изисква в теоремата. С това теоремата е доказана.

Примери за регулярни и нерегулярни езици:

$xz = a^{n-i}b^n \notin L$

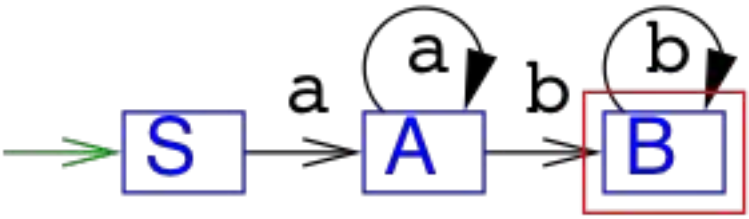
Езика не е регулярен. Ако беше тогава от горната теорема ще съществува , за което , такива че и , т.е. , но тогава , което противоречи на теоремата.

$p + sq + r = (q+1)(p+2q+r) > 0$

Езика не е регулярен. Ако беше тогава то горната теорема съществува , за което . Тогава . Тогава от теоремата следва че е просто, но за , получаваме , което е произведение на 2 естествени числа всяко от които по голямо от 1.

$L = \{a^n b^m : n > 0, m > 0\}$

Езика е регулярен. И неговия автомат би изглеждал така (автомата не е детерминиран, тъй като няма преход от S със b):



$\approx_L \in L \Leftrightarrow yz \in L$

Дефиниция: Нека е език и нека . Казваме че и са еквивалентни спрямо (), ако за всяко , следното е изпълнено: . Забележете, че е релация на еквивалентност.

$(s,y) \bullet \overset{*}{}_M (q,e)$

Дефиниция: Нека е детерминиран краен автомат. Казваме, че две думи , са еквивалентни спрямо (), ако има състояние , такова че и .

E_q

също е релация на еквивалентност и нейните класове на еквивалентност могат да бъдат идентифицирани, чрез състоянията на ,

които могат да бъдат достигнати от , чрез някаква дума. Ще означаваме класа на еквивалентност за дадено състояние , чрез .

$x \approx_{L(M)} y, \delta, s, F)$

Теорема: За всеки детерминиран краен автомат и две думи , ако , то .

$L(M)$

От горната теорема следва, че всеки един клас на еквивалентност спрямо се съдържа в клас на еквивалентност на и всеки един клас на еквивалентност на е обединение на един и няколко класа на еквивалентност на . Тъй като класовете на еквивалентност на са броя състояния на автомат разпознаващ , то можем да изкажем едно много важно свойство на всеки един автомат, които разпознава : всеки един автомат които разпознава трябва да има поне толкова състояния, колкото класове на еквивалентност има релацията . Следващата теорема доказва, че е възможно да се построи автомат разпознаващ с точно толкова състояния, което ще рече, че това е автомата с минимален брой състояния за езика .

$L(M)^*$

Теорема (на Михайл-Нероуд): Нека е регулярен език. Тогава съществува краен детерминиран автомат с брой състояния равен на броя на класове на еквивалентност на , който разпознава . Това е автомата с минимален брой състояния, който разпознава и този автомат е единствен с точност до изоморфизъм.

Доказателство:

$M = L(M), \Delta, \delta, F)$

Нека означим класа на еквивалентност спрямо , породен от , чрез . Ще построим краен детерминиран автомат, за който . Дефинираме по следния начин:

$\approx_L = \{[x] : x \in \Sigma^*\}$

, т.е. това са класовете на еквивалентност на

$s = [e]$

, класа на еквивалентност на празната дума

$F = \{[x] : x \in L\}$

$\delta([x], a) = [xa]$

Накрая дефинираме за всяко състояние и всяка буква , .

K_1

Така дефиниран има краен брой състояния, защото е регулярен, т.е. има краен детерминиран автомат , който го разпознава. От предишната теорема знаем, че има по-малко или равен брой класове на еквивалентност

в от колкото в , а има краен брой класове на еквивалентност тъй като има краен брой състояния, от където следва че и има краен брой класове на еквивалентност, т.е. е крайно.
M

Освен това е добре дефинирана функция на преходите, тъй като е еднозначна и дефинирана за всяко едно състояние на и всяка една буква от азбуката.
$x,y \in \Sigma^*)$

Остава да покажем, че . Първо ще покажем че за всеки имаме че:
$([x],y) \bullet \overset{*}{}_M ([xy],e)$

(1)

$([x],y'a) \bullet \overset{*}{}_M ([xy'],a) \bullet {}_M ([xy],e)$

Това се доказва с индукция по . За се доказва тривиално. Ако предположим, че е вярно за всяка една дължина на до и , тогава от индукционното предположение имаме: .
$x \in L^F M) \Leftrightarrow ([e],x) \bullet \overset{*}{}_M (q,e), q \in F$

Използвайки (1), доказателството е директно: за всяка , имаме че , което от (1) е същото като да кажем, че или от дефиницията на , .

$\approx_L$

Следствие: Един език е регулярен тогава и само тогава, когато релацията има краен брой класове на еквивалентност.

Алгоритъм за конструиране на минимален детерминиран автомат еквивалентен на даден детерминиран автомат:
$p \equiv_n q : (q,z) \in A_M \Leftrightarrow (p,z) \in A_M, z \in \Sigma^*, |z| \leq n$

Нека е детерминиран краен автомат. Дефинираме релацията , като . Казваме, че 2 състояния са еквивалентни (означаваме), ако за . Класовете на еквивалентност на са тези множества от състояния, които дефинират състоянията на минималният автомат, който разпознава . За да намерим тези класове на еквивалентност, ще пресметнем последователно класовете на еквивалентност на , където . Това ще направим последователно като ще намираме класовете на еквивалентност на една релация от класовете на еквивалентност на предишната релация.
$K - F$

1. За е очевидно, че класовете на еквивалентност са и .

$\equiv_n$ $a \in \Sigma, \delta(q,a) \equiv_n \delta(p,a)$

2. Да предположим, че сме намерили класовете на еквивалентност на . Тогава за всеки две състояния , и . По този начин можем да разберем дали 2 състояния са в един клас на еквивалентност или не от където можем да намерим класовете на еквивалентност на .
3. $\equiv_{n-1}$ Продължаваме докато не получим, че класовете на еквивалентност на са същите като класовете на еквивалентност на .

M

Алгоритъмът ще завърши след краен брой стъпки, тъй като на всяка една стъпка класовете на еквивалентност се увеличават поне с 1 и не може да има повече класове на еквивалентност от колкото състояния има в .

5. Контекстно-свободни граматики и езици. Стекови автомати

D Контекстно-свободна граматика G наричаме четворката (V, Σ, R, S), където:
- V е крайна азбука,
- Σ (множеството от терминали) е подмножество на V,
- R (множеството от правила) е крайно подмножество на $(V-\Sigma) \times V^*$, и
- S (начален символ) е елемент на $V-\Sigma$.

Елементите на $V-\Sigma$ се наричат нетерминали. За всяко $A \in V-\Sigma$ и $u \in V^*$ казваме, че $A \to_G u$ в G, когато $(A,u) \in R$. За всеки две думи $u, v \in V^*$ пишем $u \Rightarrow_G v$ т.с.т.к. има думи $x, y \in V^*$ и $A \in V-\Sigma$ такива че $u = xAy$, $v = xv'y$ и $A \to_G v'$. Релацията $\Rightarrow_G^*$ е рефлексивното, транзитивно затваряне на $\Rightarrow_G$. И нарая, $L(G)$, езикът генериран от G е $\{w \in \Sigma^* : S \Rightarrow_G^* w\}$, казваме още, че G генерира всяка дума от $L(G)$. Един език се нарича контекстно-свободен ако $L = L(G)$, за някоя КСГ G.

Когато граматиката се подразбира пишем $A \to w$ и $u \Rightarrow v$, вместо $A \to_G w$ и $u \Rightarrow_G v$.

Наричаме всяка последователност от вида:
$$w_0 \Rightarrow_G w_1 \Rightarrow_G w_2 \Rightarrow_G \dots \Rightarrow_G w_n$$
извод в G на w_n от w_0. Тук $w_0, \dots, w_n$ са произволни думи от V^*, а n наричаме дължина на извода, $n \in \mathbb{N}$ и $n \geq 0$. Казваме още, че изводът има n стъпки.

Пример! Следната граматика генерира всички изрази с правилно вложени скоби: всяка лява скоба трябва да има съответстваща дясна, и всяка затваряща трябва да има някаква съответстваща отваряща скоба. Още повече, нивото между всяка такава двойка скоби трябва да отговаря на същото условие. Нека $G = (V, \Sigma, R, S)$, където:
$V = \{S, (,)\}$,
$\Sigma = \{(,)\}$,
$R = \{S \to \varepsilon,$
$S \to SS,$
$S \to (S)\}$.

Два извода от тази граматика са:
$$S \Rightarrow SS \Rightarrow S(S) \Rightarrow S((S)) \Rightarrow S(()) \Rightarrow (())$$
и
$$S \Rightarrow SS \Rightarrow (S)S \Rightarrow ()S \Rightarrow ()(S) \Rightarrow ()(())$$
Следователно една дума може да има няколко различни извода в една граматика.
$L(G)$ е пример за език, който е контекстно-свободен, но не е регулярен.

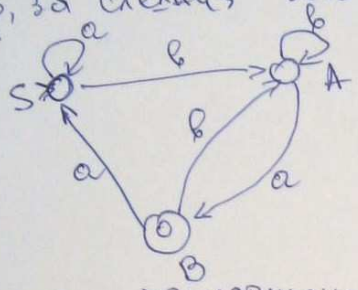
Очевидно има примери за езици, които са контекстно-свободни, но не са регулярни. Всички регулярни езици обаче са контекстно-свободни. Затова ще покажем няколко начина за директно построяване:

Нека разгледаме регулярния език, който се разпознава от крайния детерминиран автомат $M = (K, \Sigma, \delta, s, F)$. Същият език се генерира от граматиката $G(M) = (V, \Sigma, R, S)$, където $V = K \cup \Sigma$, $S = s$ и R се състои от тези правила:

$$R = \{q \to ap : \delta(q,a) = p\} \cup \{q \to e : q \in F\}$$

Тоест: нетерминалите са състоянията на автомата; за всеки преход от q към p при вход a имаме в R правило $q \to ap$.

Например, за следния автомат:



ще построим граматиката:

$S \to aS$, $S \to bA$, $A \to aB$, $A \to bA$, $B \to aS$, $B \to bA$, $B \to e$.

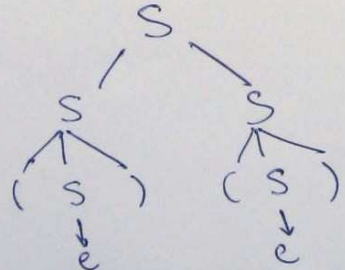
Дървета на извода / Дървета за синтактичен анализ

Нека G е КСГ. Както показахме, думата $w \in L(G)$ може да има много изводи в G. Например в разглежданата по-горе граматика за правилно вложени скоби, думата $()()$ може да бъде получена от S чрез поне два различни извода:

$S \Rightarrow SS \Rightarrow (S)S \Rightarrow ()S \Rightarrow ()(S) \Rightarrow ()()$, и

$S \Rightarrow SS \Rightarrow S(S) \Rightarrow (S)(S) \Rightarrow (S)() \Rightarrow ()()$

Ако разгледаме тези два извода ще забележим, че те са еднакви - различават се единствено по реда на прилагане на правилата. Интуитивно и двете могат да бъдат означени графично така:



Диаграми от горния тип нарисуваме дървета на извода. Точките нарисуваме върхове; всеки връх е означен със символ от V. Най-горният връх се нарича корен, а най-долните - листа. Всички листа са означени или с терминални символи, или с празната дума e. Ако слепим листата на дървото ще получим думата, която се нарича резултат на дървото за извода.

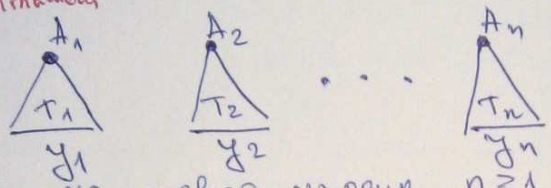
Или казано по-формално, за произволна КСГ $G=(V,\Sigma,R,S)$ дефинираме дърво на извода, корен, листа, ~~yield~~ резултат, така:

(1) 

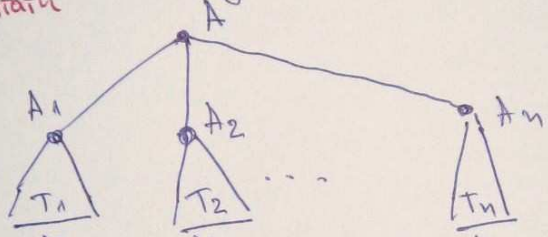
Това е дървото на извода за всяко $a \in \Sigma$. Единственият връх е и корен, и листо. ~~Yield~~ Резултатът е "a".

(2) Ако $A \to e$ е правило в R, то

е дърво на извод. Коренът му е означен с A, единственото му листо с e, ~~yield~~ резултатът е "e".

(3) Ако 

са дървета на извод, където $n \geq 1$, с корени обозначени с $A_1, \dots, A_n$ и ~~yields~~ резултати $y_1, \dots, y_n$ и $A \to A_1 \dots A_n$ е правило от R, то



е дърво на извода. Коренът му е новият връх A, листата му са листата на дърветата $A_1 \dots A_n$, yield е $y_1 \dots y_n$.

(4) Няма други дървета на извода.

Интуитивно, дърветата на извода представят изводите по такъв начин, че различните дъщерни се на реда на прилагане на правилата, се елиминират. Казано иначе, дърветата на извода дефинират класове на еквивалентности на изводите. По формално: Нека $G=(V,\Sigma,R,S)$ е КСГ, $D = x_1 \Rightarrow x_2 \Rightarrow \dots \Rightarrow x_n$ и $D' = x'_1 \Rightarrow \dots \Rightarrow x'_n$ са два извода в G, където $x_i, x'_i \in V^*$ за $i = 1 \dots n$, $x_1, x'_1 \in V - \Sigma$ и $x_n, x'_n \in \Sigma^*$. Тоест, и двата са изводи на терминални символни от един нетерминал. Казваме, че D предхожда D' (записваме $D \lessdot D'$), ако $n > 2$ и съществува $k \in \mathbb{N}$, $1 < k < n$, такова че:

(1) За всяко $i \neq k$ $x_i = x'_i$

(2) $x_{k-1} = x'_{k-1} = uAvBw$, където $u, v, w \in V^*$ и $A, B \in V - \Sigma$

(3) $x_k = uywBw$, където $A \to y \in R$

(4) $x'_k = uAvzw$, където $B \to z \in R$

(5) $x_{k+1} = x'_{k+1} = uyvzw$

С други думи, двата извода са идентични, с изключение на две последователни стъпки, през които едни и същи два нетерминала са заменени с едните същи думи, но в противоположен ред. Изводът, в който по-левият нетерминал е заменен първо предхожда другия.

Пример| Да разгледаме следните 3 извода D_1, D_2 и D_3

$D_1 = S \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow (())S \Rightarrow (())(S) \Rightarrow (())()$

$D_2 = S \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow ((S))(S) \Rightarrow (())(S) \Rightarrow (())()$

$D_3 = S \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow ((S))(S) \Rightarrow ((S))() \Rightarrow (())()$

Имаме, че $D_1 \prec D_2$ и $D_2 \prec D_3$, но $D_1 \prec D_3$ НЕ е изпълнено, тъй като D_1 и D_3 се различават в повече от една стъпка думата. И трите имат едно и също дърво на извода.

Казваме, че два извода D и D' са подобни, ако двойката (D, D') принадлежи на рефлексивното, симетрично, транзитивно затваряне на $\prec$ (следователно подобието е релация на еквивалентност).

Всеки такъв клас на еквивалентност (т.е. всяко дърво на извода), съдържа извод, който е максимален спрямо $\prec$ (не предхожда никой друг извод) – нарича се най-ляв извод и се получава като от корена A последователно заменяме най-левия нетерминал в думата според правилото, означено в дървото. Подобно се дефинира най-десен извод – извод, който не предхожда никой друг извод и се получава чрез последователно разширяване на най-десния нетерминал. Всяко дърво на извода определя точно един най-ляв и точно един най-десен извод.

За да обобщим, ще формулираме следната теорема:

Т| Нека $G = (V, \Sigma, R, S)$ е КСГ и $A \in V - \Sigma$ и $w \in \Sigma^*$. Следните твърдения са еквивалентни:

(a) $A \Rightarrow^* w$

(b) Съществува дърво на извода с корен A и резултат w.

(c) Съществува най-ляв извод $A \overset{L}{\Rightarrow}^* w$.

(d) Съществува най-десен извод $A \overset{R}{\Rightarrow}^* w$.

Стекови Автомати

Не всеки контекстно-свободен език може да бъде разпознат от краен автомат, понеже не всеки КСЕ е регулярен. За да получим автомат, който да разпознава КСЕ, е нужна свободна памет. Оказва се, че ключовата особеност на тези по-сложни автомати е добавянето на допълнителна памет, под формата на стек.

D| Ще дефинираме стеков автомат като наредена шестрка $M = (K, \Sigma, \Gamma, \Delta, s, F)$, където:
- K е крайно множество от състояния
- Σ е крайна азбука (входни символи)
- Γ е крайна азбука (символи на стека)
- $s \in K$ е начално състояние
- $F \subseteq K$ е множество от крайни състояния, и
- Δ, релация на прехода, е крайно подмножество на: $(K \times (\Sigma \cup \{\varepsilon\}) \times \Gamma^*) \times (K \times \Gamma^*)$.

Интуитивно, ако $((p, a, \beta), (q, \gamma)) \in \Delta$, когато M е в състояние p с β на върха на стека, може да прочете a от входа (или нищо, ако $a = \varepsilon$), да замени β с γ на върха на стека, и да премине към състояние q. Така двойката $((p, a, \beta), (q, \gamma))$ се нарича преход в M. Тъй като в даден момент може да има няколко възможни прехода, стековите автомати по принцип са недетерминистични.
Дефинираме две операции: push (добавяне на символ на върха на стека), и pop (премахване на символ от върха на стека).
Например преходът $((p, M, e), (q, a))$ демонстрира push, а $((p, M, a), (q, e))$ – pop.
Конфигурация на стековия автомат се дефинира като елемент на $K \times \Sigma^* \times \Gamma^*$: първият компонент е текущото състояние, вторият е частта от лентата, която още не е бегла прочетена, а третият съдържанието на стека, прочетено отгоре надолу.
Ако (p, x, α) и (q, y, β) са две конфигурации на M, казваме че (p, x, α) води след една стъпка до (q, y, β) (бележим $(p, x, \alpha) \vdash_M (q, y, \beta)$) ако съществува преход $((p, a, \beta), (q, \gamma)) \in \Delta$, такъв че $x = ay$, $\alpha = \delta\eta$ и $\beta = \gamma\eta$ за някое $\eta \in \Gamma^*$. Бележим рефлексивното и транзитивно затваряне на $\vdash_M$ с $\vdash_M^*$. Казваме, че M разпознава думата $w \in \Sigma^*$ т.с.т.к. $(s, w, e) \vdash_M^* (p, e, e)$ за някое $p \in F$. Казано иначе M разпознава w т.с.т.к. $\exists$ последователности от конфигурации $C_0, C_1, \ldots, C_n$ $(n > 0)$ такава че $C_0 \vdash_M C_1 \vdash_M \ldots \vdash_M C_n$, $C_0 = (s, w, e)$, $C_n = (p, e, e)$ за някое $p \in F$. Всяка последователност от конфигурации $C_0, C_1, \ldots, C_n$, такава че $C_i \vdash_M C_{i+1}$, $i = 0, \ldots, n-1$ се нарича изчисление от M; дължината ѝ е n, или броя на стъпки. Езикът, разпознат от M, бележим с $\mathcal{L}(M)$, е множеството от всички думи, разпознавани от M.
Когато не може да настъпи объркване пишем $\vdash$ и $\vdash^*$ вместо $\vdash_M$ и $\vdash_M^*$.

T| Класът на езиците, разпознавани от стекови автомати, съвпада с класа на контекстно свободните езици.
Доказателство: Ще разделим доказателството на две части

Лема Всеки контекстно-свободен език се разпознава от някой стеков автомат.

Доказателство:

Нека $G=(V,\Sigma,R,S)$ е КСГ; трябва да построим стеков автомат M, така че $L(M)=L(G)$. Ще построим автомат с само две състояния, p и q, който стои винаги в състояние q след първия си преход. M използва V, множеството от терминали и нетерминали, за азбуката на стека. Нека $M=(\{p,q\},\Sigma,V,\Delta,p,\{q\})$, където Δ съдържа следните преходи:

(1) $((p,e,e),(q,S))$

(2) $((q,e,A),(q,x))$ за всяко правило $A\to x$ в R

(3) $((q,a,a),(q,e))$ за всяко $a\in\Sigma$ (преход тип 1)

Стековият автомат първо слага S на върха на стека, и след това последователно или

- замяна най-горния символ A в стека, ако е нетерминал, с дясната страна x на някое правило $A\to x$ (преход тип 2), или
- премахва символа на върха на стека ако е терминал и съвпада със следващия символ на лентата (преход тип 3).

По този начин M симулира най-ляв извод на входа.

За да докажем, че $L(M)=L(G)$, ще използваме следното твърдение:

Твърдение Нека $w\in\Sigma^*$ и $\alpha\in(V-\Sigma)V^*\cup\{e\}$. Тогава $S\overset{L}{\Rightarrow}{}^*w\alpha$ т.с.т.к. $(q,w,S)\vdash_M^*(q,e,\alpha)$

Това твърдение ще ни помогне да докажем лемата, тъй като ако $\alpha=e$ ще следва, че $S\overset{L}{\Rightarrow}{}^*w$ т.с.т.к. $(q,e,S)\vdash_M^*(q,e,e)$, т.е. $w\in L(G)$ т.с.т.к. $w\in L(M)$.

$\Rightarrow$ Нека $S\overset{L}{\Rightarrow}{}^*w\alpha$, където $w\in\Sigma^*$ и $\alpha\in(V-\Sigma)V^*\cup\{e\}$. Ще покажем по индукция по дължината на най-левия извод на w от S, че $(q,w,S)\vdash_M^*(q,e,\alpha)$.

1. Ако дължината на извода е 0, тогава $w=e$, $\alpha=S$ и начална $(q,w,S)\vdash_M^*(q,e,\alpha)$

2. Допускаме, че ако $S\overset{L}{\Rightarrow}{}^*w\alpha$ при извод с дължина n или по-малко, $n\geq 0$, то $(q,w,S)\vdash_M^*(q,e,\alpha)$.

3. Нека
$$S=u_0\overset{L}{\Rightarrow}u_1\overset{L}{\Rightarrow}\dots\overset{L}{\Rightarrow}u_n\overset{L}{\Rightarrow}u_{n+1}=w\alpha$$
да е най-ляв извод на $w\alpha$ от S. Нека A е най-левият нетерминал на u_n. Тогава $u_n=xA\beta$ и $u_{n+1}=x\gamma\beta$, където $x\in\Sigma^*$, $\beta,\gamma\in V^*$ и $A\to\gamma$ е правило в R. Тъй като съществува най-ляв извод с дължина n на $u_n=xA\beta$ от S, то по И.П.
$$(q,x,S)\vdash_M^*(q,e,A\beta) \qquad (2)$$

Тъй като $A \to \gamma$ е правило в R:

$$(q, e, A\beta) \vdash_M (q, e, \gamma\beta) \qquad (3)$$

след преход от тип 2.

Тъй като u_{n+1} е ш.х, но и $x\gamma\beta$, има дума $y \in \Sigma^*$, такова че $w = xy$ и $y\alpha = \gamma\beta$. Тогава може да приложим (2) и (3), така:

$$(q, w, S) \vdash_M^* (q, y, \gamma\beta) \qquad (4)$$

Но $y\alpha = \gamma\beta \Rightarrow$

$$(q, y, \gamma\beta) \vdash_M^* (q, e, \alpha) \qquad (5)$$

след последователност от $|y|$ на брой прехода от тип 3.
Обединявайки (4) и (5) прилагаме индукционна.

$\Leftarrow$) Сега да допуснем, че $(q, w, S) \vdash_M^* (q, e, \alpha)$ с $w \in \Sigma^*$ и $\alpha \in (V - \Sigma)V^* \cup \{e\}$. Ще покажем, че $S \overset{L}{\Rightarrow}^* w\alpha$. Отново ще използваме индукция, този път по броя преходи от тип 2.

1. Тъй като първият преход винаги е от тип 2, ако $(q, w, S) \vdash_M^* (q, e, \alpha)$ без преходи от тип 2, то $w = e$ и $\alpha = S \Rightarrow$ вярно

2. Ако $(q, w, S) \vdash_M^* (q, e, \alpha)$ след n стъпки от тип 2 (или по-малко), $n \geq 0$, то $S \overset{L}{\Rightarrow}^* w\alpha$

3. Да предположим че $(q, w, S) \vdash_M^* (q, e, \alpha)$ с $n+1$ прехода от тип 2, и да разгледаме предпоследния такъв преход:

$$(q, w, S) \vdash_M^* (q, y, A\beta) \vdash_M (q, y, \gamma\beta) \vdash_M^* (q, e, a)$$

където $w = xy$ за някои $x, y \in \Sigma^*$ и $A \to \gamma$ е правило от граматиката. Според И.Х. имаме че $S \overset{L}{\Rightarrow}^* xA\beta$ и $\Rightarrow S \overset{L}{\Rightarrow}^* x\gamma\beta$. Но после $(q, y, \gamma\beta) \vdash_M^* (q, e, \alpha)$, евентуално след преход от тип 3, следва че $y\alpha = \gamma\beta$ и следователно $S \overset{L}{\Rightarrow}^* xy\alpha = w\alpha$. С това лемата е доказана, както и първата половина на теоремата.

Нека преминем към втората половина на Теоремата:

Лема 1 Ако един език се разпознава от стеков автомат, то той е контекстно свободен.

Доказателство:

Удобно ще е да ограничим някак стековия автомат, който разпознава. Ще наричаме стековия автомат *прост* ако:

Винаги когато $((q, a, \beta), (p, \delta))$ е преход в стековия автомат и q не е началното състояние, то $\beta \in \Gamma$ и $|\delta| \leq 2$.

с други думи винаги се разглежда само символа на върха на стека и са заменя или с ε, или с 1 или с 2 стекови символа.

Твърдим, че ако език се разпознава от обикновен стеков автомат, то той се разпознава и от прост стеков автомат. За дадена $M = (K, \Sigma, \Gamma, \Delta, s, F)$ е произволен стеков автомат. Ще построим $M' = (K', \Sigma, \Gamma \cup \{Z\}, \Delta', s', \{f'\})$ - прост, който също разпознава $L(M)$. Тук s' и f' са нови състояния, непринадлежащи в K, а Z е нов стеков символ, елемент дъното на стека (също непринадлежащ в Γ).

Започваме като към Δ^* добавяме преходите:

1) $((s', e, e), (s, Z))$
2) $((f, e, Z), (f', e))$ за $\forall f \in F$.

Така Δ' съдържа новите 2 прехода, както и всички преходи от Δ. Следващата ни стъпка ще е да заменим в Δ' всички преходи, които не удовлетворяват условието за простота, с еквивалентни преходи, които го удовлетворяват. Това ще стане на 3 етапа:

- Преходи с $|\beta| \geq 2$
- Преходи с $|\gamma| > 2$ (без да добавяме преходи с $|\beta| \geq 2$)
- Преходи с $\beta = e$ (без да нарушаваме предните 2).

1) За всички преходи от типа $((q, a, \beta), (p, \gamma)) \in \Delta'$ и $|\beta| > 1$, $\beta = B_1 \dots B_n$, $n > 1$, добавяме към Δ':

$((q, e, B_1), (q_{B_1}, e))$,
$((q_{B_1}, e, B_2), (q_{B_1 B_2}, e))$,
$\dots$
$((q_{B_1 \dots B_{n-1}}, a, B_n), (p, \gamma))$;

където за $i = 1 \dots n-1$, $q_{B_1 B_2 \dots B_i}$ е ново състояние със значение "състояние q след прочитане на символите $B_1 \dots B_i$ от стека". Очевидно новият автомат е еквивалентен на оригиналния.

2) По подобен начин заменяме преходите $((q, a, \beta), (p, \gamma))$, $\gamma = c_1 \dots c_m$, $m \geq 2$ с

$((q, a, \beta), (r_1, c_m))$
$((r_1, e, e), (r_2, c_{m-1}))$
$\vdots$
$((r_{m-2}, e, e), (r_{m-1}, c_2))$
$((r_{m-1}, e, e), (p, c_1))$

$r_1 \dots r_{m-1}$ са нови състояния. Нещо повече, сега $\forall$ преходи $((q, a, \beta), (p, \gamma)) \in \Delta'$ изпълняват $|\gamma| \leq 1$. Ще го възстановим до $|\gamma| \leq 2$ на следващата стъпка. Освен това, не добавихме преходи с $|\beta| > 1$.

3) Последно, нека разгледаме преходи от вида $((q,a,\varepsilon),(p,\delta))$ с $q \neq s'$ – само те всъщност нарушават условието за простота. Ще заменим тези преходи с $((q,a,A),(p,\gamma A))$ за $\forall A \in \Gamma \cup \{z\}$. Сега, ако автоматът може да се премества без да поглежда стека, то той може да се премества и ако погледне последния символ и върнато го обратно.

Лесно се вижда, че така получаваме прост автомат M' и $L(M)=L(M')$. За да докажем лемата ще покажем КСГ G, такава че $L(G)=L(M')$.

Нека $G=(V,\Sigma,R,S)$. V съдържа освен новия символ S и символите в Σ, нов символ $\langle q,A,p\rangle$ за $q,p \in K'$ и за всяко $A \in \Gamma \cup \{\varepsilon,z\}$. За да се разбере ролята на нетерминалите $\langle q,A,p\rangle$ трябва да се има предвид, че G трябва да генерира всички думи, разпознавани от $M' \Rightarrow$ нетерминалите на G представляват възможните части на входните думи, които се разпознават от M'. В частност, ако $A \in \Gamma$, тогава нетерминалът $\langle q,A,p\rangle$ представлява някоя част от входния низ, която може да бъде прочетена между момента, в който M' е в състояние q с A на върха на стека, и момента, в който M' премахва A от върха на стека и преминава в състояние p. Ако $A=\varepsilon$, то $\langle q,\varepsilon,p\rangle$ представлява частта от входната дума, която може да се прочете между състояние q и p, без стекът да се променя или променя.

Правилата в R са от 4 типа:

(1) Правилото $S \to \langle s,z,f'\rangle$, където s е началното състояние на M, а f' – новото крайно състояние.

(2) За всеки преход $((q,a,B),(r,C))$, където $q,r \in K'$, $a \in \Sigma \cup \{\varepsilon\}$, $B,C \in \Gamma \cup \{\varepsilon\}$ и за всяко $p \in K'$, добавяме правилото $\langle q,B,p\rangle \to a\langle r,C,p\rangle$.

(3) За всеки преход $((q,a,B),(r,C_1C_2))$, където $q,r \in K'$, $a \in \Sigma \cup \{\varepsilon\}$, $B \in \Gamma \cup \{\varepsilon\}$, и $C_1,C_2 \in \Gamma$ и за всяко $p,p' \in K'$, добавяме правилото $\langle q,B,p\rangle \to a\langle r,C_1,p'\rangle\langle p',C_2,p\rangle$.

(4) За $\forall q \in K'$, правилото $\langle q,\varepsilon,q\rangle \to \varepsilon$

Тъй като M' е прост, или тип 2 или тип 3 отговаря на всеки преход от M'.

Твърдение! За $\forall q,p \in K'$, $A \in \Gamma \cup \{\varepsilon\}$ и $x \in \Sigma^*$,

$\langle q,A,p\rangle \Rightarrow_G^* x$ т.с.т.к. $(q,x,A) \vdash_{M'}^* (p,\varepsilon,\varepsilon)$.

Леммата, а с това и теоремата, следват от това твърдение
Понеже $\langle s, e, f \rangle \Rightarrow_G^* x$ за някое $f \in F$ т.ст.к. $(s, x, e) \vdash_M^* (s', \varepsilon, e$
т.е. $x \in L(G)$ т.ст.к $x \in L(M')$.
И двете посоки на твърдението могат да бъдат доказани по индукция, или по дължината на изводи на G, или по дължината на изчислението на M!

Т| Контекстно-свободните езици са затворени спрямо обединение, слепване и звезда на Клини.

Доказателство:
Нека $G_1 = (V_1, \Sigma_1, R_1, S_1)$ и $G_2 = (V_2, \Sigma_2, R_2, S_2)$ са две КСГ. Без
можем да считаме, че $V_1 - \Sigma_1$ и $V_2 - \Sigma_2$ нямат общи елементи.
Обединение: Нека S е нов символ и нека $G = (V_1 \cup V_2 \cup \{S\}, \Sigma_1 \cup \Sigma_2, R, S)$, където $R = R_1 \cup R_2 \cup \{S \to S_1, S \to S_2\}$. Тогава твърдим, че $L(G) = L(G_1) \cup L(G_2)$. Понеже единствените правила, в които участва S, са $S \to S_1$ и $S \to S_2$, то $S \Rightarrow_G^* w$ т.с.т.к. или $S_1 \Rightarrow_G^* w$, или $S_2 \Rightarrow_G^* w$; и т.к. G_1 и G_2 имат ~~няма~~ множества от нетерминали без общи елементи, последното твърдение е еквивалентно на $w \in L(G_1) \cup L(G_2)$.
Слепване: Подходяждаме по подобен начин: $L(G_1)L(G_2)$ се генерира от граматиката:

$$G = (V_1 \cup V_2 \cup \{S\}, \Sigma_1 \cup \Sigma_2, R_1 \cup R_2 \cup \{S \to S_1 S_2\}, S)$$

Звезда на Клини: $L(G_1)^*$ се генерира от:

$$G = (V_1 \cup \{S\}, \Sigma_1, R_1 \cup \{S \to \varepsilon, S \to S S_1\}, S).$$

Т| Сечението на контекстно-свободен език и регулярен език е контекстно-свободен език.

Доказателство:
Нека L – КСЕ, R – регулярен език $\Rightarrow L = L(M_1)$ за някой стеков автомат $M_1 = (K_1, \Sigma, \Gamma_1, \Delta_1, s_1, F_1)$ и $R = L(M_2)$ за някой детерминиран краен автомат $M_2 = (K_2, \Sigma, \delta, s_2, F_2)$. Идеята е да комбинираме тези два автомата в един стеков автомат M, който извършва паралелно изчисленията на M_1 и M_2 и разпознава само тогава и двата бива разпознали.
Нека $M = (K, \Sigma, \Gamma, \Delta, s, F)$, където

$K = K_1 \times K_2$ (декартово произведение на състоянията)
$\Gamma = \Gamma_1$
$S = (S_1, S_2)$
$F = F_1 \times F_2$

Δ дефинираме така: За всеки преход на стековия автомат $((q_1, a, \beta), (p_1, \gamma)) \in \Delta_1$ и $\forall q_2 \in K_2$, добавяме в Δ $(((q_1, q_2), a, \beta), ((p_1, \delta(q_2, a)), \gamma))$; и за $\forall$ преход от типа ~~$((q_1, \varepsilon, \beta), (p_1, \gamma)) \in \Delta_1$~~ $((q_1, \varepsilon, \beta), ((p_1, q_2), \gamma))$. Тоест, M при четене от (q_1, q_2) в състояние (p_1, p_2) точно както M_1 минава от q_1 в p_1, освен че M запазва промените в състоянието на M_2, предизвикани от прочитането на входа.

Лесно се вижда, че ако $w \in L(M)$ т.с.т.к. $w \in L(M_1) \cap L(M_2)$.

Ранг на Thue

Δ fanout (бележим $\phi(G)$) е най-големият брой символи в дясната страна на някое правило от R.

Δ Път в дърво на извода е последователност от различни върхове, всеки свързан с предния; първият връх е коренът, а последният е листо.

Δ Дължина на път – броят ~~сегменти~~ дъги в пътя

Δ Височина на дърво – дължината на най-дългия път в него

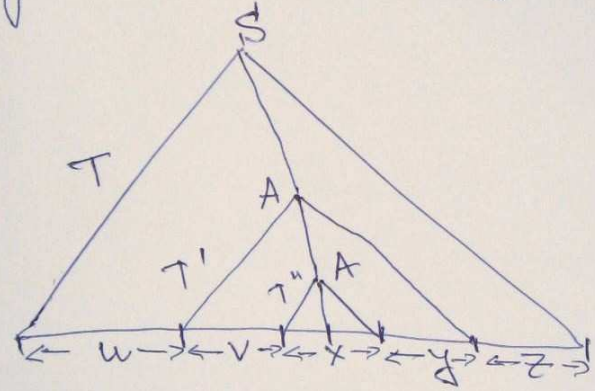
Лема Резултат ~~Yield~~ на което и да е дърво на ~~извода~~ синтактичния анализ с височина h е най-много $\phi(G)^h$.

Доказателство:
Индукция по h.
1. $h = 1$, дървото е някое правило на граматиката $\Rightarrow$ yield-ът има най-много $\phi(G)$ символа.
2. Да предположим, че твърдението е в сила за дърветата на извода с височина до h, $h \geq 1$.
3. Всяко дърво на извода с височина $h+1$ се състои от корен, свързан с най-много $\phi(G)$ дървета с височина не-повече от h. По индукция, yield-овете на всички тези поддървета са най-много $\phi(G)^h$, всяко $\Rightarrow$ yield на новото дърво е не повече от $\phi(G)^{h+1}$.

Т Нека $G=(V,\Sigma,R,S)$ е КСГ. Тогава $\forall$ дума $w \in L(G)$ с дължина поне от $\phi(G)^{|V-\Sigma|}$ може да бъде представена във вида $w=uvxyz$, така че или v или y е непразно и $uv^n xy^n z \in L(G)$ за $\forall n \geq 0$.

Доказателство:

Нека w е такава дума и T е дървото на извода с корен S и с резултат, който дума най-малък брой листа сред всички дървета на извод със същите корен и резултат. Тъй като дървото на извода T е по-дълбоко от $\phi(G)^{|V-\Sigma|}$, следва, че в T има път с дължина поне $|V-\Sigma|+1$, т.е. с поне $|V-\Sigma|+2$ върха. Само един от тези върхове може да е означен с терминал $\Rightarrow$ останалите са означени с нетерминали. Тъй като по пътя има повече върхове отколкото нетерминали, два върха от пътя са означени със същия елемент A от $V-\Sigma$. Да разгледаме пътя в повече детайли:



X - резултат на T''
V - частта от резултата на T' до началото на T''
U - частта от резултата на T до началото на T'
Z - останалата част от резултата на T.

Оттук е ясно, че частта от T' изключвайки T'' може да бъде повторена произволен брой пъти (включително 0), за да се получат други дървета на извода с yield $uv^n xy^n z$, $n \geq 0$. Остана само да докажем, че $vy \neq \varepsilon$. Но ако $vy = \varepsilon$, тогава има дърво с корен S и yield w с по-малко листа от T, получено ако от T премахнем частта от T', която изключва T'' $\Rightarrow$ Противоречие.

Т Контекстно-свободните езици не са затворени относно сечение или допълнение. Доказателство: вижте домашното.

Алгоритми за КСГ

Т1

а) Има полиномиален алгоритъм, който по КСГ конструира еквивалентен стеков автомат

б) Има полиномиален алгоритъм, който по стеков автомат конструира еквивалентна КСГ.

с) $\exists$ полиномиален алгоритъм, който по КСГ G и дума x определя дали $x \in L(G)$.

Доказателство:

Ще докажем част (с), тъй като (а) и (б) следват директно от конструкциите в доказателствата на предните леми КСГ стеков автомат.

Д1 КСГ $G=(V, \Sigma, R, S)$ е в Нормална форма на Чомски ако $R \subseteq (V-\Sigma) \times V^2$.

Т1 За всяка КСГ G $\exists$ КСГ G' в Нормална форма на Чомски, такава че $L(G') = L(G) - (\Sigma \cup \{\varepsilon\})$. Още повече, конструираното на G' може да бъде извършено за полиномиално относно големината на G време.

Доказателство:

Нека $G=(V, \Sigma, R, S)$. Съществуват 3 вида правила $A \to x$, които не отговарят на условията:

1) Дълги правила ($|x| \geq 3$).
Нека $A \to B_1 \dots B_n \in R$, $B_1, \dots, B_n \in V$, $n \geq 3$. Заместваме с $n-1$ нови правила:

$$A \to B_1 A_1$$
$$A_1 \to B_2 A_2$$
$$A_2 \to B_3 A_3$$
$$\vdots$$
$$A_{n-2} \to B_{n-1} B_n$$

$A_1 \dots A_{n-2}$ - нови нетерминали. След прилагане за $\forall$ такова правило получаваме еквивалентна граматика.

2) ε-правила ($A \to \varepsilon$).
За дадена определяме множеството:

$$\mathcal{E} = \{A \in V-\Sigma \mid A \Rightarrow^* \varepsilon\}$$

чрез следния алгоритъм:

$\mathcal{E} := \emptyset$
while there is rule $A \to \alpha$, $\alpha \in \mathcal{E}^*$ и $A \notin \mathcal{E}$ do
add A to $\mathcal{E}$

След това изтриваме от G всички ε-правила и: За $\forall$ правило от вида $A \to BC$ или $A \to CB$, където $B \in \mathcal{E}$ и $C \in V$ добавяме в граматиката $A \to C$. Единственият извод, който новата граматика би могла да изпуска, е $S \to \varepsilon$ - за щастие, формулировката на теоремата позволява това.

3) Хвси правила ($A \to a$ или $A \to B$)
За $\forall A \in V$ изчисляване множеството $D(A) = \{B \in V : A \Rightarrow^* B\}$:

$D(A) := \{A\}$
while there is a rule $B \to C$ with $B \in D(A)$ and $C \notin D(A)$ do
add C to $D(A)$.

Ако a е терминал, то $D(a) = a$.
Премахваме всички хвси правила и заменяме $\forall$ правила от вида $A \to B^* C^*$ с всички възможни правила от типа $A \to B'C'$, $B' \in D(B)$, $C' \in D(C)$. Накрая добавяме правилата $S \to BC$ за $\forall$ правило $A \to BC$ такова че $A \in D(S) - \{S\}$
Този нгт е възможно да пропуснем правила от вида $S \to a$, но отново това е позволено от информация.
Нещо повече, получения алгоритъм е полиномиален по време относно $n = |R|$.
Стъпка 1) отнема $O(n)$ и оставя граматика с големина $O(n)$.
Втората отнема $O(n^2)$, а третата - $O(n^3)$. С това теоремата е доказана.
Предимството на нормалната форма на Чомски е в това, че позволява използването на полиномиален алгоритъм за определяне дали дума може да се генерира от КСГ:
for $i := 1$ to n do $N[i,i] := \{x_i\}$; All other $N[i,j]$ are empty
for $s := 1$ to n do
for $i := 1$ to $n-s$ do
for $k := i$ to $i+s-1$ do
if there is a rule $A \to BC \in R$ with $B \in N[i,k] \& C \in N[k+1, i+s]$
add A to $N[i, i+s]$

Accept x if $S \in N(1,n)$.

Твърдение $\forall 0 \le s \le n$, след s итерации на алгоритъма за $\forall i = 1..n-s$:
$N[i, i+s] = \{A \in V : A \Rightarrow^* x_i \dots x_{i+s}\}$

Доказателство: Индукция по s.
1. $s = 0$ - изпълнява се само първият ред. Вярно
2. Допускаме, че е изпълнено за $\le s$. Да разгледаме извода на $x_i \dots x_{i+s}$ от нетерминала A. Тъй като G е в НФЧ, първото стъпка на извода е с правило от вида $A \Rightarrow BC \Rightarrow^* x_i \dots x_{i+s}$, където $B, C \in V \Rightarrow$ за някое k, $i \le k \le i+s$:
$B \Rightarrow^* x_i \dots x_k$ и $C \Rightarrow^* x_{k+1} \dots x_{i+s}$, т.е.
$x_i \dots x_k = x_i \dots x_{i+s'}$, $s' = k-i$
$x_{k+1} \dots x_{i+s} = x_{k+1} \dots x_{k+1+s''}$, $s'' = i+s-k-1$. Тъй като $i \le k < i+s$, то $s', s'' < s$
$\Rightarrow$ По И.Х. $\{A \in V : A \Rightarrow^* x_i \dots x_k\} = N[i,k]$ и $\{A \in V : A \Rightarrow^* x_{k+1} \dots x_{i+s}\} = N[k+1, i+s]$. Следователно $A \in \{A \in V : A \Rightarrow^* x_i \dots x_{i+s}\}$ т.с.т.к. $\exists k$, $i \le k \le i+s$ и два символа $B \in N[i,k]$ и $C \in N[k+1, i+s]$ такива че $A \to BC \in R$

Но това са точно обстоятелствата, при които алгоритъмът добавя A към $N[i, i+s]$

Времевата сложност на алгоритъма е $O(|x|^3|G|)$.

Сега част (c) от теоремата следва директно от теоремата и от горното твърдение. Първо, преобразуване G в КСТ в НФХ за полиномиално време, използвайки посочения алгоритъм. Ако $|x| \leq 1$, $x \in L(G)$ т.с.т.к. по време на трансформацията сме изтрили правилото $S \to x$. В противен случай използваме ~~[illegible]~~ алгоритъма за G' и низа x.

Допълнения:

Примери за езици, които не са контекстно-свободни:

→ $L = \{a^n b^n c^n : n \geq 0\}$

Да допуснем, че $L = L(G)$ за някоя КСГ G. Нека $n > \phi(G)^{|V-\Sigma|}$. Тогава $w = a^n b^n c^n$ е от $L(G)$ и може да се представи във вида $w = uvxyz$ така че v или y е непразно и $uv^m xy^m z$ е от $L(G)$, за всяко $m = 0, 1, 2, \ldots$. Има два случая:

- vy съдържа и трите символа a, b, c, тогава поне едно от v, y трябва да съдържа поне два от тях (Дирихле). Но тогава uv^2xy^2z съдържа два срещания в "грешен" ред - или "b" преди "a", или "c" преди "a" или "b"
- vy съдържа някои, но не всичките три символа. Но тогава uv^2xy^2z съдържа неравен брой a, b, c.

→ $L = \{a^n : n \geq 1 \text{ е просто число}\}$

Нека $p > \phi(G)^{|V-\Sigma|}$, където G е КСГ, която поражда L. Тогава $w = a^p$ може да бъде представена във вида $w = uvxyz$, $vy \neq \varepsilon$. Нека $vy = a^q$ и $uxz = a^r$, $q, r \in \mathbb{N}$, $q > 0$. Тогава, според лемата, $r + nq$ е просто, $\forall n \geq 0$. Но това е невъзможно, защото ако например $n = r + 2q + 2 \Rightarrow$ $r + 2q^2 + qr + 2q = r(q+1) + 2q(q+1) = (2q + r)(q+1)$ е просто $\Rightarrow$ ($2q + r \geq 2$, $q + 1 \geq 2$)

→ $L = \{w \in \{a, b, c\}^* : w \text{ има равен брой } a, b, c\}$.

Да допуснем, че L е КС $\Rightarrow$ по Th за сечение на КС и регулярен език, КС ще бъде и сечението на L с регулярния език $a^* b^* c^*$. Но като току-що доказахме, че този език ($\{a^n b^n c^n : n \geq 0\}$) не е КС $\Rightarrow$ ↯

Т Контекстно-свободните езици не са затворени спрямо сечение или допълнение.

Доказателство:

Ясно е, че $\{a^n b^n c^m : n, m \geq 0\}$ и $\{a^m b^n c^n : m, n \geq 0\}$ са КС. Сечението им е езикът $\{a^n b^n c^n : n \geq 0\}$, който показахме, че не е КС. И тъй като:

$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$ и понеже КСЕ са затворени спрямо $\cup$, ако КСЕ бяха затворени спрямо отрицание, щяха да бъдат затворени и спрямо сечение (а не са).

$\nexists$ КСЕ не са затворени спрямо $\cap$ и $-$.

5. Компютърни архитектури. Формати на данните. Вътрешна структура на централен процесор – блокове и конвейерна обработка, инструкции.

Обща структура на компютрите и концептуално изпълнение на инструкциите, запомнена програма. Формати на данните – цели двоични числа, двоично-десетични числа, двоични числа с плаваща запетая, знакови данни и кодови таблици. Централен процесор – регистри, АЛУ, регистри на състоянието и флаговете, блокове за управление, връзка с паметта, дешифрация на инструкциите, преходи.

Обща структура на компютрите.

Под **персонален компютър** разбираме изчислителна машина, използвана от един човек, за решаване на специфични, лични алгоритмични задачи и проблеми. Специфичните особености на персоналния компютър са, че той заема малък обем, има проста структура и система от команди, има ограничен обем на основната памет и опростен интерфейс, към който се свързват всички устройства в изчислителната система.

Въпреки голямото разнообразие на производители и размери, персоналните компютри се характеризират с еднакъв модел на вътрешна архитектура. Този модел е изграден от три основни компоненти:

- централен процесор;
- основна памет;
- входно-изходни устройства (периферни).

Взаимовръзката между основните компоненти определя логическата организация на съответната компютърна система.

Обменът на сигнали, команди и данни се извършва по специални информационни линии. Съществуват два вида логическа организация на компютърните системи:

1. Индивидуални информационни канали – основните компоненти се свързват по схемата на пълен граф, т.е. по схемата “всеки-всеки”;

2. Обща информационна шина - представлява специализирано устройство, отговарящо за пренасянето на информация между компонентите на компютърната система. Схематично организацията е следната:

ЦП = Централен процесор
ОП = Оперативна памет
ВИ = Входно-изходно устройство

Предимството на втората организация е, че тя има ниска цена, проста е, добре е структурирана и е гъвкава, т.е. възможна е лесна надстройка или замяна на основните компоненти. Основен недостатък на втората организация е неизбежното ограничение, че в даден момент могат да комуникират само два компонента, и произтичащата от това по-ниска производителност.

Шината представлява магистрала, по която може да бъде обменяна информация между две или повече устройства.
Шината е споделен ресурс, който всички устройства в конкурентен режим се мъчат да си разпределят. На ЦП се дава пълен приоритет, когато трябва да използва шината. Налага се въвеждане на специално устройство, което се нарича контролер на шината или арбитър, и то извършва разпределянето на шината, т.е. определя кое устройство за колко време да използва шината. Освен за обмен на данни, шината се използва и за предаване на управляваща и адресна информация. Затова условно шината се разделя на три подшини – **подшина за данни**, **подшина за адреси** и **управляваща подшина**. Подшината за данни прехвърля информация между определено място в паметта или входно-изходно устройство и централния процесор. Конкретното място в паметта или входно-изходното устройство се определя от адреса, който се предава по подшината за адреси. Управляващата подшина пренася електрически сигнали, които контролират комуникацията между централния процесор и останалите устройства. По нея се пренасят команди от централния процесор към устройствата и съответно се получават съобщения за статуса на устройствата. Например, управляващата подшина съдържа две линии за четене и за писане, които определят посоката, в която се пренасят данните. Други линии са линията за синхронизация, линии, по които се предават сигнали за прекъсване, и др.
Шината има следните характеристики:

- **ширина на подшината за данни** – количествена мярка за броя битове, които могат да преминават едновременно по подшината за данни;
- **ширина на подшината за адреси** – определя максималния размер на основната памет, например чрез 16-битова адресна подшина могат да се адресират 64KB памет, с 32-битова шина могат да се адресират 4GB памет;
- **скорост** – определя колко пъти шината може да бъде използвана (заемана) в рамките на една секунда, измерва се в MHz.

По-нататък ще разглеждаме само втората организация (обща информационна шина).

Формати на данните – цели двоични числа, двоично-десетични числа, двоични числа с плаваща запетая, знакови данни и кодови таблици.

В паметта данните се представят в двоичен вид. С помощта на n-битово поле могат да се представят точно 2^n различни стойности.

- **Цели двоични числа**

Първият начин за представяне на целите числа е чрез прав код. При него най-старшият бит в n-битовото поле е знаков – 0 означава положително число, 1 означава отрицателно число. Останалите n-1 бита представят абсолютната стойност на числото в двоична бройна система. По този начин при прав код диапазонът от цели числа, който може да се представи с n-битово поле, е $-2^{n-1}+1 \ldots 2^{n-1}-1$. При това нулата има две представяния – 000…0 и 100…0. Недостатъкът на правия код е, че събирането и изваждането на цели числа се реализират трудно апаратно.

Вторият начин за представяне на целите числа е чрез обратен код. Отново най-старшият бит е знаков. Разликата от правия код е в представянето на отрицателните числа – то се образува чрез инвертиране на двоичното представяне на абсолютната стойност на числото. Диапазонът при n-битово поле отново е $-2^{n-1}+1 \ldots 2^{n-1}-1$. Нулата отново има две представяния – 00…0 и 11…1. При обратния код събирането и изваждането се реализират

по-ефективно – числата, независимо от знака си, се събират като числа в двоична бройна система, при това, ако има пренос от най-старшия бит навън, то към резултата се добавя 1.

Третият начин за представяне на целите числа е чрез допълнителен код. Отново най-старшият бит е знаков. Разликата от правия код е в представянето на отрицателните числа – то се образува чрез инвертиране на двоичното представяне на абсолютната стойност на числото (както при обратния код) и след това добавяне на 1. Диапазонът при n-битово поле е -2^{n-1} … $2^{n-1}-1$. Нулата вече има единствено представяне – 00…0. При допълнителния код събирането и изваждането се реализират най-ефективно – числата, независимо от знака си, се събират като числа в двоична бройна система, при това преносът от най-старшия бит навън се игнорира.

- **Двоично-десетични числа**

Има още един за представяне на числата – така наречените двоично-десетични числа. При тях всяка десетична цифра се представя с уникална двоична последователност. Предимството е, че се дава възможност за лесно извеждане на числа. Недостатъкът е, че се усложняват аритметичните операции. Една цифра обикновено се представя с 4 бита, които в общия случай представят стойностите/цифрите/символите 0-9. Така 6 комбинации остават неизползвани. Тъй като компютрите съхраняват данни под формата на байтове, има два начина за съхраняване на 4-битовите цифри: всяка цифра се съхранява в отделен байт (непакетирана форма) или в един байт се съхраняват две цифри (пакетирана форма). Едно n-байтово пакетирано двоично-десетично число може да се състои от най-много 2.n-1 цифри, едната е резервирана за знак.

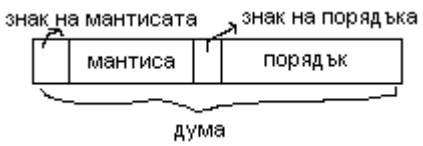
- **Двоични числа с плаваща запетая**

Реалните числа в паметта се представят чрез рационални апроксимации.

Първият подход е рационалните апроксимации да са с фиксирана точка. С други думи, в n-битово поле, първите m бита представят цялата част на числото, а останалите n-m бита представят дробната част на числото – десетичната точка е на фиксирана позиция. Проблемът на този подход е, че не винаги се използват всички битове.

Вторият подход е рационалните апроксимации да са с плаваща точка. Всяко число с плаваща точка има вида $m.2^p$, където m е цяло число, което се нарича мантиса, а p е цяло число, което се нарича порядък. Казваме, че едно число с плаваща точка е в нормализирана форма, ако най-старшият бит на мантисата е 1.

Всяко число с плаваща точка може да бъде нормализирано чрез изместване на мантисата наляво и съответно отразяване в порядъка. Нормализираната форма на едно число с плаваща точка е единствена и обикновено тя се използва за представяне на числото.

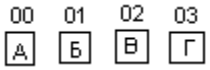


Възприет е стандартът едно число да се кодира в 32 бита, като полето за порядъка е 8 бита, а останалите 24 бита са за мантисата и знака. Освен това представяне обаче, за по-голяма точност се използват и двойни думи (по дължина) double precision от 64 бита.

- **Знакови данни и кодови таблици**

За универсално представяне на символите в паметта се постъпва по следния начин: съставя се кодова таблица, в която всеки символ от дадена азбука се асоциира с битова поредица с определена дължина. През 1978г. започва стандартизация на кодовите таблици и днес имаме два стандарта – ANSI и ASCII. ASCII таблицата първоначално е замислена като 7-битова, но впоследствие е разширена до 8-битова за нуждите на различни националности. В различните ASCII таблици символите, кодирани с 0 до 127 са едни и същи, а символите, кодирани със 128 до 255 могат да бъдат различни. Други примери за кодови таблици са EBCDIC – 8-битова кодова таблица на IBM, UNICODE – 16-битова универсална кодова таблица.

Във всяка КС има т. нар. компонент „знаков итератор". Той съдържа памет, която представлява правоъгълни матрици с еднакъв размер, като всяка матрица по битове описва чрез кодиране с 0 и 1 графичното изображение на някой символ.



Първият правоъгълник съответства на двоичен набор 00, вторият – на 01, третият – на 02 и така нататък. Така че, когато трябва да се визуализира графичен символ, програмата издава команда, която се свежда до двоичен код, който визуализира α_i.

ЦП – регистри, АЛУ, регистри на състоянието и флаговете, блокове за управление, връзка с паметта, дешифрация на инструкциите, преходи.

Централният процесор (ЦП) е основен компонент в архитектурата на персоналния компютър. Всяка задача, която се изпълнява от компютъра, директно или индиректно се изпълнява и контролира от ЦП. ЦП осъществява реализирането на машинните команди (инструкции). Някои негови съставни части са:

- управляващо устройство (УУ);
- аритметично-логическо устройство (АЛУ);
- регистри;
- блок за работа с числа с плаваща точка и др.

Регистрите представляват свръхбърза памет, която се използва за съхраняване на информация, върху която се извършват логическите и аритметични операции. Тази информация се съхранява, докато процесорът е под електрическо напрежение или докато някоя машинна команда не я измени. Според функционалността си, регистрите се делят на три вида:

- регистри с общо предназначение (R_1, R_2, …, R_n) – в тях се записват междинни резултати или се използват за адресация; програмистът или компилаторът на съответния език ги използват по свое усмотрение;
- регистри за работа с плаваща аритметика;
- служебни регистри – използват се за специални системни функции, към тях се включват флагов регистър (от голямо значение за управлението и протичането на програмата, съдържа информация за препълване, пренос, посока на обработка на низове, маскиране на прекъсванията и др.), PC (Program Counter, програмен брояч, сочи към следващата за изпълнение инструкция), SP (Stack Pointer, стеков указател, сочи към върха

на програмния стек), MSW (Machine Status Word, съдържа управляващи флагове, служещи за синхронизация на работата на ЦП с ВИ устройства и с ОС); MAR (Memory Address Register, държи адреса на мястото в паметта, където трябва да се изпълни следващата команда); MBR (Memory Buffer Register, действа като буфер, позволявайки ЦП и RAM паметта да работят независимо, без да им влияят малки разлики в работата).

Процесорът обработва стъпка по стъпка последователно, една след друга, командите от основната памет. Регистърът PC сочи в оперативната памет изпълнението на следващата команда. Процесорът изпълнява всички команди вътре в своето тяло и между своите регистри. Всяка команда се извлича от специален регистър в паметта, наречен регистър на командата, и след това процесорът изпълнява командите.

Операциите (аритметически и логически) се изпълняват от специализиран вътрешен блок – **аритметико-логическо устройство** (АЛУ), чиято основна част е двоичният суматор. Извличането на командата се изпълнява от друга част на ЦП, наречена **управляващо устройство** (УУ), което синхронизира работата на вътрешните компоненти на процесора, в това число и на АЛУ, чрез тактови импулси. **Блокът за работа с числа с плаваща точка** също като АЛУ извършва аритметични операции между две числа, но такива с плаваща точка, което е много по-сложно. Сложната структура на този блок включва и няколко АЛУ.

Концептуално изпълнение на инструкциите.

ЦП изпълнява една машинна команда чрез следната последователност от елементарни стъпки, която наричаме **основен/нормален цикъл на изпълнение на командата**:

Първите три стъпки се изпълняват от управляващото устройство (УУ):

1. Адресиране и извличане от основната памет в IR-регистър (Instruction Registry) на предстоящата за изпълнение команда (адресирането става по съдържанието на PC).
2. Обновяване на PC (съдържанието на PC се увеличава с дължината на извлечената команда, тоест се премества на следващата команда, УУ изпраща сигнал към АЛУ за изпълнение на командата).
3. Дешифриране на командата (определят се вида на операцията, броя на операндите, типа на данните и размера на данните).

Следващите стъпки се изпълняват от аритметико-логическото устройство (АЛУ):

4. Локализация на операндите (на базата на информацията от полето на командата по определени методи на адресация се определя местоположението на клетките от паметта, които съдържат операндите).
5. Извличане на операндите във вътрешни MBR регистри.
6. Изпълняване на операцията върху операндите.
7. Записване на резултата (определя се местоположението на резултата и той се занася в съответната клетка на паметта).
8. Проверява се има ли постъпил сигнал от някоя от сигналните линии за аварийна ситуация (за прекъсване). Ако да, управлението се предава на хардуерния авариен механизъм за обработка на прекъсването. След обработване на прекъсването (ако е имало такова) се преминава към стъпка 1 (започва изпълнение на следващата команда).

Типове команди.

1. **Аритметично-логическите команди** се изпълняват върху операнди, които са цели числа със или без знак. Част от тях изчисляват булевите функции конюнкция, дизюнкция, отрицание, сума по модул 2 и т.н. Друга част са операциите за сравнение – равно, различно, по-голямо, по-малко, по-голямо или равно, по-малко или равно и т.н. Последната част са аритметичните операции събиране, изваждане, умножение, целочислено деление и модул.
2. **Инструкциите за преход**. Те променят последователното изпълнение на програмата чрез промяна на стойността на програмния брояч PC. Включват инструкции за условен преход, инструкции за безусловен преход, инструкции за извикване на подпрограма и инструкции за връщане от подпрограма.
3. **Управляващи команди**. Това са инструкции, които се изпълняват, когато процесорът е в привилегирован режим – например чете или пише в служебните регистри. Процесорът преминава в привилегирован режим например при обработка на прекъсване и въобще при изпълнение на системна програма, която е част от операционната система.
4. **Транспортни команди**. Те служат за прехвърляне на данни. Основните са команда за прехвърляне на данни от едно място на паметта в друго, команда за прехвърляне на данни от паметта в регистрите и команда за прехвърляне на данни от регистрите в паметта.

Има и други типове команди – например операции върху числа с плаваща точка (събиране, изваждане, умножение, деление, степенуване, коренуване и др.), операции върху низове (прехвърляне, сравнение, конкатенация), мултимедийни операции MMX (специални векторни операции, които имат голямо приложение при обработката на изображения).

Запомнена програма.

Процесорът изпълнява и реализира командите на програмата. Всяка програма в даден момент от времето се намира в даден етап от своето изпълнение, тоест в някакво състояние. От друга страна, изпълнението на програмата представлява процес, който също се намира в някакво състояние. Това състояние се описва чрез понятието векторно състояние на процеса. Векторното състояние на процеса съдържа цялата информация, необходима за възобновяването на един спрян процес от точката на спиране по такъв начин, че все едно никога не е бил спиран. Включва в себе си: състоянието на програмата (изпълнимия код) в момента на спиране, състоянието на областите с данни, съдържанието на управляващите таблици (блокове) с данни на операционната система, съдържанието на регистрите на процеса и състоянието на входно-изходните устройства. По принцип се счита, че по време на изпълнението си програмата не се изменя, поради което няма смисъл нещо константно (неизменимо) да се съхранява. Счита се, че писането на програми, които се видоизменят по време на изпълнението си, е лош стил на програмиране, и въпреки това и днес той е все още честа практика.

10. Компютърни архитектури: Структура и йерархия на паметта. Сегментна и странична преадресация. Система за прекъсване – приоритети и обслужване.

Структура на основната памет. Йерархия – кеш, основна и виртуална памет. Сегметна и странична преадресация – селектор, дескриптор, таблици и регистри при сегметна преадресация; каталог на страниците, описател, стратегии на подмяна на страниците при странична преадресация. Система за прекъсване – видове прекъсвания, структура и обработка, конкурентност и приоритети, контролери на прекъсванията.

Структура на основната памет.

Памет в персоналния компютър се нарича всеки ресурс, който има свойството да съхранява информация във времето. Паметта представлява съвкупност от битове. Под един бит разбираме количество информация, за която отговаря един електронен елемент. Битът е минималната порция информация, която се съхранява, и има 2 стойности – 0 и 1. С развитието на компютрите се оказва, че най-удобната адресуема единица е байтът. Един байт е осем бита. Капацитетът на паметта се мери в байтове. Никоя памет концептуално не може да чете част от байт. Паметта е организирана в клетки, които са наредени последователно една след друга, т.е. като едномерен линеен масив от байтове, които се номерират с последователни номера, наречени адреси. Номерацията започва от 0 и стига до последния наличен физически адрес.

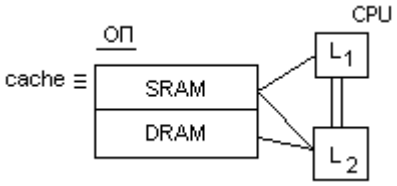
Паметта се изгражда като устройство, което може да извършва следните две операции:

- четене – подаване на адрес и извеждане съдържанието на съответния байт;
- писане – подаване на адрес и на байт, който се записва на този адрес.

Това е логическият модел на паметта. От архитектурна гледна точка паметта се реализира физически като тримерни матрици. Когато се подаде двоичен адрес, има устройство, което го дешифрира – адресът насочва устройството към онези елементи, които съдържат съответните битове. Времето за дешифрация и времето за прочитане са крайни времена. Под време за достъп до паметта разбираме времето от подаването на адреса до извличането на данните. Времето за достъп нараства с обема на паметта.

Паметта е организирана в йерархия от гледна точка на нейния обем и скорост на достъп до информация от нея. Тя се разбива на два големи класа: **основната памет** (RAM, Random Access Memory = SRAM + DRAM) и външната памет (която се организира върху магнитни ленти, магнитни барабани, магнитни дискове, CD, DVD и същевременен печат).

SRAM (Static RAM) памет, наречена още **Cache** памет – идеята е да се осъществява много бърз достъп до най-често използваните данни и команди, които да се съхраняват в кеш паметта, намираща се много близо до микропроцесора. Кешът изцяло се управлява от хардуера.



Кешът е на L1 и на L2 ниво. L1 нивото на кеша днес представлява неразделна част от централния процесор. Това не са регистрите на процесора. L2 на кеш паметта е извън ЦП и регистрите му, но достъпът между L1 и L2 е милиони пъти по-бърз, отколкото този между L2 и DRAM паметта. Тоест L1 е малка по обем памет, много бърза и много скъпа. L2 е по-

голяма по обем. Тя е по-бавна от L1 и сравнително евтина. Днес L2 паметта се прикрепя дори към управлението на самите устройства, като видеопамет, В/И система и т.н.

DRAM (Dynamic RAM) се реализира чрез една транзисторна схема, докато SRAM – с шест. DRAM с времето носи затихване на електрическия заряд и се налага през определен квант от време да се извиква т. нар. „опресняване на паметта". То е действие, което се предизвиква от кварцов итератор на квантови импулси и води до възобновяване на силата на електрическите сигнали. При SRAM такова опресняване не се налага.

Йерархия – кеш, основна и виртуална памет.

Тази схема показва **йерархията** на паметта. Колкото един вид памет е по-близо до върха, толкова тя е по-бърза, т.е. времето за достъп до нея е по-малко, но в същото време е и по-скъпа. Затова и, колкото е по-ниско в йерархията една памет, толкова е по-голяма по обем.

До регистрите има паралелен достъп от процесора. L1 кешът е разделен на две независими части – в едната се пазят само инструкции, в другата само данни. L2 кешът се намира в процесора и е част от общото адресно пространство, с което работят инструкциите. L3 кешът е извън процесора и е по-голям по обем от L2 кеша. Основната памет не съдържа всички адреси от общото адресно пространство, реално всички адреси могат да се поместят само върху диска.

Йерархията на паметта се базира на общото положение, че за малък период от време са нужни малко адреси. Нейната задача е да се реализира бърза памет чрез двете концепции за локалност:

- темпорална локалност – най-вероятно последно използваните данни ще бъдат използвани пак скоро, така всяко ниво помни последно използваните блокове от по-долното ниво;

- пространствена локалност – най-вероятно съседите в адресното пространство на последно използваните данни ще бъдат използвани скоро, така блоковете от по-ниските

нива в йерархията са по-големи, тъй като, освен че обхващат последно използваните блокове, те обхващат техните съседи.

В началото основната памет е много скъпа, докато магнитните дискове, макар по-бавни, са много по-евтини за запомняне на един бит. От друга страна програмите започват да нарастват и да се нуждаят от по-големи адресни пространства. Затова се въвежда принципът на **виртуалната памет** – работи се с виртуален адрес, който по размер е по-голям от физическия адрес на основната памет. Така виртуалното адресно пространство може да е много по-голямо по размер от реалното адресно пространство.

Странична преадресация – каталог на страниците, описател, стратегии на подмяна на страниците.

Основната концепция на виртуалната организация на паметта (ВОП) се състои в разграничаването на пространството на разработка на програма от пространството на адресите на реалната памет. Идеята е паметта за разработване на програма – виртуално адресно пространство (ВАП) – да е неограничена. От друга страна КС има една реална памет (различна за отделните модули) – физическо адресно пространство (ФАП). ВАП се разделя на фрагменти с еднакви размери, които наричаме **страници**. ФАП се разделя на фрагменти със същия размер, наричани блокове.

Апаратно или от операционната система се поддържа една таблица – **таблица на страниците**, която показва в даден момент от времето какво е изображението на виртуалната памет върху физическата, т.е. съдържа връзките между виртуалните и физическите адреси. В нея за всяка една от страниците на ВАП има по един ред, който съдържа информация за страницата, като например бит за наличност (ако е 1, то тази страница е разположена някъде в основната памет и това къде някъде се разполага в полето за номер на блок), както и dirty/modified bit (бит, който показва дали страницата е била променяна или не).

Виртуалният адрес се състои от номер на виртуалната страница и отместване в рамките на страницата. Процесът на преобразуване на виртуален адрес към реален физически адрес наричаме транслация. При транслацията номерът на виртуалната страница се преобразува в номер на физическа страница в реалната памет, отместването във физическата страница е същото като във виртуалната. Най-скорошно използваните записи от таблицата на страниците се кешират в ЦП, за да се оптимизира процеса по транслация.

Виртуалната памет има следните предимства:

- програмите могат да се изпълняват даже, когато не всичкият им програмен код или даннни са във физическата памет;
- защита – процесите са изцяло отделени един от друг;
- програмирането се опростява, тъй като всеки процес се развива в хомогенно виртуално адресно пространство с начален адрес 0.

Основен недостатък на виртуалната памет – какво става при ненамиране на физическата страница в паметта. Характерното за виртуалната памет е, че на диска се отделя файл (swap-file), в който се помества образ на пълното виртуално адресно пространство. В реалната памет се поместват само някои активни страници. При ненамиране липсващата страница трябва да се прехвърли от диска в реалната памет. Това прехвърляне е твърде

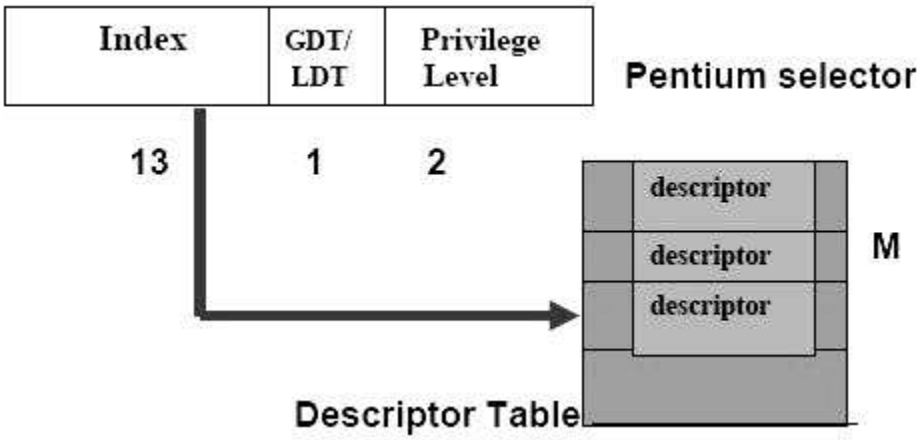
бавно и поради тази причина реалната памет винаги е write-back, т.е. никога от процесора не се записват данни в диска, така се поддържа връзка само между процесора и реалната памет.

Транслацията се извършва по следния начин: при заявка за достъп до дадена страница, първо се проверява дали тя се намира във физическата памет. Ако да – връща адреса на страницата. Ако не – зарежда търсената страница от диска и след това връща нейния адрес. При това, ако няма място в паметта за новата страница, някоя от наличните страници трябва да бъде подменена; също така, тъй като основната памет е write-back, ако подменяната страница е била променяна, тя трябва да се запише върху диска.

Сегметна преадресация – селектор, дескриптор, таблици и регистри.

Освен на страници, виртуалната памет може да бъде организирана и на **сегменти**, които за разлика от страниците са с променлива дължина и могат да се припокриват. Всеки сегмент представлява своето собствено адресно пространство. Един сегмент се състои от два компонента: базов адрес (адресът на някакво местоположение във физическата памет) и дължина (дължината на сегмента). Адресът на сегмента също се състои от два компонента: селектор на сегмента и отместване в сегмента. В машините PENTIUM при сегментацията адресът се получава от 16-битов сегментен селектор и 32-битово отместване. Всеки сегментен селектор съдържа индекс, показващ отместване в глобална или локална дескрипторна таблица (вж. по-долу за дескрипторните таблици), както и бит, показващ дали се използва ГДТ или ЛДТ.

За да бъде използван, един сегментен селектор трябва да бъде зареден в някой сегментен регистър (CS, DS, ES, SS).



Дескрипторните таблици съдържат сегментни дескриптори, като всеки сегментен дескриптор описва един сегмент чрез адреса на началото на сегмента, размера на сегмента и различни битове за състояние. Глобалната дескрипторна таблица (ГДТ) е единствена за системата и се използва най-вече за дескриптори на системни сегменти. Локалната дескрипторна таблица (ЛДТ) не е единствена, има по една за всеки процес и се използва най-вече за дескриптори на приложни сегменти. В даден момент може да се използва точно една локална дескрипторна таблица. ГДТ, освен сегментни дескриптори, може да съдържа и други неща, например дескриптори на ЛДТ. По идея една ЛДТ трябва да съдържа дескриптори на сегменти, които се отнасят до една конкретна програма, докато ГДТ съдържа глобални дескриптори.

При сегментацията, получаването на адреса по зададени сегментен селектор и отместване става по следния начин:

- първо се определя в ГДТ или ЛДТ да се търси сегментния дескриптор (в зависимост от бита GDT/LDT в сегментния селектор);
- след като е определен сегментният дескриптор, се взима адресът на началото на сегмента и към него се прибавя отместването.

Ако отместването е по-голямо от дължината на сегмента, системата сигнализира за грешка, т.е. за опит за нарушаване на сегментната защита.

Полученият адрес след сегментацията се нарича линеен адрес и той е част от линейното адресно пространство.

Възможни са два случая:

- линейното адресно пространство директно се изобразява върху физическото;
- линейното адресно пространство е виртуално – извършва се странична преадресация.

В случай на странична преадресация, линейният адрес се изпраща към блок, който го преобразува към физически адрес.

Система за прекъсване – видове прекъсвания, структура и обработка, конкурентност и приоритети, контролери на прекъсванията.

Прекъсването е процес, при които процесорът прекратява нормалното изпълнение на дадена програма, съхранява необходимата информация в стека и преминава в някакъв предварително избран адрес на паметта. След обработката на извиканата процедура, управлението се връща в изходната точка и продължава изпълнението на първоначалната програма. **Системата за прекъсване** цели във всеки момент да се даде възможност за регистрация на случващо се събитие. Механизмът на прекъсванията е ефективен начин за обмяна на информация с бавните ВИ устройства и за сигнализиция за особени състояния в работата на централния процесор. Чрез нея се избягва необходимостта от периодични проверки на флагове за дадени събития. На всяко прекъсване се съпоставя точно определен номер. За всеки вид прекъсване има специална програма, която се изпълнява при възникването на този вид прекъсване. В основната памет има специална фиксирана област, наречена таблица на векторите на прекъсванията. Всеки вектор на прекъсване съдържа указател към подпрограма за обработка на прекъсването и състояние. Тези указатели са разположени на фиксирано място в ОП (адреси от 0 до 1023 – по 4 байта за всеки от 256-те указателя).

Когато възникне прекъсване, ЦП прекъсва изпълнението на текущата програма и съдържанието на програмния брояч PC и на регистъра на състоянието MSW се запазват в стек, за да може да има вложени прекъсвания. Новото състояние на процесора при влизане в прекъсване се взима от съответния вектор на прекъсване, който се намира по номера на прекъсването. Всяка програма, която обслужва прекъсване, завършва с инструкция за връщане от прекъсване, която от върха на стека прочита предишното състояние на PC и MSW и ги зарежда в процесора и по този начин изпълнението се връща към прекъснатата програма.

Съществуват различни **видове прекъсвания**:

- по машинна грешка – грешка в апаратурата, те са най-високо приоритетни;
- входно-изходни прекъсвания – те се активират в резултат на изпълнение на входно-изходна операция;
- външни прекъсвания – свързани са с някакъв външен елемент (например бутон RESET) и са аналогични на входно-изходните прекъсвания;
- програмни прекъсвания – те са синхронни с програмата, която се изпълнява и подтискат изпълнението на някаква инструкция, например деление на 0;
- програмно-активирани (SVC) прекъсвания – при тях текущата програма издава специална команда за провеждане на прекъсване.

Друго разделяне на прекъсванията е на маскируеми и немаскируеми. Маскируемите прекъсвания могат да се игнорират, т.е. да не се обработват, докато немаскируемите прекъсвания задължително се обработват. Така немаскируемите прекъсвания винаги имат приоритет пред маскируемите.

За да се отчете важността на събитията, които може да настъпят, се въвежда **приоритет** на прекъсванията. По време на изпълнението си програмата за обработка на едно прекъсване може да бъде прекъсната само от прекъсване с по-висок приоритет. Обикновено прекъсванията с по-малки номера имат по-висок приоритет.

Ако едновременно са възникнали няколко прекъсвания, те се обработват в следния ред:

- прекъсвания от инструкцията INT и при особени случаи;
- прекъсване при стъпков режим (debugging mode);
- немаскируемите прекъсвания;
- маскируемите прекъсвания.

Управлението на прекъсванията се извършва от специална логическо устройство, което се нарича **контролер на прекъсванията**. Контролерът на прекъсванията получава заявки за прекъсвания от различните хардуерни устройства посредством линиите за заявки за прекъсвания IRQ (Interrupt Request – заявка за машинно прекъсване). Машинното прекъсване е нужно, за да се избегнат хаотичните заявки към процесора на различни външни устройства, нуждаещи се от управление или обмяна на данни, при които се губи много процесорно време. IRQ позволява бърза реакция при устройства, които се нуждаят от бърза обработка, за да не губят информация. Естествено IRQ има и недостатъци, например дадена програма може да забрани машинните прекъсвания за определено време.

По-елементарно обяснено ще изглежда така: контролерът на прекъсванията приема заявките за прекъсване на външните устройства, определя приоритетите и след това изпраща заявка за прекъсване към процесора, т.е. контролерът „решава” кое устроиство, кога и колко време ще комуникира с процесора, като ги редува. Например: видео картата е „хванала” и работи с IRQ11, това означава, че тя ще обменя информация по линия 11, след нея е примерно модемът, които ще обменя по IRQ12, после звуковата карта и т.н. Част от IRQ са резервирани за стандартни устройства на дънната платка (например IRQ0 - System timer; IRQ1 - Keyboard и т.н.), а друга част са на разположение на различни външни устройства, включени на шината (видео карта, звукова карта, вътрешен модем и др.).

11. Файлова система. Логическа и физическа организация.

Логическа организация на файлова система (ФС). Имена на файлове. Типове файлове - обикновен файл, специален файл, каталог, символна връзка, програмен канал. Вътрешна структура на файл. Атрибути на файл. Йерархична организация на ФС - абсолютно и относително пълно име на файл, текущ каталог. Физическа организация на ФС. Стратегии за управление на дисковото пространство. Системни структури, съдържащи информация за разпределението на дисковата памет и съхранявани постоянно на диска: за свободните блокове; за блоковете, разпределени за всеки един файл; за общи параметри на ФС. Примери за физическа организация на ФС: UNIX System V; LINUX; MS DOS; NTFS.
Забележки: За изпита ще бъдат избрани два от изброените примери за файлова организация.

Файлова система (ФС) това е тази компонента на операционната система, която е предназначена да управлява постоянните обекти данни, т.е. обектите, които съществуват по-дълго отколкото процесите, които ги създават и използват. Постоянните обекти данни се съхраняват на външна памет (диск или друг носител) в единици, наричани файлове.
Файловата система трябва да:
• осигурява операции за манипулиране на отделни файлове, като например create, open, close, read, write, delete и др.
• изгражда пространството от имена на файлове и да предоставя операции за манипулиране на имената в това пространство.

ЛОГИЧЕСКА СТРУКТУРА НА ФАЙЛОВА СИСТЕМА

➢ ИМЕНА И ТИПОВЕ ФАЙЛОВЕ

Най-често името на файл е низ от символи с определена максимална дължина, като в някои системи освен букви и цифри са разрешени и други символи. Много често името се състои от две части, разделени със специален символ, например ".". Втората част се нарича разширение на името и носи информация за типа или формата на данните, съхранявани във файла. Например, следните имена имат разширения, показващи типа на данните във файла.

`file.p` - програма на Паскал
`file.c` - програма на Си
`file.o` - програма в обектен код
`file.exe` - програма в изпълним код
`file.txt` - текстов файл

В някои операционни системи, като UNIX и LINUX, такива разширения представляват съглашения, които се използват от потребителите и някои обслужващи програми (транслатори, свързващи редактори, текстови процесори и др.), но ядрото не ги налага и използва. В други операционни системи се реализира по-строго именуване. Например, няма да бъде зареден и изпълнен файл, ако името му няма разширение "`.exe`" или някое друго.

Какво включва пространството от имена или какви са типовете файлове? Преди всичко то включва имена на ***обикновени файлове*** (***regular files***), съдържащи програми, данни, текстове или каквото друго потребителят пожелае. Но пространството от имена би могло да включва и имена на външни устройства, системни структури и услуги. Външните устройства са специален тип файлове, наречени ***специални файлове*** (***character special u block special device file***) в UNIX, LINUX, MINIX и др. Системните примитиви на файловата система са приложими както към обикновените файлове така и към специалните файлове. Следователно, всяка операция за четене или писане, осъществявана от програмата, е четене или писане във файл. Най-съществено предимство на този подход е, че позволява да се пишат програми, които не зависят от устройствата, тъй като действията, изпълнявани от програмата зависят само от името на файла.

Трябва да се съхранява информация за файловете във файловата система и за тази цел се използват специални системни структури, наричани ***каталог***, ***справочник***, ***директория***, ***directory***, ***folder***, ***VTOC***, които осигуряват връзката между името и данните на файла и реализират организацията на файловете.
Много системи каталогът е тип файл с фиксирана структура на записите и съдържа по един запис за всеки файл.

И така, типовете файлове, поддържани от файловите системи на UNIX, LINUX, MINIX и др. са:

- обикновен файл
- каталог
- специален файл
- програмен канал, FIFO файл
- символни връзки

Типът ***програмен канал (pipe)*** и ***FIFO файл*** се използва като механизъм за комуникация между конкурентни процеси. Чрез ***символните връзки*** (***symbolic link, soft link, junction***) един файл може да има няколко имена, евентуално в различни каталози, или както се казва реализират се няколко връзки към файл. Този тип файл е един от механизмите за осигуряване на общи файлове за различните потребители. Друг начин за работа с общи файлове са ***твърдите връзки*** (***hard links***), но те не са тип файл, а са няколко имена на обикновен файл.

➢ ВЪТРЕШНА СТРУКТУРА НА ФАЙЛ

Файлът най-често представлява последователност от обекти данни. Възможни са два вида данни - запис или байт.

Файлът е последователност от записи с определена структура и/или дължина. Основното в този подход е, че всеки системен примитив `read` или `write` чете или пише един запис.

Другата възможност, реализирана в много от съвременните операционни системи, UNIX, LINUX, MINIX, MSDOS, Windows и др., е последователност от байтове.

➢ АТРИБУТИ НА ФАЙЛ

Всеки файл има име и данни. В допълнение операционната система може да съхранява и друга информация за файла, която ние ще наричаме ***атрибути*** на файла (наричат ги също така ***метеданни***). Атрибутите, реализирани в различните системи се различават, но един списък от възможни атрибути е показан на
Таблица. 1.

Таблица 1. Някои възможни файлови атрибути

Размер	Текущия размер на файла в байтове, блокове или записи.
Време на създаване	Дата и време на създаване на файла
Време на достъп	Дата и време на последен достъп до файла
Време на изменение	Дата и време на последно изменение на файла
Собственик	Потребителят, който е текущия собственик на файла.
Права на достъп	Кой и по какъв начин може да осъществява достъп до файла.
Парола за достъп	Парола за достъп до файла
Флагове:	
Read-only флаг	1 - само за четене, 0 - за четене и писане
Hidden флаг	1 - не се вижда от командите, 0 - видим за командите
System флаг	1 - системен файл, 0- нормален файл
Archive флаг	1 - трябва да се архивира, 0 - не е променян след архивирането
Secure deletion	При унищожаване на файла блоковете, заемани от него се форматират.

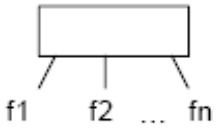
➢ ЙЕРАРХИЧНА ОРГАНИЗАЦИЯ НА ФС

Каталогът съдържа по един запис за всеки файл, който съдържа поне името на файла. Освен това може да съдържа атрибутите на файла и дисковите адреси на данните на файла или указател към друга структура, където се съхраняват дисковите адреси на данните и евентуално атрибутите на файла.

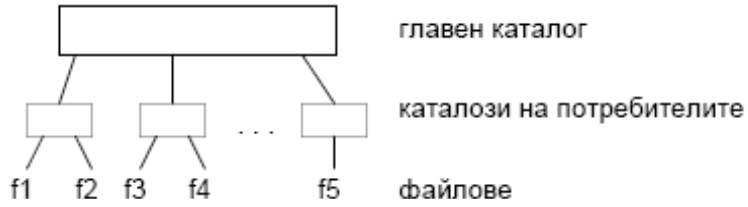
Като правило файловата система е разположена върху няколко носителя и включва файлове, принадлежащи на различни потребители.

- Еднокаталогова файлова система - На всеки носител има по един каталог, който съдържа информация за всички файлове върху носителя. Такава е организацията в някои ранни операционни системи или в

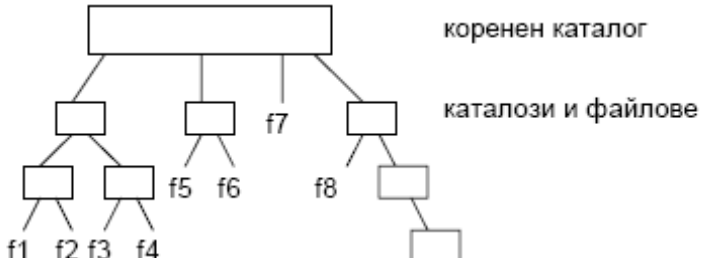
по-нови, но примитивни. Преимущество на тази организация е простотата и възможността за бързо търсене на файл .



- Йехархична структура с фиксиран брой нива - Всеки потребител има свой каталог, който съдържа записи за всички файлове на потребителя. Има главен каталог, който съдържа по един запис за каталозите на потребителите. Тогава имената, които потребителят дава на своите файлове, трябва да са уникални в рамките на неговия каталог



- Йерархична структура с произволен брой нива - Вътрешните възли на дървото трябва да са каталози, а листата могат да бъдат каталози, обикновени файлове, специални файлове и други типове фалове, ако се поддържат такива. За потребителя каталогът представлява група от файлове и каталози (подкаталози).



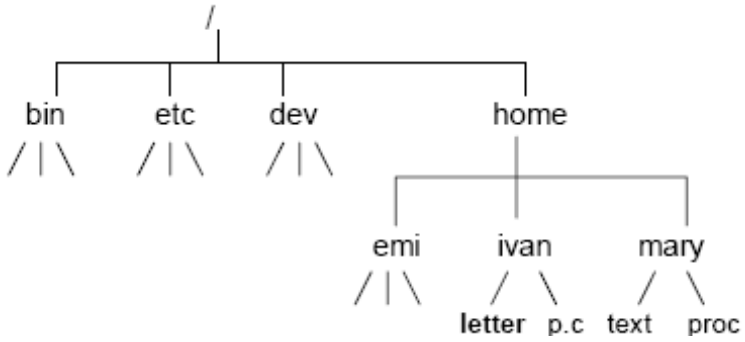
➢ АБСОЛЮТНО И ОТНОСИТЕЛНО ИМЕ НА ФАЙЛ; ТЕКУЩ КАТАЛОГ

Всеки връх в дървото, каталог или друг тип файл, има име, което потребителят избира да е уникално в рамките на съдържащия го каталог и което ние ще наричаме ***собствено име***. Тогава всеки файл ще има име, което уникално го идентифицира в рамките на цялата йерархична структура и ние ще го наричаме ***пълно име***. Използват се два начина за формиране на пълно име:

• ***абсолютно пълно име*** (***absolute path name***) - Всеки файл или каталог притежава едно абсолютно пълно име, което съответства на единствения път в дървото от корена до съответния файл или каталог.
• ***относително пълно име*** (***relative path name***) - Този начин за формиране на пълно име е свързан с понятието ***текущ или работен каталог*** (***current/working directory***). Във всеки един момент от работата на потребителя със системата, той е позициониран в един от каталозите на дървото. Относителното пълно име на файл или каталог съответства на пътя в дървото от текущия в даден момент каталог до съответния файл или каталог. Като в този случай е разрешено и движение нагоре по дървото.

Пълното име на файл, абсолютно или относително, се състои от компоненти - собствените имена на каталозите в пътя и името на самия файл, разделени със специален символ-разделител, като напр. ">", "\", "/". Движението нагоре по дървото се записва чрез специално име, което обикновено е "..". Ако първият символ в

пълното име е символа-разделител, това означава, че името е абсолютно. Всяко име, което не започва със символа-разделител, се приема за относително. За илюстрация да разгледаме една типична йерархия на файлова система в UNIX:



Абсолютно пълно име на файл е /home/ivan/letter. Ако текущ каталог е /home, то относителното пълно име на същия файл е ivan/letter. Ако текущ каталог е /home/emi, то относителното пълно име на същия файл е ../ivan/letter

ФИЗИЧЕСКА ОРГАНИЗАЦИЯ НА ФАЙЛОВА СИСТЕМА

➢ СТРАТЕГИИ ЗА УПРАВЛЕНИЕ НА ДИСКОВАТА ПАМЕТ

Стратегията за управление на дисковата памет трябва да даде отговор на два въпроса.

1. *Кога се разпределя дискова памет за файл?* Едната възможност е това да се прави статично (предварително) при създаването на файла, а другата е да се прави динамично при нарастване на файла.
2. *Колко непрекъснати дискови области може да заема един файл*? Дали за файла се разпределя една (или малко на брой) непрекъсната дискова област с нужния размер или за файла могат да се разпределят много порции дискова памет, които не е задължително да са физически съседни, а броят им най-често се ограничава от капацитета на диска.

Най-често реализираните стратегии са две:

- **Статично и непрекъснато разпределение**

За файл с размер n байта се отделя една непрекъсната област от диска, в която могат да се съхранят всичките n байта и това се прави при създаването на файла. За тази цел обаче, системата трябва да има предварителна информация за максималния размер на файла, който той може да достигне по време на съществуването си. Естествено тази информация може и трябва да се осигури от потребителя по време на създаване на файла и тогава системата ще разпредели дискова памет за файла. Преимуществото, което този метод дава, е високата производителност на системата при последователна обработка на файла, тъй като последователните байтове на файла са разположени в съседни сектори, писти и цилиндри на диска.

Проблем може да възникне при **нарастване** на файла, което е нещо обичайно, ако размерът на файла надмине разпределената му предварително памет. Едно възможно решение на този проблем е разпределяне на нова непрекъсната област с по-голям размер и преместване на файла в новата област. Това обаче не е добро решение, тъй като операцията може да се окаже бавна и скъпа при големи файлове. Друго решение е да се направи компромис при искането за непрекъснатост на дисковата памет, т.е. един файл да може да заема не една, а няколко, но малко на брой непрекъснати области от диска с размери, заявени от потребителя.

Друг недостатък на тази стратегия е известен като **фрагментация** на свободната дискова памет. Това означава съществуване на достатъчно свободни участъци дисковата памет, които обаче не са съседни и поради това не може да бъде удоволетворена заявка за първично или вторично разпределение на памет за файл, тъй като се изисква непрекъснатост.

Тази стратегия се оказа приложима и в съвремието при CD-ROM, където размера на файла е предварително известен.

- **Динамично и поблоково разпределение**

Файлът се дели на части с фиксирана дължина, които ще наричаме **блокове**. Последователните блокове на файла при записването им на диска не е задължително да са физически съседни. Това дава възможност дискова памет за файл да се разпределя по блокове тогава, когато е необходима, т.е. динамично при нарастване на файла. Така се решава и проблема с нарастване на файла и проблема с фрагментацията на

свободната дискова памет, тъй като памет се разпределя динамично по блокове, които са с еднакъв фиксиран размер.

При прилагане на тази стратегия трябва се избере **размер на блока -** който най-често е 1К байта, 2К байта, 4К байта. При по-нататъшното изложение ще предполагаме, че се използва динамично и поблоково разпределение на дискова памет, както е в повечето съвременни системи. Единицата за разпределение на дискова памет ще наричаме блок. *Следователно, за файловата система дисковото пространство е последователност от блокове с адреси (номера) 0,1,2, ...N.*

➢ СИСТЕМНИ СТРУКТУРИ, СЪДЪРЖАЩИ ИНФОРМАЦИА ЗА РАЗПРЕДЕЛЕНИЕТО НА ДИСКОВАТА ПАМЕТ, т.е. за СВОБОДНИТЕ БЛОКОВЕ, БЛОКОВЕТЕ РАЗПРЕДЕЛЕНИ ЗА ВСЕКИ ЕДИН ФАЙЛ И ОБЩИ ПАРАМЕТРИ НА ФС

Файловата система трябва да съхранява информация за разпределението на дисковата памет, а именно за *свободните блокове* и за *блоковете разпределени за всеки един файл*. Под системни структури тук ще разбираме структури, съхраняващи такава информация постоянно на диска.

Информация за всички свободни блокове трябва да бъде съхранявана, тъй като само по съдържанието на блока не може да се отличи свободния от заетия блок. Някои възможни структури данни, използвани за тази цел са следните.

- **Свързан списък на свободните блокове**

Всички свободни блокове са организирани в едносвързан списък, т.е. първата дума от всеки свободен блок съдържа адрес на следващ свободен блок. Недостатък на този метод е, че системната структура, т.е. свързаният списък, е пръсната по всички свободни блокове, което крие опасности за повредата й при системни сривове.

- **Свързан списък на блокове с номера на свободни блокове**

Свободните блокове се групират и номерата на блоковете от една група се записват в първия свободен блок. Следователно информацията се съхранява в едносвързан списък от дискови блокове, които съдържат номера на свободни блокове. Самите блокове от списъка вече не са свободни за разлика от предходната структура. Такава структура е по-компактна, големината й е пропорционална на свободното дисково пространство и позволява разработката на ефективни алгоритми за разпределение на блокове. Този подход е използван във файловата система на UNIX System V (s5fs).

- **Карта или таблица**

Структурата представлява масив от елементи, като всеки елемент съответства позиционно на блок от диска, т.е. на блок с номер равен на индекса на елемента в масива. Съдържанието на всеки елемент описва състоянието на блока. В най-простия случай може да се помнят две състояния - свободен и зает. Тогава елемент от картата може да е просто бит. Ако битът е 1, то съответният блок е свободен, а ако е 0 е зает (разпределен за файл или друга структура на диска) или обратното. Такава системна струтура се нарича ***битова карта (bit map).*** Преимущество на битовата карта е, че е компактна и с фиксиран размер, а при разпределяне на памет позволява да се отчита физическото съседство на блоковете. Битова карта е използвана във файловите системи на MINIX, LINUX , OS/2 (HPFS) и Windows (NTFS).

Друг тип информация, която трябва да бъде съхранявана, е за дисковата памет, разпределена за всеки един файл. Всеки файл от гледна точка на физическото му представяне представлява последователност от блокове, съдържащи последователните му данни, като последователните блоковете на файла може да не са физически последователни. Следователно файловата система трябва да съхранява информация за блоковете, разпределени за всеки един файл в съответната логическа последователност. Ще разглеждаме някои възможни структури за тази цел.

- **Свързан списък на блоковете на файла**

Всеки блок на файла ще съдържа в първата си дума номер на следващ блок на файла и данни в останалата част. Основният недостатък на тази структура е неефективната реализация на произволен достъп до файла. За да се прочете произволен байт от файла системата трябва да прочете всички предходни блокове в списъка докато стигне до данните, искани от програмата. Друг недостатък е, че адресната информация е пръсната по диска, а това прави файловата система неустойчива при повреди. И още един недостатък, който в някои случаи е критичен, е че броят на байтовете за данни в блока вече не е степен на 2.

- **Карта или таблица**

Същността на проблема при предходната реализация се състои в това, че адресите и данните се съхраняват заедно. Идеята на картата (таблицата) на файловете е свързаните списъци от номера на блокове за всички файлове на диска да се съхраняват в една структура отделно от данните. В същност това може да е същатата системна структура карта, в която се съхранява информация за свободните блокове и която описахме по-горе, като елемент на масива е достатъчно голям, така че да позволи съхраняване на следните състояния на блок - свободен, повреден, а ако е зает да съдържа номера на следващия блок на файла. Тази структура е използвана в MSDOS, където се нарича таблицата ***FAT (File Allocation Table).*** По-нататък по-подробно ще разгледаме физическото представяне на файловата система в MSDOS.

- **Индекси**

Основният недостатък на предишния подход е, че информацията за всички файлове се съхранява в една структура, което при големи дискове създава проблеми. При големи дискове естествено голяма става и картата на файловете, а тя трябва цялата да се зарежда и съхранява в оперативната памет по време на работа дори ако е отворен само един файл. Следователно по-добре би било ако списъците на различните файлове се съхраняват в отделни структури и тогава в паметта ще се зарежда само информацията за отворените файлове.

Структурата, в която се съхранява информацията за блоковете, разпределени за определен файл се нарича индекс. Всеки индекс съдържа адресите на дисковите блокове, разпределени за един файл, като наредбата на адресите отразява логическата наредба на блоковете във файла.

Една възможна реализация на индекса е ***индексен списък*** в XINU. Индексният списък е едносвързан списък от индексни блокове. Индексните блокове са с размер, различен от този на блоковете за данни и се намират в област на диска, отделна от областта на блоковете за данни, т.е. адресното пространство за индексните блокове е различно от това на дисковите блокове. Всеки индексен блок, освен адрес на следващ индексен блок, съдържа и определен брой адреси на дискови блокове с данни, като наредбата на адресите отразява логическата наредба на блоковете с данни във файла.

Друга възможна реализация е чрез дърво.Такъв подход е използван в UNIX, LINUX и MINIX, където структурата се нарича ***индексен описател*** (***i-node*** от ***index node***) и при големи файлове е корен на дърво, включващо и косвени блокове.

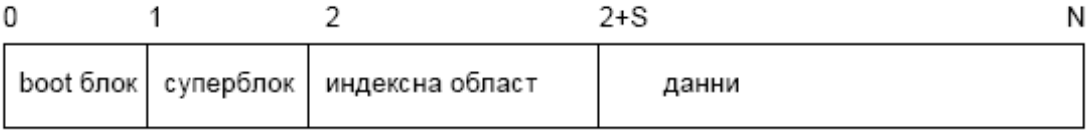
Друг тип информация, която трябва се съхранява, са *общи параметри на физическата структура на файловата система*. Обикновено системната структура се нарича ***суперблок***. В суперблока се съхраняват общи параметри, като:

1 - размер на файловата система (максимален номер на блок на тома)
2 - размер на блок
3 - размери и адреси на различни области от диска, съдържащи системни структури, напр. размер на индексната област, на битовата карта, на FAT
4 - общ брой свободни блокове на диска и други ресурси, напр. индексни описатели.
5

ПРИМЕРИ ЗА ФИЗИЧЕСКА ОРГАНИЗАЦИЯ НА ФАЙЛОВИ СИСТЕМИ

➢ UNIX System V (s5fs)

Всеки диск се състои от един или няколко дяла (partitions). Дяловете се разглеждат като независими устройства (на всеки дял съответства различен специален файл). За файловата система дисковото пространство на всеки диск или дял от диск е последователност от блокове с фиксиран размер и адресирани с номера 0, 1, 2,... N. При създаване на празна файлова система с командата mkfs, дисковото пространство на всеки дял се разделя на четири области



Блок с номер 0, наричан boot блок, съдържа програма за първоначално зареждане на операционната система. Използва се само в коренната файлова система, но заради еднотипността присъства и в другите файлови системи.

Блок с номер 1 е наречен суперблок, тъй като съдържа общи параметри на физическата структура на файловата система.

От блок 2 започва индексната област, където се съхраняват индексните описатели (i-node) на всички файлове. Размерът на тази област S блока е функция от общия размер на файловата система N и трябва внимателно да бъде изчислен така, че да не ограничава броя на файловете на диска. Този размер се задава като параметър при изграждане на празна файлова система с командата `mkfs` и не може да бъде променян след това.

В последната област - данни, се съхраняват блоковете на всички файлове и каталози, косвените блокове, блоковете от списъка на свободните блокове и евентуално някои блокове са свободни.

Индексни описатели

Всеки файл от произволен тип има точно един индексен описател, независимо от това в кой и колко каталога е включен и под какви имена. Индексните описатели се номерират от 1 и номерът съответства на позицията му в индексната област. Индексният описател е с размер 64 байта и има следната структура:

```
struct dinode
{
ushort di_mode; /* mode and type of file */
short di_nlink; /* number of links to file */
ushort di_uid; /* owner's user id */
ushort di_gid; /* owner's group id */
off_t di_size; /* number of bytes in file */
char di_addr[40]; /* disk block addresses */
time_t di_atime; /* time last accessed */
time_t di_mtime; /* time last modified */
time_t di_ctime; /* time last changed */
};
```

В полето `di_mode` се съхранява типа на файла в старшите 4 бита и кода на защита на файла в останалите 12 бита.

Полето `di_nlink` съдържа броя на твърдите връзките или имената на файла, т.е. колко пъти в записи на каталози е цитиран този номер на индексен описател.

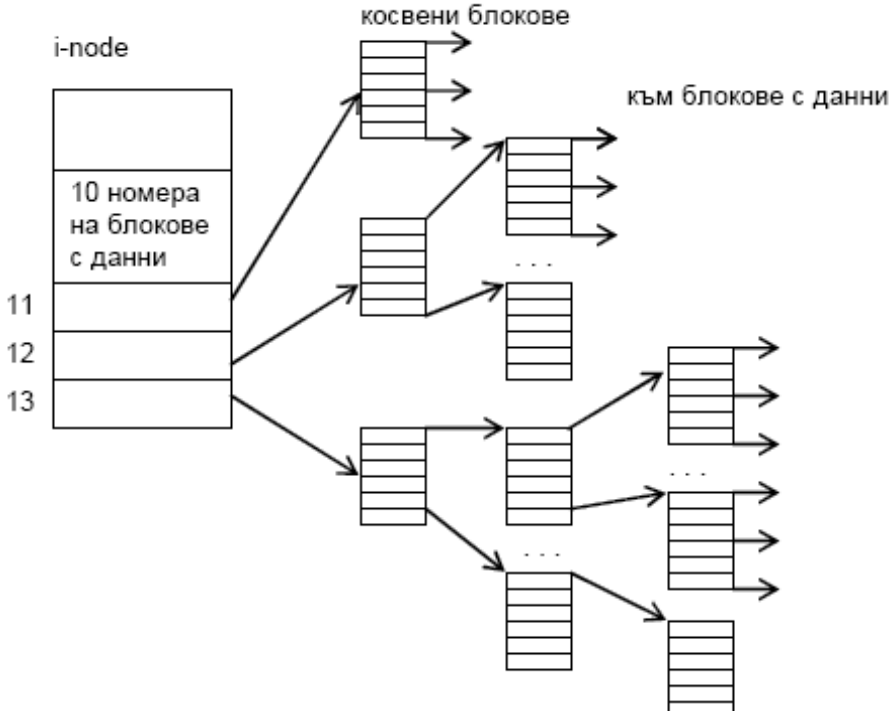
Полетата `di_uid` и `di_gid` определят собственика и групата на файла.

Полето `di_size` при обикновени файлове и каталози съхранява размера на файла в брой байтове.

В полетата `di_atime, di_mtime и di_ctime` се записват дата и време, съответно на последен достъп, последно изменение и последно изменение на i-node на файла.

Ако файлът е специален в полето `di_addr` се съхранява номера на устройството, на което съответства специалния файл.

За другите типове файлове в полето di_addr се съхраняват 13 дискови адреса (номера на блокове от областта за данни) всеки в 3 байта. Първите 10 адреса са директни адреси на първите 10 блока с данни на файла. Ако файлът стане по-голям от 10 блока, тогава се използват косвени блокове. Косвените блокове се намират в областта за данни, но съдържат номера на блокове, а не данни на файла. Единадесетият адрес съдържа номер на косвен блок, който съдържа номерата на следващите блокове с данни на файла. Това се нарича единична косвена адресация. Чрез дванадесетия адрес се реализира двойна косвена адресация, т.е. там е записан номер на косвен блок, който съдържа номера на косвени блокове, които вече съдържат номера на блокове с данни. За представяне на много големи файлове се използва тринадесетия адрес и тройна косвена адресация



Представянето на файловете в UNIX системите съчетава следните предимства: малък индексен описател с възможност за представяне на големи файлове при осигуряване на бърз достъп до произволен байт от файла. И при най-големи файлове за достъп до произволно място във файла са необходими най-много три допълнителни достъпа до диска за трите нива на косвени блокове (индексният описател се зарежда в паметта при отваряне на файла и се съхранява там до затваряне).

Суперблок

Суперблокът съдържа изключително важна информация, характеризираща физическата структура на файловата система. Основните данни, съдържащи се в него, са описани в следната структура:

```
struct filsys
{
      ushort s_isize; /* size in blocks of i-list */
      daddr_t s_fsize; /* size in blocks of entire volume */
      short s_nfree; /* number of addresses in s_free */
      daddr_t s_free[NICFREE]; /* free block list */
      short s_ninode; /* number of i-nodes in s_inode */
      ushort s_inode[NICNODE]; /* free i-node list */
      char s_flock; /*lock during free list manipulation*/
      char s_ilock; /* lock during i-list manipulation */
      char s_fmod; /* super block modified flag */
      char s_ronly; /* mounted read only */
      . . .
      daddr_t s_tfree; /* total free blocks */
      ushort s_tinode; /* total free i-nodes */
      . . .
};
```

Полето `s_isize` съдържа дължината на индексната област в блокове, която се задава и зарежда в полето при създаване на файловата система (параметъра S)

Полето `s_fsize` съдържа максималния номер на блок във файловата система и също както `s_isize` се задава и зарежда при изграждане на файловата система (параметъра N)

Полето `s_nfree` съдържа брой свободни блокове, чиито номера са записани в масива `s_free`.

Полето `s_ninode` съдържа брой свободни индексни описатели, чиито номера са записани в масива `s_inode`.

Полетата `s_flock` и `s_ilock` се използват като флагове за заключване на достъпа до списъка на свободните блокове и свободните индексни описатели по време на манипулирането им с цел избягване на състезание между конкурентни процеси.

Полетата `s_tfree` и `s_tinode` съдържат общия брой свободни блокове и свободни индексни описатели на диска.

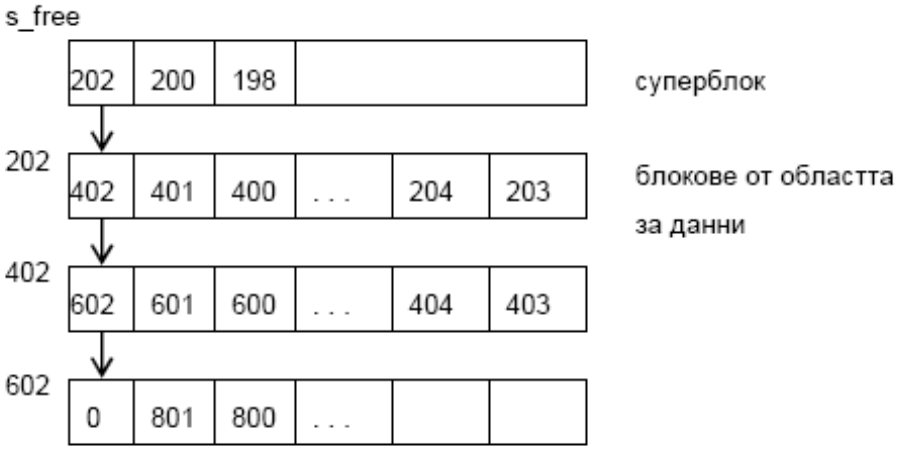
Суперблоковете на коренната и всички монтирани файлови системи се зареждат в оперативната памет в таблицата на суперблоковете и остават там през цялото време на работа на системата. Това осигурява бърз достъп до най-важните характеристики на файловите системи и се използва от алгоритмите за разпределяне и освобождаване на ресурсите - блокове и индексни описатели.

Заб: От анотацията на въпроса не става ясно дали трябва да се включват въпорсните алгоритми за разпределение и освобождаване на блокове и индексни описатели.Алгоритмите и отбелязаните по-долу структури трябва ли да се включват?

Списък на свободните блокове

Всички блокове от областта за данни, които не се разпределени за файлове, каталози, за косвени блокове, т.е не съдържат данни на файловата система са свободни. Но файловата система не е в състояние да отличи свободния от заетия блок само по съдържанието му. Затова информацията за всички свободни блокове се съхранява в специална структура - *едносвързан списък от блокове*, които съдържат номера на свободни блокове. Масивът s_free в суперблока представлява глава на списъка. Останалите елементи от списъка на свободните блокове, които са дискови блокове, са разположени в областа за данни. На Фиг. 8 е изобразен примерен списък на свободните блокове.

Задължителни ли са?



Фиг. 8. Списък на свободните блокове в s5fs на UNIX

Списък на свободните индексни описатели

Другият ресурс, който файловата система трябва да управлява, са индексните описатели и те са разположени в отделно пространство - индексната област. При създаване на файл за него трябва да се разпредели свободен индескен описател, а при унищожаване на файл индексния му описател трябва да се отбележи като свободен.

В суперблока е разположен масив s_inode, в който са записани известен брой номера на свободни индексни описатели. За разлика от управлението на блоковете, този масив не продължава в свързан списък от блокове. Това означава, че освен номерата в масива s_inode в индексната област може да има и други свободни индексни описатели. Това не създава проблеми при управлението им, тъй като ядрото може да различи свободния от заетия i-node по полето di_mode - битовете за тип са 0 в свободен i-node. В същност би могло и без масива s_inode, т.е. когато е необходим свободен i-node ще се търси в индексната област. Но такъв алгоритъм би бил неефективен, ако всеки път когато се създава файл се обхождат блоковете на индексната област в търсене на свободен i-node. Затова е въведен масива s_inode, в който са кеширани известен брой номера на свободни индексни описатели, но структурата не продължава в пълен списък.

Каталози

Каталозите са тип файлове, които осигуряват връзката между името и данните на файл и йерархичната структура на файловата система. В разглежданата s5fs файлова система записът в каталога има следната структура:

```
struct direct
{
ushort d_ino;
char d_name[DIRSIZE];
};
```

Полето d_name съдържа собственото име на файл или подкаталог, а полето d_ino съдържа номера на индексния му описател. Във всеки каталог има два стандартни записа, в единия полето d_name съдържа "..", а d_ino номера за родителския каталог, в другия полето d_name съдържа ".", а d_ino номера за самия каталог. Тези два записа присъстват и в празен каталог и са част от представянето на дървовидната структура на файловата система. Някои записи може да са празни. Номер 0 в полето d_ino означава, че записът е освободен при унищожаване на файл.

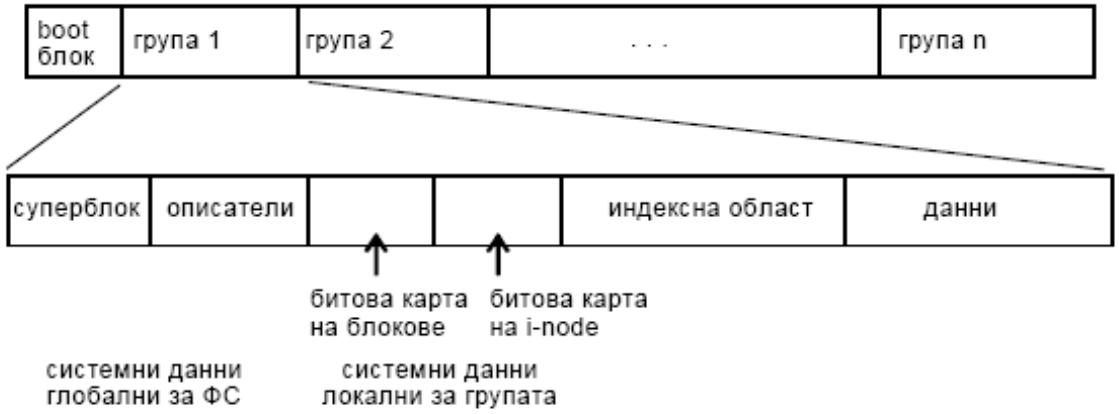
Първите няколко индексни описатели са резервирани за някои важни файлове, като коренния каталог, файла на лошите блокове и др. Така номерът на i-node на коренния каталог е известен, а самия каталог е разположен в произволни блокове от областта за данни

Заб: Трябва ли да включа по-подробно описание за каталози – в лекциите има пример? Също така трябва ли да се включват разяснения за твърдите и символните връзки?

Разгледаната файлова система s5fs има много преимущества, които отбелязахме в хода на разглеждането ѝ, но има и слаби места. От гледна точка на надеждността слабо място е суперблока, за който се съхранява само едно копие. За производителността на системата е критично, че индексните описатели са разположени в началото на файловата система и са далеч от данните на файла. Индексната област е с фиксиран размер, който се задава при създаване на файловата система, и неподходящия му избор може да доведе до липса на свободни индексни описатели при наличие на свободно дисково пространство.

➢ LINUX

Основната файлова система в LINUX е Extended File System-version 2 (ext2), а вече и ext3. При управлението на дисковото пространство се използват блокове с фиксиран размер - 1KB, 2KB или 4KB. Това е и адресуемата единица на диска, единна за всички файлови системи. Фиг.13 представя областите, в които е организирано дисковото пространство.



Фиг.13. Разпределение на дисковото пространство в LINUX

Всяка група съдържа част от файловата система и копие на глобални системни структури, критични за цялостността на системата - суперблока и описателите на групите. Ядрото работи със суперблока и описателите от първата група. Когато се извършва проверка на непротиворечивостта на файловата система (изпълнява се командата fsck), ако няма повреди, то двете системни структури от първата група се копират в останалите групи. Ако се открие повреда, тогава се използва старо копие на системните структури от друга група и обикновено това позволява да се ремонтира файловата система.

Битови карти

Битовите карти описват свободните ресурси - блокове и индексни описатели в съответната група. Значение 0 означава свободен, а 1 използван блок или индексен описател. Размерът на групите е фиксиран и зависи от размера на блок, като стремежът е битовата карта на блоковете да се събира в един блок.

Индексни описатели

Индексните описатели са разпределени равномерно във всички групи, но се адресират в рамките на файловата система. Структурата на индексния описател в ext2 и ext3 е разширение на i-node от UNIX с нови полета. Размерът е 128 байта. Адресните полета са 12+1+1+1, т.е. използва се косвена адресация на три нива, но броят на директните адреси е увеличен с два, т.е. е 12. Освен това адресните полета вместо по 3 байта са по 4 байта

Добавени са нови атрибути на файл, например:

1 - размер на файла в брой блокове по 512 байта (i_blocks)

2 - още едно четвърто поле за дата и време (i_dtime)

3 - флагове:

immutable - Файлът не може да се изменя, унищожава, преименува и да се създават нови връзки към него (дори от администратора).

append only - Писането във файла е винаги добавяне в края му.

synchronous write - Системният примитив write завършва след като данните са записани на диска.

secure deletion - При унищожаване на файла блоковете му се форматират.

undelete - При унищожаване на файла съдържанието му се съхранява за евентуално възстановяване впоследствие.

compress file - При съхраняване на файла ядрото автоматично го компресира.

Някои от добавените атрибути са за бъдещо планирано развитие на файловата система.

В този индексен описател има две полета за размер на файл i_size и i_blocks. Файлът се съхранява в цяло число блокове и сл., в общия случай i_size<= 512*i_blocks. Но е възможно и обратното - i_size> 512*i_blocks, ако файлът съдържа „дупка”.

Описатели на групи блокове (Group descriptors)

Всяка група е описана чрез запис, с размер 32 байта съдържащ:

1 - адрес на блок с битовата карта на блоковете за групата;

2 - адрес на блок с битовата карта на индексните описатели за групата;

3 - адрес на първи блок на индексната област в групата;
4 - брой свободни блокове в групата;
5 - брой свободни i-node в групата;
6 - брой каталози в групата;
7 - резервирано поле от 14 байта.

Описателите на всички групи са събрани в областта описатели, копие на която се съдържа във всяка група.

Суперблок

Суперблокът съдържа общите параметри на физическата структура на файловата система. Някои от основните данни, съдържащи се в него, са следните:
1 - общ брой блокове - размер на файловата система;
2 - общ брой индексни описатели;
3 - брой блокове резервирани за администратора (обикновено е 5% от общия брой блокове). Тези резервирани блокове позволяват на администратора да продължи работа, дори когато няма свободни блокове за другите потребители.
4 - общ брой свободни блокове и индексни описатели;
5 - размер на блок;
6 - брой блокове в група;
7 - брой i-node в група;
8 - полета използвани за автоматична проверка на файловата система (fsck) при boot, напр., дата и време на последен fsck, брой монтирания след последния fsck, максимален брой монтирания преди следващ fsck, максимален интервал време между две изпълнения на fsck.

Каталог

Ограничението за максимална дължина на името на файл е 255 символа, затова записите в каталога са с променлива дължина и съдържат:
1 - номер на i-node;
2 - дължина на записа;
3 - дължина на името на файла;
4 - име на файла, съхранявано в толкова байта, колкото са небходими.
5
И така, новото във физическото представяне на ext2 и ext3 в сравнение с файловата система в UNIX е:

• Използване на битови карти при управление на свободните ресурси - блокове и i-node.

• Разделяне на дисковото пространство на групи блокове.

И двете промени, както и съответните промени в стратегиите и алгоритмите, имат за цел да се постигне по-висока степен на локалност на файловете и системните им структури (да се намали разстоянието между i-node и блоковете на файл), а от там и по-добра производителност.

• За системните структури, съдържащи критична за файловата система информация се съхраняват няколко копия.

➢ MSDOS

За файловата система на MSDOS дисковото пространство е последователност от сектори по 512 байта. Броят им зависи от типа на диска. Твърдите дискове могат да се разделят на дялове (partitions). Във всеки дял от диск или цял диск (ако не е разделян или е дискета) се изгражда независима файлова система, наричана том (volume). На Фиг. 14 е изобразено разпределението на дисковото пространство в една файлова система, изградена на един том.

boot сектор	FAT	FAT	Коренен каталог	данни

Фиг.14. Разпределение на дисковото пространство в том на MSDOS

Boot сектор съдържа програмата, която стартира зареждането на операционната система в паметта и параметри на тома, като размер на клъстер, размер на тома, размер и брой копия на FAT, размер на коренния

каталог. Тази област присъства на всички томове, дори и да не са системни, но при опит да се стартира системата от несистемен диск се извежда съобщение за грешка.

Следващата област е **FAT** (File Allocation Table) или Таблица за разпределение на дисковото пространство. Тя представлява карта на файловете и съдържа цялата информация за дисковата памет, разпределена за файловете, за свободното дисково пространство и за дефектните сектори. От гледна точка на файловата система това са най-важните и критични данни на диска, затова се пазят две копия на FAT, разположени едно след друго.

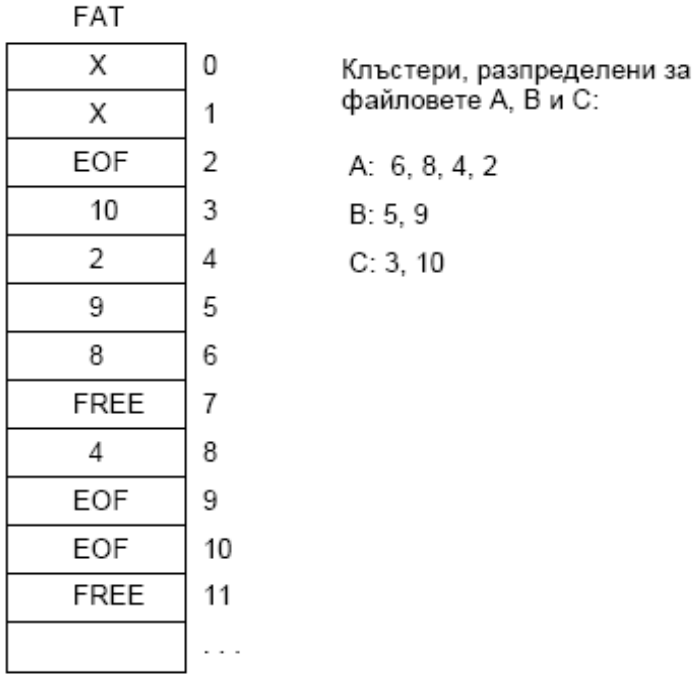
Следва областта, в която се съхранява коренния каталог на файловата система. Файловата система в MSDOS е йерархична, но за разлика от UNIX, тук всеки том се разглежда като независима дървовидна структура. По тази причина, може би, коренният каталог на всеки диск е разположен в отделна област, след FAT.

В областта данни се съхраняват всички файлове и каталозите, различни от коренния.

FAT - Таблица за разпределение на дисковото пространство

Дискова памет за файлове се разпределя динамично и единицата за разпределение се нарича клъстер (cluster). Клъстерът е последователност от 1 или повече сектора (обикновено степен на двойката брой сектори) и всички клъстери на един том са с еднакъв размер. За различните томове в една система клъстерите може да са с различни размери. (Заб: Клъстерът е аналога на блок)

FAT съдържа елементи (с дължина 12 или 16 бита), като броят им зависи от размера на тома и от размера на клъстера. Елементи 0 и 1 съдържат код, идентифициращ формата на тома. Всеки от останалите елементи съответства позиционно на клъстер от областта за данни. За удобство номерацията на клъстерите започва от 2, т.е. първият клъстер в областта за данни е с номер 2. Съдържанието на елемент от FAT определя състоянието на съответния клъстер - свободен, разпределен за файл или повреден. Код 0 в елемент означава свободен клъстер. Код 2, 3, 4, ..., N (максимален номер на клъстер) означава, че съответният клъстер е разпределен за файл, а числото в елемента е указател към следващ елемент на FAT (което е и адрес на следващ клъстер на файла), т.е. елементът е част от верига елементи на FAT, представяща клъстерите, разпределени за файл. Максималният код, който може да се запише в елемент, означава край на верига, т.е. съответният клъстер е последен във файл. На Фиг.15 е показано примерно съдържание на FAT. Използвани са следните обозначения: EOF - код за последен клъстер (0xFFFF) и FREE - код за свободен клъстер (0).



Фиг. 15. Таблица FAT в MSDOS

Каталози

Едно от различията при представяне на каталозите в MSDOS е, че коренният каталог е в отделна област и с фиксиран по време на форматирането размер (до 256 записа). Всички останали каталози са разположени в областта за данни, имат променлива дължина и памет за тях се разпределя динамично, както и при обикновените файлове. Но всички каталози имат еднаква структура. Съдържат записи с дължина по 32 байта, всеки от които описва файл или подкаталог. Структурата на запис от каталога е показана на Фиг. 16.

Фиг.16. Структура на запис от каталог в MSDOS

В полетата име (8 байта) и разширение (3 байта) е записано собственото име на файла. Компонентата разширение на името не е задължителна.

В полето флагове (1 байт) се съхраняват различните атрибути-флагове, които определят характеристики на файла

Таблица 3. Значение на битовете в атрибута флагове

7	6	5	4	3	2	1	0	Предназначение
							1	само за четене
						1		скрит файл
					1			системен файл (от CP/M)
				1				етикет на тома
			1					каталог
		1						бит при архивиране

В полето време (2 байта) се съхранява времето на създаване или последно изменение на файла.

Полето дата (2 байта) съдържа датата на създаване или последно изменение на файла.

В полето първи клъстер (2 байта) е записан номера на първия елемент от FAT, от който започва веригата, представяща разпределените за файла клъстери. Ако за файла още не е разпределена памет, полето съдържа 0.

В полето размер (4 байта) е записан размера на файла в брой байта, въпреки че отделената дискова памет може да е повече, тъй като се разпределя в клъстери. Това ограничава размера на файл до 4GB.

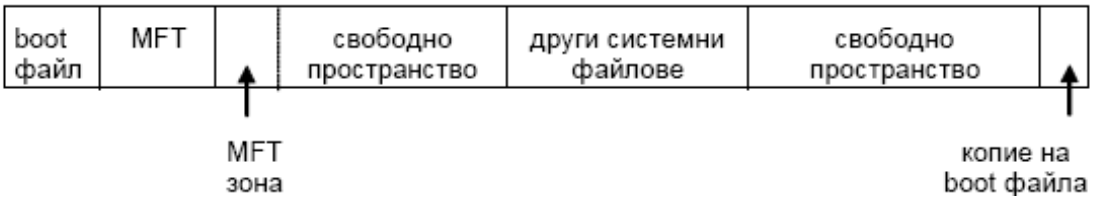
Всички каталози без коренния съдържат двата стандартни записа - с име "." за самия каталог и с име ".." за родителския каталог. Коренният каталог пък съдържа един специален запис за етикета на тома, който е символен низ с дължина до 11 символа и се записва в първите две полета на записа. Това е другата разлика в представянето на коренния и останалите каталози.

- NTFS

В NTFS Схемата на разделяне на дялове и изграждане на томове е усъвършенствана. Освен обикновените томове (simple volumes), се реализират многодялови томове (multipartition volumes). Многодялов том означава, че една файлова система се изгражда върху няколко дяла.

Единицата за разпределяне и адресиране на дисковото пространство от NTFS е клъстер. Адресът на клъстер относно началото на тома се нарича **LCN** (Logical cluster number), а адресът в рамките на определен файл се нарича **VCN** (Virtual cluster number).

Една от ключовите новости в NTFS е принципа „всичко на диска е файл”: и данните и метаданните се съхраняват в тома като файлове, т.е. на всеки том има обикновени файлове, каталози и системни файлове. Това, че метаданните се съхраняват като файлове, позволява динамично разпределяне на дискова памет при нарастване на метаданните, без да са необходими фиксирани области върху диска за тях. Сърцето на файловата система и главният системен файл е **MFT** (Master File Table).



Файлът MFT представлява индекс на всички файлове на тома. Съдържа записи по 1KB и всеки файл на тома е описан чрез един запис, включително и MFT. Освен MFT има и други файлове с метаданни, чиито имена започват със символа $ и са описани в първите записи на MFT. Самите системни файлове са разположени към средата на тома. В началото на тома е boot файла. Скрито в края на тома е копие на boot файла. За да се намали фрагментирането на MFT файла, се поддържа буфер от свободно дисково пространство - MFT зона, докато останало дисково пространство не се запълни. Размерът на MFT зоната се намалява на половина винаги когато останалата част от тома се запълни.

MFT файл

Всеки файл на тома е описан в поне един запис на MFT файла. Индексът (номерът) на началния MFT запис за всеки файл се използва като идентификатор на файла във файловата система (File reference number). Първите 24 записа са резервирани за системните файлове, т.е. те имат предопределени индекси. Следва списък на част от тези файлове.

Име на файл	Индекс	Описание на съдържанието на файла
$Mft	0	MFT файл
$MftMirr	1	Копие на първите записи от MFT файла
$LogFile	2	Журнал при поддържане на транзакции
$Volume	3	Описание на тома
$AttrDef	4	Дефиниции на атрибутите
\	5	Коренен каталог на тома
$Bitmap	6	Битова карта на тома
$Boot	7	Boot сектори на тома
$BadClus	8	Списък на лошите клъстери на тома

Адресът на MFT файла (на началото му) се намира в Boot файла. Първият запис в MFT файла описва самия файл и следователно съдържа адресна информация, осигуряваща достъп до целия MFT файл, ако той е фрагментиран и заема няколко области на тома. Файлът $MftMirr съдържа копие на първите няколко (16) записа от $Mft. Целта на това дублиране е по-висока надеждност на файловата система. Файлът $LogFile се използва за поддържане на транзакции при манипулиране структурата на файловата система. Всяка последователност от дискови операции, които реализират една операция на файловата система, като създаване, преименуване, унищожаване на файл и др., представлява транзакция. NTFS създава записи за тези операции в $LogFile. Тези записи се използват за възстановяване на коректността на файловата система след сривове. Действието на всички незавършили транзакции се отменя и файловата система се възстановява до състоянието си преди началото на тези транзакции. Файлът $Volume съдържа информация за тома, като сериен номер и име на тома, версия на NTFS, дата на създаване и др. Файлът $Bitmap представлява битова карта на клъстерите на тома. Причината и boot сектора да е файл, е може би за да се спази правилото всичко на диска е файл. Всички повредени сектори на тома са организирани във файл на лошите клъстери на тома $BadClus.

Атрибути на файл

Всеки файл се съхранява като съвкупност от двойки „атрибут/значение". Един от атрибутите са данните на файла (наричан unnamed data attribute). Други атрибути са име на файл, стандартна информация и други. Всеки атрибут се съхранява като отделен поток от байтове. Обикновен файл може да има и други атрибути данни, наричани named data attribute. Това променя представата ни за файл, като една последователност от байтове, т.е. файлът може да има няколко независими потока данни. Следва списък на част от типовете атрибути (общият им брой е около 14).

Име на тип атрибут	Описание на значението на атрибута
$FILE_NAME	Името на файла. Един файл може да има няколко атрибута от този тип, при твърди връзки или ако се генерира кратко име в MSDOS стил.
$STANDARD_INFORMATION	Атрибути на файл, като флагове, време и дата на създаване и последно изменение, брой твърди връзки.
$DATA	Данните на файла. Всеки файл има един неименован атрибут данни и може да има допълнителни именовани атрибути данни.
$INDEX_ROOT, $INDEX_ALLOCATION, $BITMAP	Три атрибута използвани при реализацията на каталозите.
$ATTRIBUTE_LIST	Този атрибут се използва, когато за файл има повече от един запис в MFT. Съдържа списък от атрибутите на файла и индексите на записите, където се съхраняват.
$VOLUME_NAME, $VOLUME_INFORMATION	Тези атрибути се използват само в файла $Volume. Те съхраняват информация за тома, като етикет, версия.

Всеки тип атрибут освен име на типа, има и **числов код** на типа. Този код се използва при наредба на атрибутите в MFT запис на файла. MFT записът съдържа числовия код на атрибута (който го идентифицира), значение на атрибута и евентуално име на атрибута. Значението на всеки атрибут е поток от байтове. Един файл може да има няколко атрибута от един тип, например няколко $FILE_NAME или $DATA атрибута.

Атрибутите биват **резидентни** и **нерезидентни**. Резидентен е атрибут, който се съхранява изцяло в MFT записа. Някои атрибути са винаги резидентни, например $FILE_NAME, $STANDARD_INFORMATION, $INDEX_ROOT. Ако значението на атрибут, като данните на голям файл, не може да се съхрани в MFT записа, то за него се разпределят клъстери извън MFT записа. Такива атрибути се наричат нерезидентни. Файловата система решава как да съхранява един атрибут. Нерезидентни могат да бъдат само атрибути, чиито значения могат да нарастват, например $DATA.

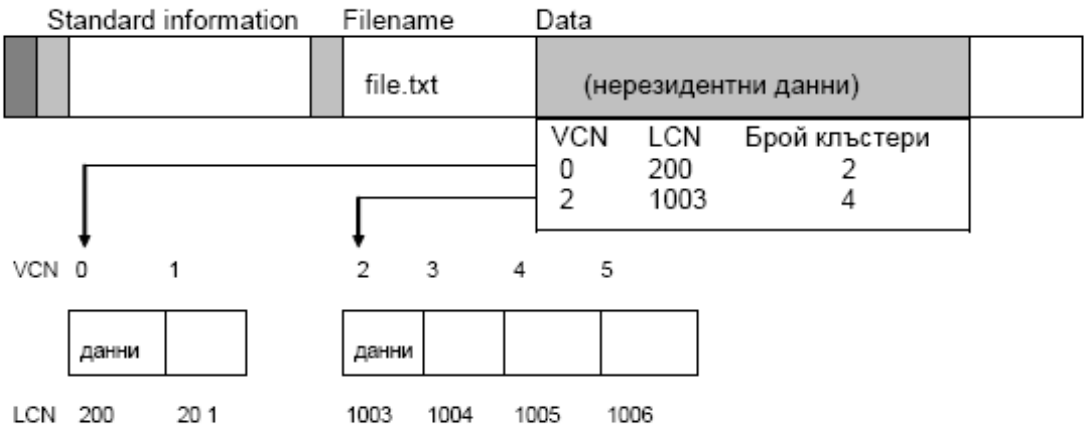
MFT запис

Всеки MFT запис съдържа заглавие на записа **(record header)** и атрибути на файла. Всеки атрибут се съхранява като заглавие на атрибута **(attribute header)** и данни. Заглавието на атрибута съдържа код на типа, име, флагове на атрибута и информация за разположението на данните му. Един атрибут е резидентен ако данните му се поместват в един запис заедно със заглавията на всички атрибути. На Фиг.20 е изобразена структурата на MFT запис за малък файл, т.е. всички атрибути на файла могат да се съхранят в MFT записа.

Фиг.20. MFT запис за файл само с резидентни атрибути

Заб: Може ли без разясненията за файл с нерезидентни атрибути

Ако един атрибут не е резидентен, заглавието му (което е винаги резидентно) съдържа информация за клъстерите, разпределени за данните му. Адресната информация се съхранява подобно на подхода в HPFS, т.е. представлява последователност от описания на екстенти (run/extent entry). Всеки екстент е непрекъсната последователност от клъстери, разпределени за данните на съответния атрибут и се описва от адрес на началния клъстер и дължина (VCN, LCN, брой клъстери). За разлика от HPFS, тук описанията на екстентите се съхраняват в последователна структура, а не в B+дърво.
На Фиг.21 е изобразена структурата на MFT запис за по-голям файл, т.е. данните на файла се съхранят в два екстента.

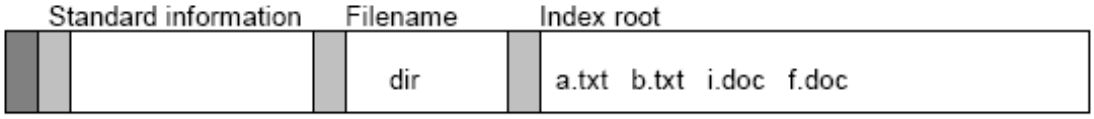


Фиг.21. MFT запис за файл с нерезидентен атрибут данни

Каталози

Каталогът съдържа записи с променлива дължина, всеки от които съответства на файл или подкаталог, в съответния каталог. Всеки запис съдържа името на файла и индексът на основния MFT запис на файла, както и копие на стандартната информация на файла. Това дублиране на стандартната информация изисква две операции писане при изменението ѝ, но ускорява извеждането на справки за съдържанието на каталога.

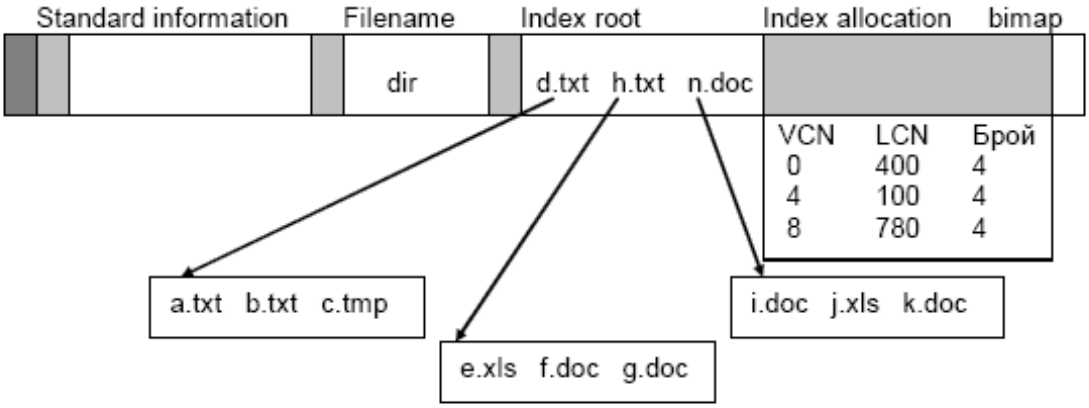
Записите в каталога са сортирани по името на файла и се съхраняват, подобно на HPFS, в структура B дърво. Ако каталогът е малък, тези записи се съхраняват в атрибута $INDEX_ROOT, който е резидентен, т.е. целият каталог се намира в MFT записа си. На Фиг.23 е изобразена структурата на MFT запис за малък каталог.



Фиг.23. MFT запис за малък каталог

Заб: Може ли без разясненията за представяне на голям каталог?

Когато каталогът стане голям, за него се разпределят екстенти с размер 4KB, наречени индексни буфери (index buffers). Атрибутът $INDEX_ROOT и тези екстенти са организирани в B дърво. В този случай каталогът има и атрибут $INDEX_ALLOCATION, който съхранява адресна информация за разположението на екстентите-индексни буфери. Атрибутът $BITMAP е битова карта за използването на клъстерите в индексните буфери. Фиг.24 илюстрира представянето на голям каталог (за простота всеки клъстер съдържа 1 запис).



Фиг.24. Представяне на голям каталог

14. Компютърни мрежи: Компютърни мрежи и протоколи – OSI модел. Канално ниво. Маршрутизация. IP, TCP, HTTP.

OSI модел – най-обща характеристика на нивата. Канално ниво – кадри, прозорци, предаване и грешки. Какво е характерно за Ethernet. Статична маршрутизация – таблица и избори. Централизирана маршрутизация – недостатъци. Разпределена маршрутизация – алгоритъм с дистантен вектор, алгоритъм със следене състоянието на връзката. IP дейтаграма. IP адресация – класова, безкласова, преобразуване на IP в MAC и обратно. TCP – 3-way hand shake. Формат. Разлика с UDP. HTTP.

OSI модел – най-обща характеристика на нивата.

Стандартът **OSI** (Open System Interconnection), разработен от Международната организация за стандартизация (International Standard Organization – ISO) стандартизира начините за връзка между отворени системи. В случая терминът „отворена система" означава система, чиито ресурси могат да се използват и от другите системи, образуващи мрежата. Важен принцип е разслояването, което разделя логически всяка от системите, образуващи мрежата, на йерархично подредени подсистеми. Слоевете са 7 и се номерират отдолу нагоре. Всеки слой осигурява определено обслужване за по-горния, като използва функциите на по-долния.

7	Приложен слой
6	Представителен слой
5	Сесиен слой
4	Транспортен слой
3	Мрежов слой
2	Канален слой
1	Физически слой

- **Физически слой** (*Physical layer*): Обектите от този слой са технически средства, реализиращи предаването на данни през определена физическа среда, т.е. това е нивото на физическите връзки. Основната му функция е да управлява кодирането и декодирането на сигналите, интерпретиращи двоичните цифри 0 и 1, без да се интересува от предназначението на тези битове и на предаваните данни. В този слой предавателят има за цел да изпрати поредица от кодирани 0 и 1, а приемникът да ги декодира в цифров вид.

- **Канален слой** (*Data-link layer*): Този слой реализира връзките на логическо ниво, т.е. се занимава с обмена на данни, без да го интересува начинът, по който те се преобразуват в електрически сигнали във физическия слой. Типична функция е откриването и коригирането на грешки при предаването на данните. Обикновено данните на канално ниво се обменят на порции с фиксирана дължина, наречени кадри, чийто формат се определя от избрания протокол за предаване на канално ниво. Функциите на този слой обикновено се реализират смесено – апаратно и програмно.

- **Мрежов слой** (*Network layer*): Този слой реализира връзките на мрежово ниво. Занимава се с изпращането и получаването на пакети (дейтаграми) и доставката им (маршрутизация). Този слой отговаря и за адресацията на машините в мрежата. Комутирането на пакетите е друга важна задача, както и предотвратяването на претоварвания в мрежата. Функциите на мрежовия, както и на всички по-горни слоеве, обикновено се реализират програмно.

- **Транспортен слой** (*Transport layer*): Осигурява транспортирането на съобщения от системата-източник до системата-приемник и представлява надстройка над мрежовия слой. Слоят управлява обмена на данните и осъществява връзка от тип

„край-край". Гарантира за надеждното и ефективното транспортиране на данните. Неговите действия остават скрити за по-горните слоеве по отношение на управлението и движението на потока от данни от единия хост към другия. Функциите, изпълнявани от транспортния слой, отговарят за цялостта на управлението на диалога между крайните потребители, включително откриване на загубени съобщения и повторното им изпращане.

- **Сесиен слой** (*Session Layer*): Отговорен за диалога между две комуникиращи програми и за управление на обмена на данни между тях, използва интерактивен диалог между двете програми. Дефинира 3 типа диалози: двупосочен едновременен, двупосочен алтернативен и еднопосочен диалог. При организиране на диалога в сесийния слой се включват синхронизиращи елементи, които позволяват прекъсване на диалога и в последствие възстановяването му от мястото на прекъсването.

- **Представителен слой** (*Presentation Layer*): Най-ниският слой, който се интересува от значението на потоците обменяни битове. Грижи се за запазването на информационното съдържание на данните. Има две основни функции: да договаря общ синтаксис за предаване на съобщенията и да осигурява възможност едната програмна система да не се грижи какъв е форматът на вътрешната структура на представените данни, които другата система използва. Ако двете системи са например персонални компютри, работещи на Basic, представителният слой няма особени функции, тъй като двете програми имат общи вътрешни структури на данните. В противен случай обаче представителният слой извършва необходимото преобразуване, като позволява на всяка програма да работи със собствения си формат, без да се интересува от формата, който използва другата програма.

- **Приложен слой** (*Application Layer*): Най-горният слой, към който се свързват потребителските процеси. Предоставя средства за работа с приложни програми, за които осигурява достъп до системите за осъществяване на комуникациите. На ниво приложен слой програма, работеща на един хост, изпраща съобщение, което се получава от програма, работеща на друг хост, и този слой не се интересува как точно съобщението на програмата от предаващия компютър е стигнало до приемащия.

При изпращане на данни всеки слой възприема цялата информация, получена от по-горния слой като данни, и добавя своя собствена управляваща информация, наречена заглавна част (header), за да осигури правилна доставка на информацията. Добавянето на информация за осигуряване на доставката се нарича опаковане. При получаване на данни от по-долен слой настъпват процеси, обратни на горните. Всеки слой премахва своята заглавна част, преди да подаде данните на по-горния слой. Информацията, получена от по-долен слой, се интерпретира като съвкупност от заглавна част и данни.

Канално ниво – кадри, прозорци, предаване и грешки.

Каналното ниво е второто ниво в седемслойната архитектура, между физическия и мрежовия слой. То организира прехвърлянето на данни от мрежовия слой на машината-източник към мрежовия слой на машината-получател. Обектите на този слой реализират връзките на логическо ниво, т.е. не се интересуват от начина, по който данните се

преобразуват в електрически сигнали от съоръженията, работещи на физическия слой. Функциите на канално ниво се реализират смесено – апаратно и програмно.

Обща схема на действие: Мрежовият слой на абоната А подава данни на каналния слой на абоната А, които трябва да достигнат до мрежовия слой на абоната В, ако А и В са директно свързани, или до мрежовия слой на маршрутизатор С, ако не са. Каналният слой на А кадрира според съответния протокол данните и ги предава на физическия слой на А; физическият слой на А ги предава на физическия слой на В (или на С); физическият слой на В ги предава на каналния слой на В; извършва се проверка на контролната сума на кадъра и, ако няма грешка, кадърът се предава на мрежовия слой на В, а, ако има грешка, кадърът се отхвърля. Аналогично се извършва предаването на кадри между маршрутизатори.

Обикновено данните на канално ниво се обменят на порции с фиксирана дължина, наречени **кадри**. В мрежите с пакетна комутация (такива, при които съобщенията се разделят на пакети; всеки пакет се предава индивидуално в комуникационната подмрежа и възловият компютър се грижи за съхраняването и предаването не на единици-съобщения, а единици-пакети; други видове комутация – на канали, където се установява физически канал за обмен на информация и по този канал се предава едно съобщение; на съобщения, където всяко съобщение се изпраща в комуникационната подмрежа, която избира неговия маршрут до назначението му) обикновено кадрите съвпадат по дължина с размера на пакета. Форматът на кадрите се определя от избрания протокол за предаване на канално ниво.

Кадърът е пакет, който каналният слой използва при обмена на информация с физическия. Физическият слой възприема информацията от каналния слой като поток от битове. Кадърът е поредната част от този поток, която каналният слой контролира за наличието на грешки. Формирането на кадрите се осъществява, като информационният поток, получен от мрежовия слой, се дели на части, към всяка от които се добавя служебна информация. Обратно – в посока от битове от физическия слой, каналният слой идентифицира (разпознава начало и край) кадрите, проверява ги за грешки (при наличие на такива може да ги коригира, зависи от конкретната реализация), премахва се служебната информация на каналния слой и информационният поток се предава към мрежовия слой. Възниква въпросът как да стане разделянето на кадри. Понастоящем се използват три метода:

- Броят се отделните символи, като в заглавието на кадъра се указва и броят им (и символите от заглавната част участват в броенето). Проблемът е, че може поради грешка да не може да се установи точно размерът на даден кадър, а от там и на всички останали.

- Началото и краят на всеки кадър се определят от уникална символна последователност (STX и ETX); start – DLE STX; end – DLE ETX. Ако в информацията на кадъра се срещне DLE, то се удвоява. Проблемът е, че методът е базиран на ASCII кодирани 8-битови символи, което е свързано с нерационално използване на възможностите на комуникационната среда.

- Всеки кадър започва и завършва със специална последователност 01111110, наречена флагов байт. В съдържанието на кадъра след всяка последователност от 11111 се добавя 0.

В много протоколи се среща комбинация от първите два или от първия и третия начини.

Предаване – За да се избегнат грешките на канално ниво, се използват два механизма: потвърдено обслужване (потвърждаване на всеки получен кадър) и установено

обслужване (номериране на последователностите от изпращани кадри). Три са основните варианти на комуникационната услуга, която каналният слой може да осигурява:

- Непотвърдено неустановено обслужване – отсъства обратна информация от B или C за приемането на кадрите; няма установяване на връзка и освобождаването ѝ след това => не се държи сметка за последователността, в която се получават кадрите; използва се, когато навременното регулярно получаване на кадрите е по-важно от тяхната достоверност, например при предаване на реч или видео.
- Потвърдено неустановено обслужване – всеки кадър се потвърждава; по принцип потвърждаването се прави на ниво транспортен слой, но то се отнася до последователност на пакети; потвърждаване на канално ниво се налага при ненадеждна комуникационна среда, например безжична.
- Потвърдено установено обслужване – характеризира се с три фази: установяване на връзката със заделяне и инициализация на необходимите ресурси; предаване на кадрите; освобождаване на връзката и заделените за нея ресурси => така се гарантира и предаването на кадрите, и последователността, в която те се предават.

Грешки:

- Кадърът пристига в B с грешка -> B изпраща на A потвърждаващ кадър, в който съобщава за грешката; A изпраща кадъра отново. Ако не е налице потвърдено обслужване, A изпраща кадъра след определен TIMEOUT > от времето за потвърждаването.
- Кадърът не пристига в B -> след изтичане на TIMEOUT A изпраща кадъра отново.
- Кадърът е пристигнал в B, но потвърждението се е загубило -> A изпраща кадъра отново след TIMEOUT, B получава и новия кадър и се получава дублиране -> присвояват се номера на всеки кадър от изпращаните (установено обслужване), така че B да различава оригинала от дубликатите.
- A изпраща кадри по-бързо, отколкото B може да ги приеме -> въвеждат се механизми за управление на потока от кадри, които осигуряват обратна информация на A за темпа на предаване.

За установяване на състоянието на канала в A се прави настройка по брой повторения n. След като A предаде един и същи кадър n пъти, A счита канала за неработещ и прекратява опитите за изпращане на кадъра.

При препредаване на кадри не е нужна корекция на грешка, а само установяването ѝ.

Какво е характерно за Ethernet.

Локалната мрежа Етернет използва метода за достъп МДОН/РК (Множествен достъп с откриване на носеща [честота] и разпознаване на конфликтите). Същността на метода се състои в непрекъснато подслушване на канала – каналът се подслушва преди предаване на данни и данни не се предават, докато той не се освободи. След като каналът се освободи, данни не се предават още определено време >= прозореца за конфликти. Така цялата съобщителна среда преминава в пасивно състояние. Предаването започва, но подслушването продължава. Ако се открие интерференция в приемания сигнал, това означава, че е настъпил конфликт. След установяването на конфликта предаването продължава още известно време = прозореца за конфликти. Станцията, участвала в конфликта, отлага препредаването на пакета със случайно подбран интервал от време.

Основно предимство на МДОН/РК е, че предаващата станция със сигурност знае кога пакетът е предаден успешно в канала. Другото предимство е малкото време, в което каналът стои зает по време на конфликт. Основният недостатък е силната зависимост от времето за разпространение на сигнала.

Протоколът на поднивото за достъп до средата (MAC – Media Access Control) в Етернет стандартизира следния формат на кадъра:

7 B	1 B	6 B	6 B	2 B	0-1500 B	0-46 B	4 B
Преамбюл	Начало на кадъра / Начален разделител	Адрес на получателя	Адрес на източника	Дължина на данните	Данни	PAD	Контролна сума

- Кадърът започва със синхронизиращи байтове преамбюл – поредица от 10101010.
- Началният разделител / Началото на кадъра (SOF, Start of Frame) е поредицата 10101011.
- Най-старшият бит на адреса на получателя е 0 за нормален адрес и 1 за групов адрес (multicast). Адрес, съставен само от 1, е broadcast адрес. Бит 46 се използва за различаване на локални от глобални адреси.
- Полето за дължина на данните сочи броя байтове в следващото поле за данни + PAD. Кадри, по-къси от 64 байта, създават проблеми, затова минималната дължина на кадъра (от адреса на получателя до контролната сума включително) е 64 байта => трябва да има поне 46 байта данни, а, ако няма, се запълва с PAD до 46 байта.
- Контролната сума се използва, за да се следи за това дали кадърът е коректно получен.

Предаването на къс кадър е проблемно, тъй като е възможно, преди достигането на първия бит до края на кабела, друга станция да започне предаване. Тогава възниква конфликт. По-близката до конфликта станция разбира за конфликта, прекратява предаването и генерира 48-битова поредица (jam), с което предупреждава всички станции за конфликта. Ако обаче другата станция, която е участвала в конфликта, вече е прекратила предаването, тя е счела, че кадърът е успешно предаден, и го е изчистила от буферите си преди jam сигналът да достигне до нея. Така кадърът е загубен. За предотвратяването на това времето за предаване на кадър трябва да е повече от 2 пъти времето за разпространение на кадър от единия до другия край на кабела.

Статична маршрутизация – таблица и избори.

Основна функция на мрежовия слой е доставянето на пакети от източника до местоназначението им. Изпълнението на тази задача може да изисква няколко на брой стъпки, в които пакетите преминават последователно през различни междинни маршрутизатори. Всеки от маршрутизаторите избира подходяща следваща стъпка за предаване на пакетите въз основа на записаната в неговата **маршрутна таблица** информация. В маршрутната таблица обикновено са записани само част от всички възможни следващи стъпки към дадено местоназначение. Съществуват различни критерии за избор на най-добър път, например – брой на стъпките, закъснение на пакетите, пропускателна способност, натоварване, надеждност или цена на връзките. С всеки маршрут е асоциирана **метрика**, която представлява функция на една или повече от тези променливи.

Маршрутизацията се нарича **статична**, ако маршрутната таблица е попълнена ръчно от администраторите. Тя е удобна единствено в малки мрежи, при които рядко настъпват промени. Полезното ѝ е, че е предсказуема, но пък изисква много обстойно планиране и не се адаптира динамично към промени в топологията на мрежата.

Централизирана маршрутизация – недостатъци.

При **централизираната маршрутизация** маршрутизирането се извършва чрез специален управляващ мрежата компютър (маршрутен управляващ център (routing command center)), който изчислява конфигурацията на мрежата и построява маршрутната таблица за всички възли в мрежата. Периодично се препостроява тази маршрутна таблица и се изпраща към възлите в мрежата. Основно предимство е простотата на този модел – решение за маршрутизацията се взима самостоятелно от един компютър. Освен това, информация като параметри на мрежата и на връзките в нея, които не се променят често, могат да се пазят в централна база от данни и няма нужда да бъдат обявявани всеки път.

Недостатъци на централизираната маршрутизация:

- Ако се случи нещо с управляващия компютър, инструкциите за маршрутизиране не могат да бъдат променяни, докато не се поправи този компютър.
- Маршрутната таблица не отчита претоварванията в мрежата.
- Когато се промени маршрутната таблица, се хабят мрежови ресурси за разпращането на обновената таблица на всички възли в мрежата.
- Ако мрежата се разцепи на две, едната част остава без централен маршрутизиращ компютър и тази част от мрежата не може да се адаптира към промени в топологията.

Разпределена маршрутизация.

Разпределената маршрутизация позволява всички възли в мрежата да взимат свои решения за маршрутизирането, следвайки формален протокол за маршрутизиране. При разпределената динамична маршрутизация маршрутизаторите самостоятелно и автоматично определят следващите стъпки към всички известни направления и реагират

своевременно на евентуални промени в топологията на мрежата, отпадането на устройства или връзки между тях. При възникнали промени, времето, необходимо за преизчисляване на маршрутните таблици и достигане до непротиворечиво описание на новата топология, определя скоростта на сходимост на маршрутизиращия протокол.

Има два основни типа маршрутизиращи протоколи:

- **Протоколи с дистантен вектор/вектор на разстоянието** – маршрутизаторите обменят информация за топологията на цялата мрежа само със своите съседи. Тези протоколи изискват по-малко мрежови и изчислителни ресурси, но имат по-ниска скорост на сходимост и не гарантират отсъствието на цикли в маршрутите. Сериозен недостатък е, че добрите новини (включването на нов маршрутизатор) се разпространяват бързо в мрежата, но лошите новини (изключване на маршрутизатор) обикновено изискват твърде голям брой периодични съобщения, за да достигнат до всички маршрутизатори. Всеки ред в маршрутната таблица съдържа адрес на дадено местоназначение, адрес на следващата стъпка към това местоназначение и метрика. Предполага се, че всеки маршрутизатор знае метриката на връзките до своите съседи. Метриката може да бъде „брой стъпки", „пропускателна способност", „цена на връзката", „натоварване", „надеждност" или „закъснение на пакетите". Действието е следното: през определен интервал от време всеки маршрутизатор изпраща на своите съседи съдържанието на маршрутната си таблица; когато маршрутизатор получи таблицата на свой съсед, той преизчислява своята.
- **Протоколи със следене състоянието на връзката** – маршрутизаторите обменят информация само за своите връзки към мрежата с всички маршрутизатори в нея. Тези протоколи имат по-висока скорост на сходимост и предотвратяват възникването на цикли в маршрутите, но с цената на повече процесорно време и памет, както и увеличен обем на обменяната между маршрутизаторите информация и увеличена сложност на протоколните реализации. Действието е следното: маршрутизаторът изследва първо своите съседи, като им изпраща ехо-пакети, а те му отговарят с идентификационните си номера (имена) и със служебните си адреси; прави изчисления, изследва околната мрежа; създава специални пакети, които съдържат вече получената информация за връзките със съседите; изпраща тези пакети до всички маршрутизатори в мрежата; след като получи всички пакети, строи граф на мрежата и по алгоритъма на Дейкстра изчислява най-кратките пътища до всички

маршрутизатори в мрежата. Тъй като всеки трябва да получи всичко, се използва алгоритъм с наводняване.

IP дейтаграма.

Интернет протоколът (IP) дефинира основната единица за предаване на данни в интернет – **дейтаграмата**. Най-общо, терминът „пакет" се използва за всякакво съобщение, оформено като пакет, докато „дейтаграма" се използва в случаите на ненадеждна услуга.

IP (Internet Protocol v4) е протокол на мрежовото (3-то) ниво, който използва дейтаграмен (ненадежден) метод на предаване без установяване на връзка. По тази причина всяка дейтаграма трябва да съдържа пълна информация за адресите на получателя и източника. IP не гарантира успешното получаване на дейтаграмите в местоназначението. Ако за дадени приложни програми е необходима надеждност при предаването, то тя се гарантира от протоколите от по-високо ниво (например TCP). Основните функции на IP са: адресиране (чрез заглавната част на дейтаграмата се задават адреси, чрез които междинните маршрутизатори избират път за пакета); фрагментиране (позволява се големи по размер пакети да преминават през мрежи, които могат да обработват само малки пакети); таймаут на пакетите (стойността на полето „време на живот" – TTL, се инициализира от изпращача и се намалява с 1 всеки път, когато пакетът премине през маршрутизатор; така се предотвратява евентуално зацикляне); тип на услугата (може да се задават приоритети на трафика) и опции (изисквания за пътя и проследяване на пътя).

Формат на дейтаграмата за протокол IP версия 4:

<table>
<tr><td colspan="3">0 15</td><td colspan="3">16 31</td></tr>
<tr><td>Версия – 4 бита</td><td>Дължина на заглавната част – 4 бита</td><td>Тип на услугата – 8 бита</td><td colspan="3">Обща дължина на дейтаграмата в байтове – 16 бита</td></tr>
<tr><td colspan="3">Идентификатор – 16 бита</td><td>Флагове – 3 бита</td><td colspan="2">Отместване на фрагмента – 13 бита</td></tr>
<tr><td colspan="2">Time-To-Live – 8 бита</td><td>Протокол – 8 бита</td><td colspan="3">Контролна сума на заглавната част – 16 бита</td></tr>
<tr><td colspan="6">IP адрес на източника – 32 бита</td></tr>
<tr><td colspan="6">IP адрес на получателя – 32 бита</td></tr>
<tr><td colspan="6">Опции (не е задължително да присъстват, при необходимост се добавят 0)</td></tr>
<tr><td colspan="6">Данни (не е задължително да присъстват)</td></tr>
</table>

- В зависимост от версията дейтаграмата има различен формат. В случая на IPv4 стойността на полето за версия е 4.

- Дължината на заглавната част (header length) е броят 32-битови думи, вкл. полето за опции. Минималната стойност е 5, а максималната – 15, т.е. може да има до 40 байта опции.

- Полето за тип на услугата (type of service) се състои от 3 водещи бита, 4 бита за вид на услугата и 1 неизползван бит, който трябва да е 0. 4-те бита за вид на услугата задават съответно изисквания за: минимум закъснение, максимална пропускателна способност, максимум отказоустойчивост, минимум цена, като наведнъж само един от тези 4 бита може да бъде 1.

•Максималната стойност на общата дължина на дейтаграмата е 65,535 байта. Това поле може да бъде променено при фрагментиране на дейтаграмата.

•Полето идентификатор има еднаква уникална стойност за всички фрагменти, принадлежащи на дадена дейтаграма.

•Старшият бит на полето флагове (flags) е резервиран. Ако следващият е 1, не трябва да се извършва фрагментация на дейтаграмата. Най-младшият бит е 1 за всички фрагменти от една дейтаграма с изключение на последния.

•Отместването на фрагмента (fragment offset) указва местоположението на фрагмента в дадена дейтаграма. Дължината на всички фрагменти без последния трябва да е кратна на 8 байта (елементарен фрагмент).

•Времето за живот (Time-To-Live) е горната граница на броя маршрутизатори, през които може да премине дейтаграмата.

•Протокол (protocol) – указва с кой протокол от по-високо ниво се работи.

•Контролната сума (header checksum) се пресмята само върху полетата на заглавната част на IP дейтаграмата: 1. Първоначалната стойност се инициализира да бъде 0; 2. Заглавната част се разглежда като последователност от 16-битови думи в допълнителен код. Изчислява се сумата (побитова сума по модул две) на тази първоначална стойност и последователността от думи на заглавната част; 3. Допълнението до 16 бита единици на така изчислената сума се записва в полето за контролна сума.

IP адресация – класова, безкласова.

Адресите в IPv4 са 32-битови двоични числа, които е прието да се записват като 4 десетични числа, разделени от точки (всяка от тези 4 части се нарича октет). Логически мрежовите адреси имат 2 части: мрежов идентификатор (netid) и идентификатор на хоста (hostid). Ако hostid е само от 0, това е адресът на мрежата, към която хостът принадлежи. Ако пък е изцяло от 1, това е адрес за предаване до всички машини (broadcast address), който едновременно адресира всички машини в дадена мрежа. Например 149.76.255.255 адресира всички хостове в мрежа 149.76.0.0. Има и 2 резервирани адреса – 0.0.0.0 (маршрут по подразбиране, използва се при маршрутизацията на IP дейтаграмите) и 127.0.0.0 (резервиран за локален IP трафик; присвоява се интерфейс за обратна връзка на хоста – loopback interface). Адресът 127.0.0.1 (може и 127.x.x.x, но за удобство пишем 127.0.0.1) е резервиран и се използва за служебен адрес на самия хост (localhost address).

Адресни класове в Интернет:

A	0	network (1+7)				host (24)				
B	1	0	network (2+14)				host (16)			
C	1	1	0	network (3+21)				host (8)		
D	1	1	1	0	multicast address (28)					нямат адреси на хостове
E	1	1	1	1	reserved for future use (27)					

Съществуват 128 мрежи от клас A и над 16 милиона хоста за всяка мрежа от този клас. Т.е. има малко мрежи с много хостове. Адресите от клас A са предназначени за много големи мрежи, тези от клас B са за средни по размер мрежи (16 384 мрежи от клас B и над 64 000 хоста за всяка от тях), а от клас C са за малки мрежи.

При клас C: мрежите 192.0.0.0 и 223.255.255.0 са резервирани за служебно ползване.

Т.нар. частни интернет мрежи (private internets) са резервирани за използване от различни организации за реализиране на връзка между компютрите само в рамките на съответната организация. Такива мрежи са: 10.x.x.x от клас A, 16 мрежи 172.16.x.x до 172.31.x.x от клас B и 256 мрежи 192.168.x.x от клас C. Адресите от тези мрежи не се маршрутизират в глобалния Интернет. Най-честото им приложение е при firewall-и.

При адресите от клас D последните 28 бита се използват за адрес за едновременно предаване до група машини (multicast address). Използването им позволява дадена IP дейтаграма да се предаде до „група от хостове". Например едновременно предаване до група хостове се използва за мрежови видео и аудио конференции и за LAN TV.

С разрастването на Интернет най-бързо са изчерпани свободните IP адреси на мрежи от клас B, което налага на организации, притежаващи голям брой компютри, да се дават две или повече мрежи от клас C. Това води до увеличаване на размерите на глобалните таблици: от една страна, защото в маршрутизиращите таблици има по няколко записа, които водят до общ маршрутизатор на организацията, а от друга – заради произволното раздаване на номера на мрежи, което налага маршрутизаторите да пазят запис за всяка мрежа, без възможност за агрегиране. За намаляване на този обем се въвежда **безкласова адресация и маршрутизация** – CIDR (Classless Inter-Domain Routing). CIDR записът представя всеки адрес като 32-битово число, последвано от наклонена черта „/" и броя единици в двоичния запис на subnet mask-ата. Например, 192.0.2.96/28 означава IP-адрес, в който първите 28 бита са netid-то, т.е. subnet mask-ата е 255.255.255.240.

Преобразуване на IP в MAC и обратно.

За адресация в IPv4 се използват 32-битови IP адреси, а хостовете, свързани към Етернет, притежават 48-битови хардуерни/MAC адреси (които се записват в 16-ичен вид). За **установяване на съответствие между IP адреса и MAC адреса** се използва **ARP** (Address Resolution Protocol). Когато даден хост иска да изпрати дейтаграма към машина в локалната мрежа, знаейки IP адреса, но не и Етернет адреса, той изпраща ARP пакет-заявка, от тип broadcast. Търсеният хост изпраща ARP съобщение до подателя с информация за Етернет адреса си, а всички останали хостове игнорират заявката.

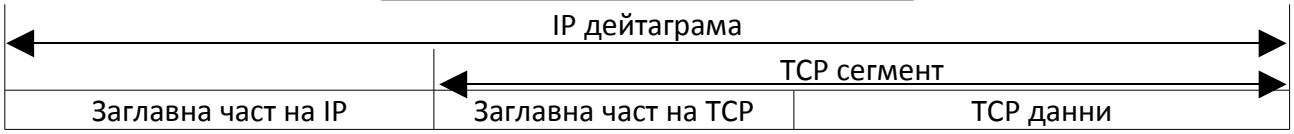
Протоколът **RARP** (Reverse Address Resolution Protocol) се използва в обратната посока, за **намирането на съответствие между Етернет адреси и IP адреси**. Действието на RARP се основава на наличието на уникален хардуерен Етернет адрес за всяка система в локалната мрежа. При инициализиране на машината без твърди дискове RARP протоколът прочита този адрес от интерфейсната карта и предава до всички станции в мрежата пакет-заявка. RARP сървърът отговаря на тази заявка, като в пакета-отговор се съдържа IP адресът, съответстващ на изпратения Етернет адрес.

TCP формат.

TCP (Transmission Control Protocol) е протокол на транспортното (4-то) ниво, създаден специално да предоставя надеждни транспортни услуги за потока от байтове на протоколите от по-горните нива въпреки ненадеждността на използваната мрежова среда. Функциите на този протокол се осъществяват от програмен модул, който в общия случай е част от ядрото на операционната система. TCP установява логическа връзка „от край до край" между 2 приложни програми. TCP осигурява възможност за двупосочно предаване

на данните (пълен дуплекс), както и за надежден обмен на поток от номерирани байтове. TCP се справя с проблеми като загубване на пакети, дублиране на пакети и получаване на пакети не в реда в който са изпратени.

Опаковане на TCP данните в IP дейтаграма:



Обменът на информация се извършва посредством сегменти. При предаване TCP получава данни от по-горния слой, разделя ги на части, опакова ги в „сегменти" и ги изпраща на IP протокола. Той от своя страна опакова сегментите в дейтаграми и извършва маршрутизирането на всяка им. При приемане IP разопакова пристигналите дейтаграми, след което предава получените сегменти на TCP, който сглобява и подрежда данните от сегментите в съобщения към по-горните слоеве така, както те са били изпратени.

Всеки край на TCP връзката се идентифицира с IP адреса на хоста и със 16-битов номер на порт, който определя програмата, използваща тази връзка. Двойката „адрес на хост + номер на порт" се нарича гнездо (socket). Комбинацията от гнездата на източника и на получателя е уникална и идентифицира TCP връзката. Това позволява едно гнездо да се използва едновременно от няколко TCP връзки, т.е. да се мултиплексира.

Заглавната част на TCP сегмента включва задължителни полета с фиксиран размер от 20 байта (5 32-битови думи), към които може да бъде добавено поле *опции*, след които може да има поле на обменяните данни.

Заглавна част на TCP сегмент:

<table>
<tr><td colspan="8">0 15 16 31</td></tr>
<tr><td colspan="6">Номер на порта на източника – 16 бита</td><td colspan="2">Номер на порта на получателя – 16 бита</td></tr>
<tr><td colspan="8">Пореден номер (Sequence number) – 32 бита</td></tr>
<tr><td colspan="8">Номер на потвърждението (Acknowledgement number) – 32 бита</td></tr>
<tr><td>Дължина на заглавната част – 4 бита</td><td>Резерви-рани – 6 бита</td><td>URG</td><td>ACK</td><td>PSH</td><td>RST</td><td>SYN</td><td>FIN</td><td>Размер на прозореца – 16 бита</td></tr>
<tr><td colspan="6">Контролна сума на TCP (checksum) – 16 бита</td><td colspan="2">Указател за спешни данни – 16 бита</td></tr>
<tr><td colspan="8">Опции (не е задължително да присъстват)</td></tr>
<tr><td colspan="8">Данни (не е задължително да присъстват)</td></tr>
</table>

Номерата на портовете на източника (source port) и на получателя (destination port), заедно със съответните IP адреси, образуват двете гнезда, идентифициращи връзката. Следва поредният номер на първия байт, записан в полето *данни* на този сегмент. Номерът на потвърждението е номерът на първия байт данни, който се очаква да се получи със следващия сегмент. Дължината на заглавната част (header length) се мери в брой 32-битови думи. Фактически с него се определя началото на полето данни и затова се нарича и Data Offset. Ако някой от следните 6 еднобитови флага е установен, т.е. има стойност 1, то:

- *URG* – валиден е указателят за спешни данни (urgent pointer), т.е. трябва да се преустанови обработката на получените данни, докато не се обработят байтовете, към които сочи указателят за спешни данни;

- *ACK* – валиден е номерът на потвърждение;
- *PSH* (push) – наличните данни трябва да се изпратят възможно най-бързо към техния получател, т.е. източникът не изчаква образуването на пълен сегмент и съответно получателят не чака запълването на приемния буфер;
- *RST* (reset) – ако е 1, то сегментът служи за прекратяване на TCP връзката;
- *SYN* (synchronization) – сегментът се използва при установяване на TCP връзката и за изпращане на началния номер (задава се в полето *пореден номер)*, от който ще бъдат номерирани байтовете, т.е. се иска синхронизиране на номерацията на сегментите;
- *FIN* (finish) – изпращачът прекратява предаването на данни.

Размерът на прозореца (window size) определя колко байта могат да бъдат изпратени и съответно приети наведнъж, без препълване на входния буфер. Указателят за спешни данни (urgent pointer) указва позицията на първия байт на спешните данни спрямо началото на полето данни. Контролната сума (checksum) се изчислява върху целия TCP сегмент и осигурява проверка за коректността на данните. При нейното пресмятане участва и т.нар. псевдо-заглавна част (pseudoheader) – 12 байта, включващи някои полета от заглавната част на IP дейтаграмата.

Псевдо-заглавна част на TCP дейтаграма:

0	15	16	31
IP адрес на източника – 32 бита			
IP адрес на получателя – 32 бита			
Нула	Протокол (06 за TCP)	Дължина на TCP сегмента	

Ако няма *опции* или са по-малко от 32 бита, останалите битове се запълват с 0 (padding).

TCP – 3-way hand shake.

Първоначално отваряне на връзката (Connection Establishment Protocol): Необходимо е всеки един от двата хоста да изпрати на другия началния номер (initial sequence number) на байтовата последователност, която ще изпраща, и съответно да получи насрещното потвърждение за получаването на този номер. Процедурата е следната:

1. Клиентът изпраща SYN сегмент, в който задава и номера на порта на сървъра, както и началния номер на потока байтове (x).
2. Сървърът отговаря със собствен SYN сегмент, включващ началния номер на неговия поток от байтове (y). Чрез този сегмент сървърът изпраща и ACK за потвърждение на SYN сегмента на клиента.
3. Клиентът потвърждава получаването на този SYN сегмент от сървъра със сегмент ACK.

При обмена и потвърждението на началните номера на байтовите последователности от всеки хост са необходими общо три стъпки, при които се обменят съответно 3 сегмента. Затова тази процедура се нарича **диалог с три съобщения** (**three-way handshake**):

Стъпка	Клиент	Сървър
1	SYN = 1, SEQ = x	
2		SYN = 1, SEQ = y, ACK = x + 1
3	SEQ = x + 1, ACK = y + 1	

TCP – разлики с UDP.

UDP (User Datagram Protocol) е прост транспортен протокол за предаване на дейтаграми в мрежите с комутация на пакети. За разлика от TCP, UDP не осигурява надежден транспорт и подредба на пакетите, не се справя със загубени или дублирани пакети. Дейтаграмите се изпращат от източника, без да се контролира дали са пристигнали до получателя. Услугите, които UDP добавя върху услугите на IP са мултиплексиране на логическия канал за връзка между двата хоста, както и откриване на грешки и контролна сума на данните. UDP, за разлика от TCP, може да се използва за multicast.

HTTP.

HTTP (Hypertext Transfer Protocol) е протокол на приложното (7-мо) ниво. Той представлява прост текстов протокол, който се използва от услугата WWW за осигуряване на достъп до практически всякакъв вид данни, наричани събирателно ресурси.

При HTTP протокола има понятия като клиент (обикновено това са Web-браузърите) и сървър (това са Web-сървърите). HTTP най-често се използва с TCP/IP, но практически може да работи върху всякакъв протокол, който предоставя надежден транспорт. Обикновено HTTP протоколът работи върху стандартен TCP сокет, отворен от клиента към сървъра. Стандартният порт за HTTP е 80, но може да се използва и всеки друг TCP порт.

Комуникацията по HTTP се състои от заявка (request) – съобщение от клиента към сървъра, и отговор (response) – отговор на сървъра на съобщението от клиента. HTTP заявките имат 3 основни елемента: метод на достъп, Request-URI и header-полета.

Методът описва вида на HTTP заявката, изпратена от клиента. Най-често използваните методи са GET и POST. Чрез GET клиентът изисква някакъв ресурс от Web сървъра, а POST служи за предаване на данни към сървъра и извличане на ресурс. Идентификаторът

Request-URI определя ресурса, над който ще оперира заявката. Могат да се използват два вида идентификатори: URI идентификатор или релативен път спрямо главната директория на Web-сървъра. URI (Uniform Resource Identifier) е идентификатор на ресурс, определен или по местоположение чрез URL (Uniform Resource Locator, например http://www.example.com/folder/page.html), или по име чрез URN (Uniform Resource Name). Релативният път спрямо главната директория на Web-сървъра задава местоположението на ресурс в рамките на текущия Web-сървър. Това е частта от URL-а, която стои след името на хоста (сървъра) в URL идентификатора, например /folder/page.html.

21.1. Бази от данни: Релационен модел.

Релационен модел на данните: домейн; релация; кортеж; атрибути; схема на релация; схема на релационна база от данни; реализация на релационната база от данни; видове операции върху релационната база от данни; заявки към релационната база от данни. Релационна алгебра: основни (обединение; разлика; декартово произведение; проекция; селекция) и допълнителни (сечение; частно; съединение; естествено съединение) операции.

Релационен модел на данните: домейн; релация; кортеж; атрибути.

В основата на **релационния модел** стои математическото понятие n-членна **релация**. Всяка релация е множество от елементи, които се състоят от n компоненти и се наричат n-торки. Всъщност чрез една релация се моделира даден клас от обекти, а всяка n-торка от релацията представлява конкретен обект от този клас.

Предимства на релационния модел пред другите модели на данни:

- Простота на модела – възможност за опериране с релациите както с числата, като се използват операциите на т. нар. **релационна алгебра**
- Висока степен на независимост на данните (няма необходимост от познаване на вътрешното представяне на данните и начина на достъп до тях)
- Съответства на начина на човешкото мислене
- Еднообразен начин за представяне на данните и връзките между тях във вид на таблица

Нека D_1, D_2, ..., D_n са множества, не непременно различни. Всяко от тези множества е именувано, разглежда се като множество от допустими стойности на някаква величина и се нарича още **област** или **домейн**. Домейните могат да бъдат безкрайни или да съдържат краен брой елементи. Всяко подмножество на декартовото произведение $D_1 \times D_2 \times ... \times D_n$ на n области се нарича **релация**. В контекста на БД, **релацията** е двумерна таблица, в която се поместват данните. Т.е. релацията може да се разглежда като таблица от елементи, в която редовете са n-торки, а стълбовете съдържат елементи само от една и съща област.

Всеки именуван стълб от табличното представяне на дадена релация е прието да се нарича **атрибут** на релацията. Т.е. атрибутите са означенията (имената) на колоните на релацията (например: *title*, *year*, *length* и *filmType*). С всеки атрибут е асоцииран **домейн** (например: *Movies (title: string, year: integer, length: integer, filmType: string)*). Редовете в релацията (без заглавната част – атрибутите) се наричат **кортежи** (tuples) (например *(Star Wars, 1977, 124, color)*). С други думи, релацията е множество от кортежи, откъдето следва, че никой кортеж не може да се появява повече от веднъж. Релационният модел изисква всеки компонент на кортеж да бъде атомарен (неделим), т.е. се разрешават само прости типове (integer, string), а сложните типове като списъци и масиви са забранени.

Схема на релация; схема на релационна база от данни.

Релациите са такива структури от данни, които са променливи по съдържание и във всеки момент представят текущото състояние на класа от обекти. Съществено в случая е това, че те запазват структурата си през цялото време от съществуването си. Структурата на всяка релация се задава с нейната схема. **Схемата на релацията** се определя от името на релацията и множеството от атрибутите й. Изписваме схемата на релацията, като указваме името й, следвано от атрибутите в скоби – например *Movies (title, year, length, filmType)*.

Независимо от това, че атрибутите в релационната схема са множество, а не списък, ние трябва да укажем стандартна подредба на атрибутите един спрямо друг. В релационния модел проектът се състои от една или няколко релационни схеми. Съвкупността от всичките тези релационни схеми, с които се описват класовете от обекти и връзките между тях, се нарича **релационна схема на базата от данни** (накратко **схема на базата от данни**).

Схема на релация = име на релация + атрибути

База от данни = колекция от релации

Схема на база от данни = множество от схемите на релациите в базата от данни

Множеството от кортежи за дадена релация ще наричаме **екземпляр на релацията**. Екземплярите на релациите се променят с времето. Кардиналността на един екземпляр на дадена релация е равен на текущия брой кортежи в релацията. Всяко множество от конкретни екземпляри на релации се нарича **текущо състояние** на релационната БД.

Реализация на релационната база от данни.

Процесът на проектиране на една релационна база от данни се състои от следните стъпки:

- Определяне на данните (обектите и техните характеристики), които ще се съхраняват;
- Определяне на връзките между обектите;
- Определяне на ключовите и на свързващите колони;
- Определяне на ограниченията върху обектите и връзките между тях;
- Отстраняване на евентуални недостатъци (излишества и пр.);
- Реализиране на базата от данни.

Дефинирането на отделните релационни схеми е твърде свободно, но най-често при създаването им се спазват следните две правила:

- Всеки клас от обекти се представя с релация, чиято схема включва всички негови атрибути, които стават атрибути и на релацията. Всяка n-торка от релацията представя конкретен обект/екземпляр от класа. Ключовият атрибут или списъкът от ключовите атрибути на класа от обекти, т.е. тези, които еднозначно идентифицират отделните екземпляри в класа, се приемат за ключ на релацията.

- Връзките между два или повече класове от обекти се представят чрез релация, чиято релационна схема включва ключовите атрибути на всеки от тези класове от обекти.

Видове операции върху релационната база от данни. Заявки към релационната база от данни.

Както беше посочено, една реална релация не може да бъде статична, а се променя с времето. Възможни са два вида **операции върху релационната база от данни**:

- Selection, insertion, updating, deletion of tuples (често) – **DML** (Data Manipulation Language). Чрез тези операции се извличат, добавят, изтриват и променят данните в БД. DML операциите са SELECT, INSERT, UPDATE и DELETE. След като се изпълни един DML statement, трябва да се направи COMMIT, за да станат промените постоянни. Примери:

```
SELECT *
FROM books
```

```
SELECT books.title, count(*) AS Authors
FROM books
JOIN book_authors
ON books.isbn = book_authors.isbn
WHERE books.price > 100.00
GROUP BY books.title;

INSERT INTO my_table (field1, field2, field3) VALUES ('test', 'N', NULL);

UPDATE my_table SET field1 = 'updated value' WHERE field2 = 'N';

DELETE FROM my_table WHERE field2 = 'N';
```

• Промени в схемата (не много често) – **DDL** (Data Definition Language). Тези операции служат за дефиниране и променяне на структурата на таблиците и другите обекти в базата от данни (редове, колони, индекси и т.н.). Когато се изпълни един DDL statement, промените се прилагат незабавно. DDL операциите са CREATE (за създаване), DROP (за премахване) и ALTER (за промяна). Примери:

```
CREATE TABLE employees (
    id            INTEGER   PRIMARY KEY,
    first_name    CHAR(50)  NULL,
    last_name     CHAR(75)  NOT NULL,
    date_of_birth DATE      NULL
);
DROP TABLE employees;
ALTER TABLE employees ADD place_of_birth CHAR(50);
ALTER TABLE employees DROP COLUMN place_of_birth;
```

Релационна алгебра.

Алгебрата представлява математическа система, при която чрез използването на различни оператори и операнди могат да се получат нови такива. **Релационната алгебра** е такава алгебра, чиито операнди са релации или променливи, представляващи релации. Операторите са проектирани да извършват такива действия, каквито бихме искали да извършваме с релации в една база от данни. РА е в основата на всеки език за заявки към база от данни – всички системи за управление на бази от данни (СУБД) използват РА като междинен език за изчисление на заявките. В зависимост от това дали релациите представляват множества (един кортеж не може да се среща повече от веднъж) или мултимножества (може да се съдържат повторения на кортежи) от кортежи, говорим за **ядро на РА** или за **разширена РА**.

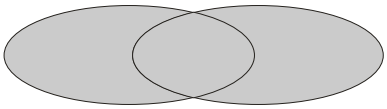
Нека $D_1, D_2, \ldots, D_n$ са n области. Разглеждаме списъка от атрибути $A = A_1, A_2, \ldots, A_n$, при който атрибутът A_i взема стойности от областта D_i, i = 1, 2, …, n. Нека B е друг списък от m атрибута $B_1, B_2, \ldots, B_m$. Релациите $R(A_1, A_2, \ldots, A_n)$ и $S(B_1, B_2, \ldots, B_m)$ са съвместими, ако: 1) са с еднаква степен (т.е. m = n); 2) $dom(A_k) = dom(B_k)$, т.е. A_k и B_k са от един и същи тип за всяко k = 1, 2, …, n; 3) редът на атрибутите в двете релации съвпада.

Основни (обединение; разлика; декартово произведение; проекция; селекция) и допълнителни (сечение; частно; съединение; естествено съединение) операции в релационната алгебра.

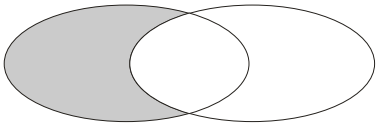
Освен на **основни** и **допълнителни**, операциите в релационната алгебра могат и да се класифицират по следния начин:

1. Операции върху множества – в случай, че R и S са с една и съща релационна схема:

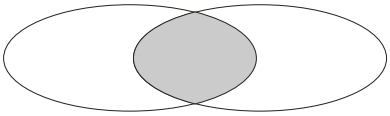
- **<u>Обединение</u>** (union) – бинарна (двучленна), комутативна, асоциативна операция – **R ∪ S**



- **<u>Разлика</u>** (set difference) – бинарна операция; за разлика от другите операции, разликата не е комутативна, т.е. при размяна на местата на операндите резултатът не е един и същ (R – S ≠ S – R) – **R – S**



- **<u>Сечение</u>** (intersection) – бинарна, комутативна, асоциативна операция – **R ∩ S**: Сечението на две релации с обща релационна схема е трета релация със същата релационна схема, която съдържа общите за двете релации кортежи.



Не е задължително да се включва

- **<u>Частно</u>** – бинарна, комутативна, асоциативна операция – **R : S**: Нека са дадени две релации $P(A_1, A_2, \ldots, A_k, B_1, B_2, \ldots, B_m)$ и $Q(B_1, B_2, \ldots, B_m)$. Частното на P и Q е трета релация $R(A_1, A_2, \ldots, A_k)$, която има свойството, че за всяко $t \in R$ и всяко $s \in Q$, конкатенацията $ts \in P$.

<u>Пример</u>:

R			
NAME	**ADDRESS**	**GENDER**	**BIRTHDATE**
Carrie Fisher	123 Maple St., Hollywood	F	9/9/99
Mark Hamil	456 Oak Rd., Brentwood	M	8/8/88

S			
NAME	**ADDRESS**	**GENDER**	**BIRTHDATE**
Carrie Fisher	123 Maple St., Hollywood	F	9/9/99
Harrison Ford	789 Palm Dr., Beverly Hills	M	7/7/77

<u>Резултатите от прилагането на трите основни операции</u>:

R ∪ S			
NAME	**ADDRESS**	**GENDER**	**BIRTHDATE**
Carrie Fisher	123 Maple St., Hollywood	F	9/9/99
Harrison Ford	789 Palm Dr., Beverly Hills	M	7/7/77
Mark Hamil	456 Oak Rd., Brentwood	M	8/8/88

R - S			
NAME	**ADDRESS**	**GENDER**	**BIRTHDATE**
Mark Hamil	456 Oak Rd., Brentwood	M	8/8/88

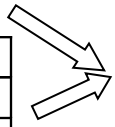
Не е задължително да се включва

$R \cap S$			
NAME	**ADDRESS**	**GENDER**	**BIRTHDATE**
Carrie Fisher	123 Maple St., Hollywood	F	9/9/99

Пример за частно:

MOVIE					
TITLE	**YEAR**	**LENGHT**	**TYPE**	**STUDIO NAME**	**PRODUCER**
Star Wars	1977	124	Color	FOX	Carrie Fisher
Mighty Ducks	1991	104	Color	DISNEY	Emilio Estevez
Wayne's World	1992	95	Color	PARAMOUNT	Dana Carvey

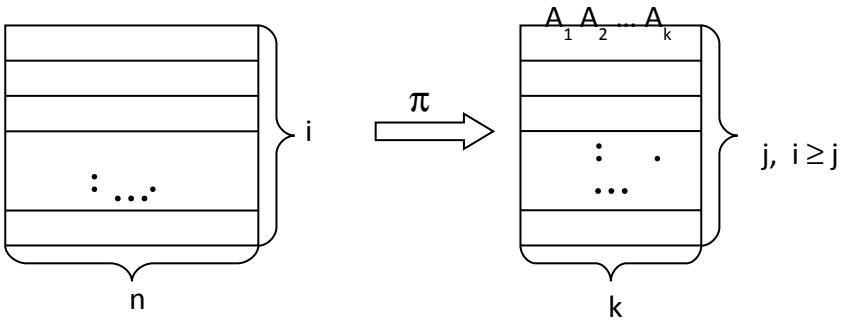
STUDIO	
STUDIO NAME	**PRODUCER**
FOX	Carrie Fisher
DISNEY	Emilio Estevez
PARAMOUNT	Dana Carvey



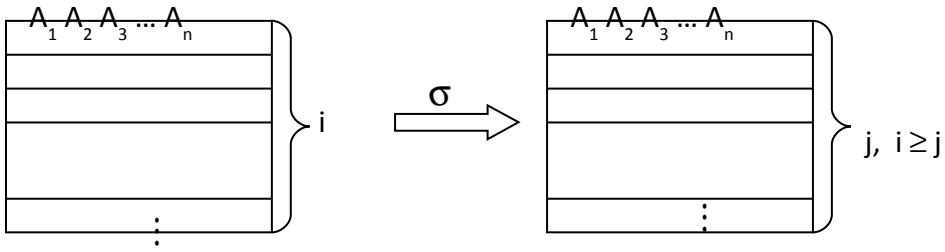
MOVIE:STUDIO			
TITLE	**YEAR**	**LENGHT**	**TYPE**
Star Wars	1977	124	Color
Mighty Ducks	1991	104	Color
Wayne's World	1992	95	Color

2. Операции, които отстраняват части от релациите

- **Проекция** (хоризонтална рестрикция, намаляване в ширина): Нека P е n-членна релация с релационна схема $P(A_1, A_2, \ldots, A_n)$. Проекцията на релацията P по отношение на атрибутите $A_{i1}, A_{i2}, \ldots, A_{ik}$ е k-членна релация $R(A_{i1}, A_{i2}, \ldots, A_{ik})$, която се получава от P чрез отстраняване на всички атрибути, нефигуриращи в схемата на R и в която повтарящите се елементи (кортежи) са представени само по един път. Записва се по следния начин: $\pi_{<attr\_list>}(R)$, където <attr_list> е списък от атрибутите, които ще бъдат резултат от операцията. Въпреки че основният смисъл тук е намаляване в ширина, възможно е и намаляване в дълбочина, когато избраните атрибути имат много повтарящи се стойности, а, тъй като тук се работи само с множества, резултатът е унифициран. Това е един от минусите на работата с множества – може да се получи загуба на информация.



- **Селекция** (рестрикция, вертикална рестрикция, намаляване в дълбочина) – унарна, комутативна операция: Нека P е n-членна релация с релационна схема $P(A_1, A_2, \ldots, A_n)$. Селекцията на релацията P по отношение на дадено условие F е друга релация R със същата релационна схема, всеки елемент на която удовлетворява условието F. Записва се по следния начин: $\sigma_{<predicate>}(R)$, където <predicate> е условието, на което трябва да отговарят атрибутите. Предикатите имат следния вид: **<attribute> <op> <attribute>** или **<attribute> <op> <constant>**, където **<attribute>** са атрибути на R, **<constant>** са константи, а **<op>** са оператори като =, ≠, <, >, ≤, … , AND, OR.



Пример:

MOVIE					
TITLE	**YEAR**	**LENGHT**	**FILM TYPE**	**STUDIO NAME**	**PRODUCER**
Star Wars	1977	124	Color	FOX	Carrie Fisher
Mighty Ducks	1991	104	Color	DISNEY	Emilio Estevez
Wayne's World	1992	95	Color	PARAMOUNT	Dana Carvey

Примери за проекция:

Не е задължително да се включва

$\pi_{\text{title, year, length}}$ (Movie):

TITLE	YEAR	LENGHT
Star Wars	1977	124
Mighty Ducks	1991	104
Wayne's World	1992	95

$\pi_{\text{film type}}$ (Movie):

FILM TYPE
Color

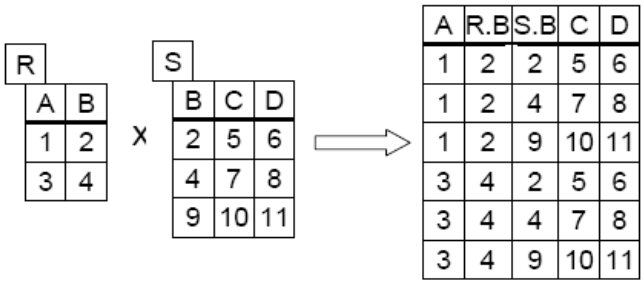
Пример за селекция:

$\sigma_{\text{length}\geq 100}$ (Movie)

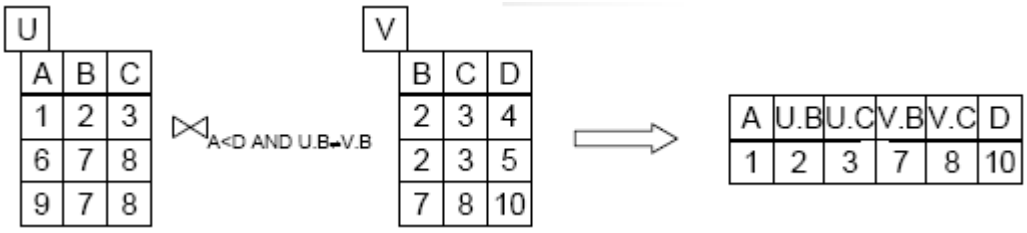
MOVIE					
TITLE	**YEAR**	**LENGHT**	**FILM TYPE**	**STUDIO NAME**	**PRODUCER**
Star Wars	1977	124	Color	FOX	Carrie Fisher
Mighty Ducks	1991	104	Color	DISNEY	Emilio Estevez

3. Операции, които комбинират кортежи от две или повече релации

- **Декартово произведение** (Cartesian product) – бинарна, комутативна, асоциативна операция – **R x S**: Нека са дадени две релации $P(A_1, A_2, \ldots, A_m)$ и $Q(B_1, B_2, \ldots, B_n)$. Декартово произведение на P и Q е трета (m+n)-членна релация $R(A_1, A_2, \ldots, A_m, B_1, B_2, \ldots, B_n)$, съдържаща всички възможни конкатенации на елементи от P и Q. Ако в P и Q има атрибут с едно и също име (например A), в резултата се записва P.A и Q.A за уточняване.



- **Съединение** (θ-съединение, theta join) – **R** $\triangleright\!\triangleleft_{\mathbf{c}}$ **S**: Съединение на релациите $P(A_1, A_2, \ldots, A_m)$ и $Q(B_1, B_2, \ldots, B_n)$ по отношение на условието C е (m+n)-членна релация с релационна схема $R(A_1, A_2, \ldots, A_m, B_1, B_2, \ldots, B_n)$, която е такова подмножество на декартовото произведение P x Q, в което всеки кортеж отговаря на условието C. Този вид съединение се извършва на два етапа: първо се прави декартово произведение на двете релации; след това се прави селекция върху получената релация по отношение на условието C.



- **Еквисъединение** (equijoin) – частен случай на θ-съединението, когато θ е отношението равенство (=), т.е. условието на съединението включва само съвпадение по атрибутите.

- **Естествено съединение** (natural join) – **R** $\triangleright\!\triangleleft$ **S**: Разширение на еквисъединението, при което се извършва свързване по всички атрибути с еднакви имена. Автоматично се отстранява повтарящата се колона.

Пример:

Sells

BAR	BEER	PRICE
Joe's	Bud	2.50
Joe's	Miller	2.75
Sue's	Bud	2.50
Sue's	Coors	3.00

⋈

Bars

BAR	ADDRESS
Joe's	Maple St.
Sue's	River Rd.

⟹

Bar Info

BAR	BEER	PRICE	ADDRESS
Joe's	Bud	2.50	Maple St
Joe's	Miller	2.75	Maple St
Sue's	Bud	2.50	River Rd.
Sue's	Coors	3.00	River Rd.

4. **Преименуване** (rename) – тези операции не оказват влияние върху конструкцията на релацията, а само променят имената на таблиците или на техните атрибути. Операторът **ρ** дава нова схема на релацията. **R1 := $\rho_{R1(A1,...,An)}$ (R2)** превръща R1 в релация с атрибути A_1, ..., A_n и същите кортежи като на R2. Записва се още R1(A_1, ..., A_n) := R2.

Пример: R(bar, addr) := Bars

Bars

Name	Address
Joe's	Maple St.
Sue's	River Rd.

⟹

R

Bar	Addr
Joe's	Maple St.
Sue's	River Rd.

Представяне на допълнителните операции чрез основните.

- Сечение: **$R \cap S = R - (R - S)$**
- Съединение: **$R \triangleright\triangleleft_C S = \sigma_C (R \times S)$**
- Естествено съединение: **$R \triangleright\triangleleft S = \pi_L (\sigma_C (R \times S))$**, където **L** е списък от атрибутите на R, последвани от атрибутите на S, които не присъстват в R; а **C** е условие от вида $R.A_1 = S.A_1$ AND $R.A_2 = S.A_2$ AND … $R.A_n = S.A_n$, където A_1, A_2, …, A_n са всички общи за R и S атрибути.

Приоритет на релационните оператори.

- Унарни оператори – селекция, проекция, преименуване
- Декартово произведение и съединение
- Сечение
- Обединение и разлика

21.2. Бази от данни: Нормални форми.

Нормални форми. Проектиране схемите на релационните бази от данни. Аномалии, ограничения, ключове. Функционални зависимости, аксиоми на Армстронг. Първа, втора, трета нормална форма, нормална форма на Бойс-Код. Многозначни зависимости; аксиоми на функционалните и многозначните зависимости; съединение без загуба; четвърта нормална форма.

Аномалии.

Да разгледаме един пример, с който се илюстрират различни недостатъци на една конкретна релационна схема, известни като **аномалии** на излишество, обновяване, добавяне и отстраняване.

Дадена е релационната схема *БИБЛИОТЕКА (Биб#, Адрес, Книга, Брой)*, с която всяка библиотека е представена чрез своя номер (*Биб#*) и адрес (*Адрес*), а всяка книга е представяна с името си (*Книга*) и броя екземпляри (*Брой*), които са налични в съответната библиотека. Естествено е да се приеме, че всяка библиотека е еднозначно определена от своя номер, както и, че една и съща книга може да се съхранява в няколко библиотеки. Като следствие от това могат да се направят следните изводи:

1. Аномалия от излишество – За всяка книга от една и съща библиотека ще има повторение на адреса на библиотеката.

2. Аномалия при обновяване – От (1) следва, че при промяна на адреса на библиотеката ще е необходимо да се обновят толкова реда на релацията, колкото са книгите, които се съхраняват в нея. При тази операция не е изключена възможността някои от редовете да останат необновени, при което ще настъпи противоречивост в данните на БД.

3. Аномалия при добавяне – Адресът на една новосъздаваща се библиотека не може да бъде въведен в БД, докато не се посочи поне една книга, която ще се съхранява в тази библиотека. Причината за това е, че ключовите атрибути в тази релационна схема са атрибутите *Биб#* и *Книга* и следователно те трябва да имат точно определени стойности за всеки елемент от релацията.

4. Аномалия при изтриване – Ако поради ремонт на дадена библиотека книгите, които са се съхранявали в нея, бъдат преместени в друго книгохранилище или библиотека, ще се наложи всички редове от релацията, отнасящи се до тази библиотека, да бъдат отстранени от релацията. Като следствие от това ще бъде "изгубен" и адресът на библиотеката, независимо че тя съществува и адресът й остава същият.

Ключове.

Понеже всяка релация е множество от неповтарящи се n-торки, то всяка релация има **ключ**, който определя еднозначно кортежите в релацията. Естествено е една релация да има и повече от един ключ. Ако едно множество от атрибути A е ключ за една релация R, то всяко съдържащо A множество X също ще идентифицира еднозначно елементите на R. Затова към всеки ключ се предявява и изискването за минималност: Ако K е ключ на релацията R, тогава: 1) За всеки два елемента $r_1 \in R$ и $r_2 \in R$ е в сила $r_1[K] \neq r_2[K]$; 2) Не съществува подмножество от атрибути на K, за което (1) да остава в сила.

Един от всички възможни ключове на всяка релация се избира за ключ на релацията и той се нарича **<u>първичен ключ</u>**.

<u>Външният ключ</u> K* за една релация R е такъв атрибут или списък от атрибути, който не е ключ за R, но съществува релация Q, за която K* е първичен ключ.

Ограничения.

<u>Ограниченията</u> дефинират правила за стойностите, допустими за съответните колони, и представляват стандартен механизъм за налагане на цялост. С други думи, ограничението е свойство на таблица или колона от таблица, което предотвратява въвеждането на невалидни стойности на данните. Ето основните ограничения в релационния модел:

Може по-накратко да се споменат

- Ограниченията UNIQUE и PRIMARY KEY (първичен ключ) не допускат да бъде въведена стойност, която дублира съществуваща стойност;
- Стойностите на атрибутите, съставящи първичния ключ, не могат да бъдат нулеви;
- Ограничението CHECK не допуска да бъде въведена стойност, която не отговаря на определени условия (например Age > 0 или length(EGN) = 10);
- Ограничението FOREIGN KEY (външен ключ) налага връзка между данните в две таблици – за всяка ненулева стойност на атрибут, който е външен ключ, трябва да съществува ключов атрибут от друга релация, който съдържа тази стойност;
- Ограничението NOT NULL задава, че не се допуска стойността на атрибут (не непременно ключов) да е нулева (null). Стойност null не означава нула, интервал или символен низ с нулева дължина, като "". Null означава, че не са били въведени данни. Наличието на null обикновено означава, че стойността е или неизвестна, или недефинирана.

Освен тези ограничения, друг широк клас от ограничения за цялостност се въвеждат с помощта на функционални зависимости.

Функционални зависимости.

Нека A_1, A_2, …, A_n е един какъв да е списък от n атрибута, n>=1, който накратко ще означаваме с $A_1A_2...A_n$ (без запетайки) или само с A. Предполага се, че списъкът A е неподреден и няма повтарящи се елементи, т.е. A е множество от имена на атрибути. Знае се, че ако A има n елемента, то броят на подмножествата на A е 2^n.

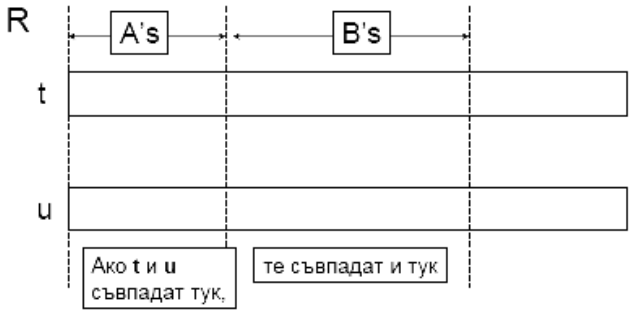
Нека r е релация, а R(A) е нейната релационна схема. r(R) е математическа релация от степен n върху домейните $dom(A_1)$, $dom(A_2)$, …, $dom(A_n)$ – подмножество на декартовото произведение на домейните, които дефинират R. Това може да се изрази чрез $r(R) \subseteq (dom(A_1) \times dom(A_2) \times \ldots \times dom(A_n))$. r(R) е множество от n-торки (tuples), $r = \{t_1, t_2, \ldots, t_m\}$. Всяка n-торка е подреден списък от n стойности $t = <v_1, v_2, \ldots, v_n>$, където v_i, $1 \le i \le n$, е елемент от $dom(A_i)$ или специалната стойност NULL.

Нека $X, Y \in A$. Казва се, че атрибутът X **<u>функционално определя</u>** атрибута Y, ако за всяка релация r с релационна схема R(A) при наличието на кои да е два кортежа $t_1, t_2 \in r$, за които $t_1.X = t_2.X$, следва, че $t_1.Y = t_2.Y$, т.е. при равенство на кортежи по атрибута X следва равенството им по атрибута Y. X и Y могат да бъдат както прости атрибути, така и съставни, т.е. да се състоят от няколко прости атрибута (ако $A_1A_2...A_n \rightarrow B_1$ и $A_1A_2...A_n \rightarrow B_2$ и ....

$A_1A_2...A_n \rightarrow B_m$, то $A_1A_2...A_n \rightarrow B_1B_2...B_m$). Когато X функционално определя Y, казва се също, че Y **<u>зависи функционално</u>** от X или че е налице **<u>функционална зависимост</u>** (ФЗ) от X в Y, като се използва означението R.X → R.Y. Множеството от ФЗ се означава с F, а с малка буква f се означава единична ФЗ. Зависимостта $A_1A_2...A_n \rightarrow B$ се нарича <u>функционална</u>, защото има функция, която на списък от стойности (по една за всяко $A_1A_2...A_n$) съпоставя уникална стойност за B. Тук функцията обаче не се изчислява по стандартния начин, а „изчислението" става чрез търсене в релацията.

ФЗ са твърдения за схемата на релацията, не за конкретен екземпляр, т.е. те са свойства на семантиката на атрибутите и всички данни ги удовлетворяват. Затова те не могат да се определят чрез просто преглеждане на данните.

Графично представяне на ФЗ A → B:



Например, в релацията Movies се наблюдават следните функционални зависимости: *title year → length*; *title year → filmType*; *title year → studioName*; но НЕ! title year → starName.

K = {A_1, A_2, ..., A_n} е **<u>ключ</u>** на релацията R, ако:

1. Множеството K <u>функционално определя</u> всички атрибути на R;
2. За нито едно подмножество на K не е вярно (1).

Ако K удовлетворява (1), но не удовлетворява (2), то K е **<u>суперключ</u>**.

Например, за релацията Movies (title, year, length, filmType, studioName, starName) ключът е *{title, year, starName}* и няма други ключове, но има много суперключове – всички супермножества на {title, year, starName}, например {title, year, starName, length, type}.

Аксиоми на Армстронг.

Зависимостите, които е възможно да се определят в рамките на една релационна схема, макар и краен брой, понякога могат да бъдат твърде много. Затова естествено е да се постави въпросът, как по дадено множество от ФЗ F могат да се получат всичките ФЗ.

Една ФЗ X → Y се нарича <u>логическо следствие</u> на множеството от ФЗ F, ако за всяка релация r, удовлетворяваща зависимостите F, следва, че е X → Y се удовлетворява от r. **<u>Армстронг</u>** показва, че като се използват т.нар. правила или **<u>аксиоми за извод</u>** при дадено множество от ФЗ, се получават нови ФЗ, при това могат да се получат всичките ФЗ. При формулировката на аксиомите, дадени по-долу, се предполага, че: R(A) е релационна схема; F е множество от ФЗ в A; X, Y, Z и W са списъци от атрибути на A; r е произволна релация със схема R(A).

A1. <u>Рефлексивност</u> (reflexivity): Ако $Y \subseteq X$, то X → Y.

Пример: *title year → title*

A2. Разширение, попълнение (augmentation): Ако X → Y, то XW → YW.

Пример: От *title year* → *length* получаваме *title year filmType* → *length filmType*

A3. Транзитивност (transitivity): Ако X → Y и Y → Z, то X → Z.

Може да не се доказват

Доказателства на аксиомите на Армстронг:

A1. Рефлексивност: Всеки 2 кортежа t и u съвпадат по всички атрибути на X, ⇒ те съвпадат и по всяко подмножество на X, включително Y.

A2. Разширение, попълнение: Да допуснем, че има кортежи t и u, които съвпадат по всички атрибути на XW, но не съвпадат по YW. t и u задължително съвпадат по W, ⇒ се различават по някой от атрибутите на Y, което противоречи на X → Y.

A3. Транзитивност: Да допуснем, че има 2 кортежа (x, y_1, z_1) и (x, y_2, z_2), които съвпадат по всички атрибути на X. X → Y, следователно, щом съвпадат по всички атрибути на X, задължително съвпадат по всички атрибути на Y, т.е. $y_1 = y_2$. Аналогично, от Y → Z следва, че $z_1 = z_2$. ⇒ двата кортежа съвпадат.

Следствия от аксиомите на Армстронг:

Сл.1. Обединение: Ако X → Y и X → Z, то X → YZ.

Сл.2. Псевдотранзитивност: Ако X → Y и WY → Z, то XW → Z.

Сл.3. Декомпозиция: Ако X → Y и Z ⊆ Y, то X → Z.

Може без док-ва

Доказателства на следствията от аксиомите на Армстронг:

Сл.1. Обединение: X → Y, следователно X → XY (A2). X → Z, ⇒ XY → ZY (A2). От получените X → XY и XY → ZY следва, че X → ZY (A3).

Сл.2. Псевдотранзитивност: X → Y, следователно WX → WY (A2). Но имаме, че WY → Z, следователно WX → Z (A3).

Сл.3. Декомпозиция: Z ⊆ Y, следователно Y → Z (от A1). Но имаме, че X → Y, следователно X → Z (A3).

Правило за декомпозиция: Ако $AA \to B_1B_2...B_n$, то $AA \to B_1$, $AA \to B_2$, ..., $AA \to B_n$.

Правило за обединение: Ако $AA \to B_1$, $AA \to B_2$, ..., $AA \to B_n$, то $AA \to B_1B_2...B_n$.

Функционалната зависимост $A_1A_2...A_n \to B$ се нарича тривиална, ако атрибутът B съвпада с някой от атрибутите $A_1A_2...A_n$. В противен случай се нарича нетривиална.

Когато B е съставен атрибут, т.е. $B = B_1B_2...B_m$, то ФЗ $A_1A_2...A_n \to B_1B_2...B_m$ е:

- тривиална, ако $B_1B_2...B_m \subseteq A_1A_2...A_n$;
- нетривиална, ако поне един атрибут от $B_1B_2...B_m$ не е от $A_1A_2...A_n$.
- напълно нетривиална, ако никой от атрибутите от $B_1B_2...B_m$ не е от $A_1A_2...A_n$.

Правило на тривиалната зависимост: Имаме право от дясната част на една ФЗ да премахнем тези атрибути, които принадлежат на лявата част – така от $A_1A_2...A_n \to B_1B_2...B_m$ получаваме $A_1A_2...A_n \to C_1C_2...C_k$, където $\{C_1, C_2, ..., C_k\} \subseteq \{B_1, B_2, ..., B_m\}$ и нито един от атрибутите C не е от $A_1A_2...A_n$.

Първа, втора, трета нормална форма, нормална форма на Бойс-Код.

Виждаме, че основен проблем при изграждането на модели на дадена предметна област е определянето на същностите и на свойствата, които ги характеризират. Сложността на проблема е следствие от нееднозначността на неговото решение. Естествено е в такъв случай да се поставят въпроси като: кога една релационна схема е добра; кога две схеми са еквивалентни или коя от двете схеми е по-добрата. Всеки от тези въпроси всъщност се докосва до необходимостта от формален инструмент за анализ на релационни схеми. Такъв инструмент е т.нар. **нормализация** – процес, насочен към преобразуването на релационни схеми, при който на новополучените релационни схеми се налагат известни ограничения, елиминиращи някои нежелани свойства.

Една релационна схема е в **първа нормална форма** (1НФ), ако областите на съставящите я атрибути са атомарни (прости), т.е. атрибутите й са атомарни. Иска се и, естествено, да няма повтарящи се кортежи, което е основно изискване за една релация.

Един атрибут X от релационната схема R(A) с множество от ФЗ F се нарича първичен атрибут, ако влиза в състава на ключа (първичния ключ). В противен случай се нарича непървичен. Една релационна схема е във **втора нормална форма** (2НФ) относно множеството от ФЗ F, ако тя е в 1НФ и всеки непървичен атрибут е в пълна ФЗ от ключа, т.е. зависи от целия ключ, а не от някакво негово подмножество.

Нека R(A) е релационна схема със съответно множество от ФЗ F, а X и Z са подмножества на A. Казва се, че атрибутът Z е транзитивно зависим от X, ако съществува такова множество от атрибути Y, $Y \subset A$, $Z \notin XY$, за което са в сила свойствата: $X \rightarrow Y$, $Y \rightarrow Z$. Една релационна схема R(A) е в **трета нормална форма** (3НФ) относно множеството от ФЗ F, ако тя е в 1НФ и нито един първичен атрибут не е транзитивно зависим от ключа (а всеки непървичен атрибут е не-транзитивно, т.е. директно зависим от ключа). Друга дефиниция на **3НФ**: R е в 3НФ тогава и само тогава, когато за всяка нетривиална ФЗ или лявата страна е суперключ, или дясната е първичен атрибут.

В определението на 2НФ се иска наличие на 1НФ, т.е. всяка релационна схема, която е във 2НФ, е и в 1НФ. В определението на 3НФ не се изисква релационната схема да е във 2НФ, но може да се покаже, че всяка релационна схема, която е в 3НФ, е и във 2НФ.

Релацията R е в **нормална форма на Бойс-Код** (Boyce-Codd Normal Form, BCNF) тогава и само тогава, когато за всяка нетривиална зависимост $A_1A_2...A_n \rightarrow B_1B_2...B_m$ от R, съответното множество от атрибути $\{A_1, A_2, ..., A_n\}$ е суперключ за R.

Многозначни зависимости

Терминът "**многозначна зависимост**" се използва, когато два атрибута или множество атрибути са независими помежду си. Това състояние обобщава идеята за ФЗ в този смисъл, че всяка функционална предполага съответна многофункционална зависимост.

Съществуват схеми в BCNF, които съдържат излишни данни. Най-често това се получава при опит да се поставят две или повече връзки "много към много" в една релация. Това излишество е резултат от независимостта на атрибутите. Например, в Tutor/Student Cross-Reference без проблем могат да се въведат два различни номера на социална осигуровка за един и същ ръководител, а това не е желателно.

Пример:

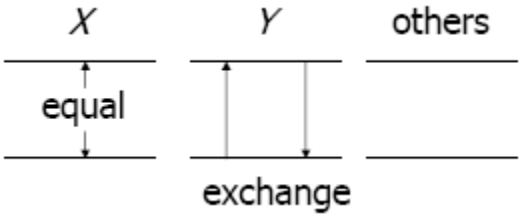
Ще предположим, че известните личности имат по няколко адреса. Разделяме тези адреси на части (град, улица). Редом до информацията за "звездите" ще включим и познатата ни информация за заглавията и годините на филмите, в които са участвали те.

Name	**Street**	**City**	**Title**	**Year**
Carrie Fisher	123 Maple Str	Hollywood	Star Wars	1977
Carrie Fisher	5 Locust Ln.	Malibu	Star Wars	1977
Carrie Fisher	123 Maple Str	Hollywood	Empire Strikes Back	1980
Carrie Fisher	5 Locust Ln.	Malibu	Empire Strikes Back	1980
Carrie Fisher	123 Maple Str	Hollywood	Return or the Jedi	1983
Carrie Fisher	5 Locust Ln.	Malibu	Return or the Jedi	1983

Нека обърнем внимание на двата адреса и трите най известни филма на Carrie Fisher. Няма причина да асоцииряме определен адрес с определен филм, а не с друг филм. Поради това единственият начин да покажем, че адресите и филмите са независими свойства, е да добвим всеки адрес към всеки филм. Но, когато повторим адресите и данните за филмите във всички комбинации, се получава явно излишество на данни. Например, всеки от адресите на Carrie Fisher се повтаря по три пъти (веднъж за всеки от филмите) и всеки филм се повтаря по два пъти (веднъж за всеки от адресите).

Въпреки това, няма нарушение на BCNF в тази релация. Няма и никакви нетривиални ФЗ. Например атрибутът City не е функционално определен от другите 4 атрибута. Може да има звезда с два дома, които имат един и същ адрес на улица в различни градове. Тогава ще има два кортежа, чиито атрибути си съответстват напълно, освен по атрибута City. Така че *name street title year* → *city* НЕ е ФЗ за нашата релация. Никой от петте атрибута не е функционално определен от другите четири. Тъй като няма нетривиални ФЗ, следва, че всички пет атрибута формират един суперключ и няма нарушение на BCNF.

Многозначна зависимост (multivalued dependency, MVD, МЗ) е твърдение за релация R, за която, при фиксиране на стойностите за определени атрибути, стойностите в точно определени други атрибути са независими от стойностите на всички други атрибути в релацията. **Многозначната зависимост** $X \rightarrow\rightarrow Y$ утвърждава, че, ако 2 кортежа в една релация съвпадат по всички атрибути на X, техните компоненти от множеството атрибути Y могат да бъдат разменени и резултатът ще даде 2 нови кортежа, които също принадлежат на релацията.



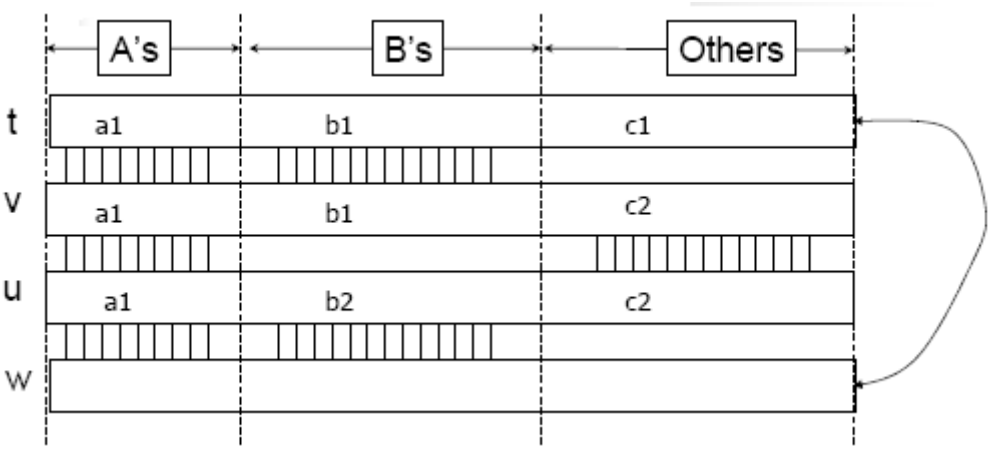
Дефиниция: $A_1A_2\ldots A_n \rightarrow\rightarrow B_1B_2\ldots B_m$ е **многозначна зависимост** в R, ако за всяка двойка кортежи t, u от R, за които $t[A_1A_2\ldots A_n] = u[A_1A_2\ldots A_n]$, съществува кортеж v от R, за който:

- $v[A_1A_2\ldots A_n] = t[A_1A_2\ldots A_n] = u[A_1A_2\ldots A_n]$
- $v[B_1B_2\ldots B_m] = t[B_1B_2\ldots B_m]$
- $v[C_1C_2\ldots C_k] = u[C_1C_2\ldots C_k]$,

където $C_1C_2\ldots C_k$ са всички атрибути от R, с изключение на $(A_1A_2\ldots A_n \cup B_1B_2\ldots B_m)$.

Можем да използваме това правило с разменени t и u, за да подскажем съществуването на четвърти кортеж w, който се съгласува с u и t. В последствие за всички фиксирани

стойности на A съответните стойности от B и останалите атрибути се появяват във всички възможни комбинации в различни кортежи.



Пример: name →→ street city

Name	Street	City	Title	Year
C. Fisher*	123 Maple Str	Hollywood	Star Wars*	1977
C. Fisher*	5 Locust Ln.*	Malibu*	Star Wars*	1977
C. Fisher	123 Maple Str	Hollywood	Empire Strikes Back	1980
C. Fisher*	5 Locust Ln.*	Malibu*	Empire Strikes Back	1980
C. Fisher	123 Maple Str	Hollywood	Return or the Jedi	1983
C. Fisher	5 Locust Ln.	Malibu	Return or the Jedi	1983

Аксиоми на многозначните зависимости.

МЗ $A_1A_2...A_n \rightarrow\rightarrow B_1B_2...B_m$ се нарича **тривиална**, когато $B_1B_2...B_m \subseteq A_1A_2...A_n$ или ($A_1A_2...A_n \cup B_1B_2...B_m$) съдържа всички атрибути на R.

МЗ $A_1A_2...A_n \rightarrow\rightarrow B_1B_2...B_m$ е **нетривиална**, когато нито един от атрибутите $B_1B_2...B_m$ не съвпада с $A_1A_2...A_n$ и не всички атрибути на R принадлежат на ($A_1A_2...A_n \cup B_1B_2...B_m$).

Има няколко правила за МЗ, които са подобни на тези за ФЗ:

Пр.1. Транзитивно правило: Ако $A_1A_2...A_n \rightarrow\rightarrow B_1B_2...B_m$ и $B_1B_2...B_m \rightarrow\rightarrow C_1C_2...C_k$, то $A_1A_2...A_n \rightarrow\rightarrow C_1C_2...C_k$.

Пр.2. Правило на обединението: Ако $X_1X_2...X_n \rightarrow\rightarrow Y_1Y_2...Y_m$ и $X_1X_2...X_n \rightarrow\rightarrow Z_1Z_2...Z_k$, то $X_1X_2...X_n \rightarrow\rightarrow (Y_1Y_2...Y_m \cup Z_1Z_2...Z_k)$.

Пр.3. Правило на допълнението: Ако $A_1A_2...A_n \rightarrow\rightarrow B_1B_2...B_m$, то $A_1A_2...A_n \rightarrow\rightarrow C_1C_2...C_k$, където $C_1C_2...C_k$ е м-то от всички атрибути на R с изключение на ($A_1A_2...A_n \cup B_1B_2...B_m$).

Пример: От name →→ street city, съгласно правилото на допълнението name →→ title year също трябва да е в сила за тази релация, защото title и year са атрибути, които не са споменати в първата МЗ. Втората МЗ означава, че всяка всяка известна личност е участвала в няколко филма, които са независими от адресите на звездата.

Забележка: Подобно на ФЗ, не може да се разделя лявата част на МЗ. За разлика от ФЗ обаче, не може да се разделя и дясната част – понякога се налага да се оставят няколко атрибута в дясната част. Например, ако върху МЗ name →→ street city се приложи правилото за разделяне, тогава трябва да очакваме, че е вярно и следното: name →→

street. Тази МЗ показва, че всеки адрес на звезда е независим от другите атрибути, включително и от града. Обаче това твърдение не е вярно. Например – първите 2 кортежа. Предполагаемата МЗ ще ни позволи да загатнем, че кортежите с улиците са се разменили:

Name	**Street**	**City**	**Title**	**Year**
C. Fisher	5 Locust Ln.	Hollywood	Star Wars	1977
C. Fisher	123 Maple Str	Malibu	Star Wars	1977

Но тези кортежи не са правилни, защото домът на 5 Locust Ln. се намира в Малибу, а не в Холивуд.

Пр.4. Правило FD-IS-AN-MVD (всяка функционална зависимост е многозначна зависимост):
Ако $A_1A_2...A_n \rightarrow B_1B_2...B_m$ то: $A_1A_2...A_n \rightarrow\rightarrow B_1B_2...B_m$.

Доказателство: Да предположим, че R е релация, за която е вярна ФЗ $A_1A_2 \ldots A_n \rightarrow B_1B_2 \ldots B_m$, и да предположим, че t и u са кортежите, които съвпадат в A. За да покажем, че е вярна МЗ $A_1A_2 \ldots A_n \rightarrow\rightarrow B_1B_2 \ldots B_m$, трябва да докажем, че R също съдържа кортеж v, който се съгласува с t и u по A, с t по B и с u по всички останали атрибути. Но v може да е u. Със сигурност u съвпада с t и u в A. ФЗ $A_1A_2 \ldots A_n \rightarrow B_1B_2 \ldots B_m$ предполага, че u съвпада с t в B. И разбира се u се съгласува със себе си по другите атрибути. Така че, когато е в сила функционалната зависимост, то е в сила и многозначната зависимост.

Декомпозиция на релации. Съединение без загуба.

Декомпозиция на релацията $R(A_1, \ldots, A_n)$ е заместването ѝ с множество релации $R_1, \ldots, R_n$, получени чрез проекции така, че R и $R_1 \cup R_2 \cup \ldots \cup R_n$ имат една и съща схема.

Например, дадена е релация R със схема $\{A_1, A_2, \ldots, A_n\}$. За да няма загуба на информация и да се намери най-оптималното решение за представяне, трябва да се декомпозира в две релации S и T със схеми $\{B_1, B_2, \ldots, B_m\}$ и $\{C_1, C_2, \ldots, C_k\}$, така, че: $\{A_1, A_2, \ldots, A_n\} = \{B_1, B_2, \ldots, B_m\} \cup \{C_1, C_2, \ldots, C_k\}$. Кортежите в релацията S са проекции на всички кортежи в R върху $\{B_1, B_2, \ldots, B_m\}$ (за всеки кортеж t от текущия екземпляр на R се избират компонентите, които съответстват на атрибутите $B_1, B_2, \ldots, B_m$; тези компоненти образуват нов кортеж, който принадлежи на текущия екземпляр на S; прави се само по 1 копие на кортеж). Аналогично, кортежите в релацията T са проекции на всички кортежи в R върху атрибутите $\{C_1, C_2, \ldots, C_k\}$.

Нека $R (A_1, A_2, \ldots, A_n)$ е релация и R се декомпозира на две релации $S (B_1, B_2, \ldots, B_m)$ и $T (C_1, C_2, \ldots, C_k)$. Казваме, че декомпозицията е със **съединение без загуба**, ако $R = S \bowtie T$.

Декомпозицията на релацията $R (A_1, A_2, \ldots, A_n)$ на 2 релации $S (B_1, B_2, \ldots, B_m)$ и $T (C_1, C_2, \ldots, C_k)$ е със съединение без загуба тогава и само тогава, когато за R е изпълнена поне една от следните функционални зависимости: $S \cap T \rightarrow S$ или $S \cap T \rightarrow T$.

Нека $R (A_1, A_2, \ldots, A_n)$ е релация, за която е изпълнено множеството от функционални зависимости F и R се декомпозира на две релации $S (B_1, B_2, \ldots, B_m)$ и $T (C_1, C_2, \ldots, C_k)$. Казваме, че декомпозицията е със **съединение без загуба на функционалните зависимости**, ако $F_1 \cup F_2 = F$, където F_1 и F_2 са проекциите на F съответно върху S и T.

Чрез подходящи декомпозиции, всяка схема на релация може да се декомпозира на няколко схеми, така че да са изпълнени следните условия: 1) Всички получени релации да са в BCNF; 2) Декомпозицията да е със съединение без загуба.

Стратегията за декомпозиция, която възприемаме, е следната.

Нека $A_1\ A_2\ \dots\ A_n \rightarrow B_1\ B_2\ \dots\ B_m$ е нетривиална функционална зависимост и $\{A_1, A_2, \dots, A_n\}$ не е суперключ. Тогава декомпозираме релацията R на следните две релации:

1. Първата релация има атрибути $A_1, A_2, \dots, A_n, B_1, B_2, \dots, B_m$.
2. Втората релация има атрибути $A_1, A_2, \dots, A_n$ и всички останали атрибути на R, които не участват във функционалната зависимост.

Ако новополучените релации не са в BCNF, то към тях прилагаме същата процедура. При това, функционалните зависимости в новите релации се изчисляват чрез проектиране на ФЗ от изходната релация (т.е. функционалните зависимости в новите релации са точно онези, които следват от предишните ФЗ и в които участват само новите атрибути).

Процесът на декомпозиране ще е краен, тъй като винаги получаваме релации с по-малко атрибути, а всяка релация с два атрибута е в BCNF.

В общия случай декомпозицията в BCNF не е със съединение без загуба на функционалните зависимости. Съществува обаче алгоритъм за декомпозиция в 3НФ, който е със съединение без загуба и запазва ФЗ. Този алгоритъм в повечето случаи се справя с излишествата, породени от фукционални зависимости.

Не е задължително да се включва

Пример за съединение със загуба:

T	Employee	Project	Branch
	Brown	Mars	L.A.
	Green	Jupiter	San Jose
	Green	Venus	San Jose
	Hoskins	Saturn	San Jose
	Hoskins	Venus	San Jose

Функционалните зависимости са: Employee → Branch и Project → Branch.

Декомпозиция на релацията T:

T1	Employee	Branch
	Brown	L.A.
	Green	San Jose
	Hoskins	San Jose

T2	Project	Branch
	Mars	L.A.
	Jupiter	San Jose
	Saturn	San Jose
	Venus	San Jose

След прилагане на естествено съединение резултатът е различен от първоначалната релация и информацията не може да бъде възстановена:

T	Employee	Project	Branch
	Brown	Mars	L.A.
	Green	Jupiter	San Jose
	Green	Venus	San Jose
	Hoskins	Saturn	San Jose
	Hoskins	Venus	San Jose
	Green	Saturn	San Jose
	Hoskins	Jupiter	San Jose

Четвърта нормална форма.

Излишеството, което произтича от МЗ, не може да се отстрани чрез привеждане в НФ на Бойс-Код. Необходима е по-строга НФ, наречена **<u>четвърта нормална форма</u>** (4НФ), която третира МЗ като ФЗ по отношение на декомпозицията, но не и по отношение на ключовете. В тази НФ всички МЗ за елиминирани, както всички ФЗ, които нарушават BCNF. В резултат, декомпозираните релации нямат излишък нито от ФЗ, нито от МЗ. В основата си четвъртата нормална форма е BCNF, но е приложена върху МЗ вместо върху ФЗ.

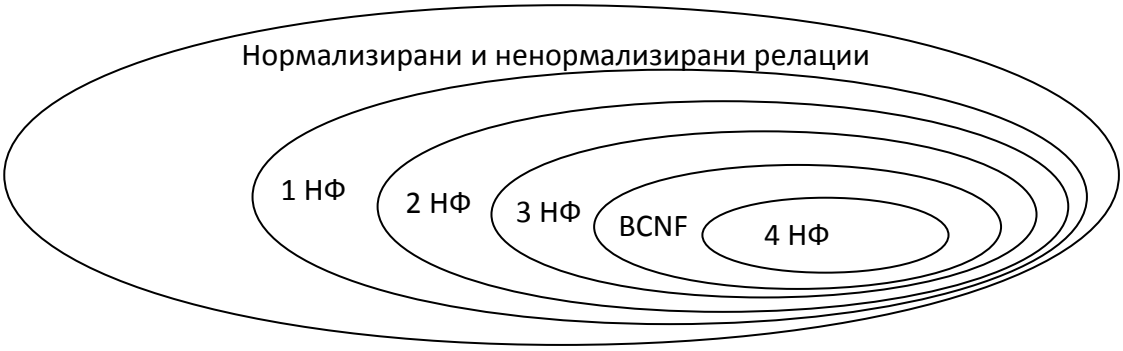
Релацията R е в **<u>четвърта нормална форма</u>**, ако за всяка нетривиална МЗ $A_1A_2…A_n \rightarrow\rightarrow B_1B_2…B_m$ е изпълнено, че $A_1A_2…A_n$ е суперключ.

<u>Пример</u>: Релацията, която разглеждахме досега, нарушава условието за 4НФ. Например, name →→ street city е нетривиална МЗ, въпреки това name не е суперключ. Всъщност единственият ключ за релацията са всички атрибути.

<u>Декомпозиция към 4NF</u>: Ако X →→ Y нарушава 4NF, правим следната декомпозиция: XY е едната от декомпозираните релации, а всички атрибути без $X \cup Y$ е другата.

<u>Пример</u>: Нека да продължим със същия пример. Видяхме, че name →→ street city нарушава 4НФ. Правилото за декомпозиция ни кара да заменим схемата с пет атрибута със схема, която има само три атрибута от горната МЗ, и друга схема, състояща се от лявата страна (name) + атрибутите, които не се появяват в тази МЗ. Тези атрибути са title и year. Така след декомпозирането получаваме следните две схеми: R (name, street, city) и S(name, title, year). Във всяка от схемите няма нетривиални многозначни (или функционални) зависимости, така че те са в 4НФ.

Обобщение на нормалните форми.



Свойство	3НФ	BCNF	4НФ
Отсъствие на FD излишество	В повечето случаи	Да	Да
Отсъствие на MVD излишество	Не	Не	Да
Запазване на FD	Да	Не винаги	Не винаги
Запазване на MVD	Не винаги	Не винаги	Не винаги

BCNF (следователно и 4НФ) елиминират излишеството и други аномалии, които са причинени от ФЗ, докато само 4НФ елиминира допълнителния излишък, причинен от наличието на нетривиални МЗ, които не са ФЗ. Често 3НФ е достатъчна за премахването на този излишък, но има примери, където не е. BCNF не гарантира запазването на ФЗ и никоя от нормалните форми не гарантира запазване на МЗ, въпреки че в типични случаи зависимостите се запазват.

10. Процедурно програмиране – основни информационни и алгоритмични структури (C++).

Основните принципи на структурното програмиране са принципа за **модулност** и принципа за **абстракция на данните**.Съгласно принципа за модулност, програмата се разделя наподходящи взаимосвързани части, всяка от които се реализирачрез определени средства. Целта е промените в представянето наданните да не променят голям брой от модулите на програмата. Съгласно принципа за абстракция на данните, методите за използване на данните се отделят от методите за тяхното конкретно представяне. Програмите се конструират така, че да работят с абстрактни данни – данни с неуточнено представяне. След това представянето се конкретизира с помощта на множество функции, наречени конструктори, мутатори и функции за достъп (селектори), които реализират абстрактните данни по конкретен начин. Така при решаването на даден проблем се оформят следните нива на абстракция:
1. Приложения в проблемната област.
2. Модули, реализиращи основните операции над данните.
3. Примитивни операции – конструктори, мутатори, селектори.
4. Избор на представянето на данните. Реализацията на подхода трябва да е такава, че всяко ниво използва единствено средствата на непосредствено следващото го ниво. По този начин, промените, които възникват на едно ниво ще се отразят само на предходното ниво. При изпълнение на компютърна програма се извършват определени действия над данните, дефинирани в програмата. Тези данни се съхраняват в **информационните структури**, допустими в съответния език за програмиране. Най-общо типовете информационни структури могат да бъдат разделени на два вида: вградени и абстрактни. Вградените типове са предварително дефинирани и се поддържат от самия език, а абстрактните типове се дефинират от програмиста. Друга класификация на типовете е следната: скаларни и съставни типове.

1.) Скаларните типове представят данни, които се състоят само от една компонента. При съставните типове данни, данните представляват редица от компоненти. Скаларните типове, поддържани в C++ са следните: булев, цял, реален, символен, изброен, указател, псевдоним. Съставните типове, поддържани в C++ са следните: масив, вектор, запис.

Логически тип

Нарича се още булев тип. Типът е стандартен, вграден в реализацията на C++. За означаването му се използва запазената дума bool (съкращение от boolean).

Множество от стойности

Състои се от два елемента – стойностите true (истина) и false (лъжа). Тези стойности се наричат още булеви константи.

<булева_константа> ::= true | false.

Променлива величина, множеството от допустимите стойности, на която съвпада с множеството от стойности на типа булев, се нарича булева или логическа променлива или променлива от тип булев. Дефинира се по обичайния начин.

Примери:
bool b1, b2;
bool b3 = false;

Дефиницията свързва булевите променливи с множеството от стойности на типа булев или с конкретна стойност от това множество като отделя по 1 байт оперативна памет за всяка от тях. Стойността на тази памет е неопределена или е булевата константа, свързана с дефинираната прменлива, в случай, че тя е инициализирана.

Оператори и вградени функции

Логически оператори

Оператор за логическо умножение (конюнкция)
Той е двуаргументен (бинарен) оператор. Означава се с and или && (за Visual C++)
Операторът се поставя между двата си аргумента. Такива оператори се наричат инфиксни.

Оператор за логическо събиране (дизюнкция)
Той също е бинарен, инфиксен оператор. Означава се с or или || (за Visual C++ 6.0)
Оператор за логическо отрицание

Той е едноаргументен (унарен) оператор. Означава се с not или ! (за Visual C++ 6.0)
Поставя се пред единствения си аргумент. Такива оператори се наричат префиксни.
Смисълът на операторите and, or и not е разширен чрез разширяване смисъла на булевите константи. Прието е, че true е всяка стойност, различна от 0 и че false е стойността 0.

Оператори за сравнение

Над булевите данни могат да се извършват следните инфиксни оператори за сравнение:
==, !=, >, >=, <, <=
Сравняват се кодовете.
Примери:
false < true e true
false > false e false
true >= false e true

Въвеждане

Не е възможно въвеждане на стойност на булева променлива чрез оператора >>, т.е. операторът
cin >> b1;
където b1 е булевата променлива, дефинирана по-горе, е недопустим.

Извеждане

Осъществява се чрез оператора
cout << <булева_константа>;
или по-общо
cout << <булев_израз>;
където синтактичната категория <булев_израз> е определена по-долу.
Извежда се кодът на булевата константа или кодът на булевата константа, която е стойност на <булев_израз>.

Булеви изрази

Булевите изрази са правила за получаване на булева стойност. Дефинират се рекурсивно по следния начин:

Булевите константи са булеви изрази.
Булевите променливи са булеви изрази.
Прилагането на булевите оператори not (!), and (&&), or (||) над булеви изрази е булев израз.
Прилагането на операторите за сравнение ==, !=, >, >=, <, <= към булеви изрази е булев израз.

Числени типове

Целочислени типове

Ще разгледаме целочисления тип int.

Типът е стандартен, вграден в реализацията на езика. За означаването му се използва запазената дума int (съкращение от integer).

Множество от стойности

Множеството от стойности на типа int зависи от хардуера и реализацията и не се дефинира от ANSI (American National Standarts Institute). Състои се от целите числа от някакъв интервал. За реализацията Visual C++ 6.0, това е интервалът [-231, 231-1].
Целите числа се записват като в математиката, т.е.
<цяло_число> ::= [+|-]<цяло_без_знак>
<цяло_без_знак> ::= <цифра>|<цифра><цяло_без_знак>
<цифра> ::= 0|1| … |9.
Обикновено знакът + се пропуска.
Допълнение: Целите числа могат да са в десетична, осмична и шестнадесетична позиционна система. Осмичните числа започват с 0 (нула), а шестнадесетичните - с 0x (нула, x).

Елементите от множеството от стойности на типа int се наричат константи от тип int. Променлива величина, множеството от допустимите стойности, на която съвпада с множеството от стойности на типа int, се нарича цяла променлива или променлива от тип int. Дефинира се по обичайния начин. Дефиницията свързва променливата с множеството от стойности на типа int или с конкретна стойност от това множество като отделя по 4 байта (1 дума) оперативна памет за всяка от тях. Ако променливата не е била инициализирана, стойността на свързната с нея памет е неопределена, а в противен случай е указаната при инициализацията стойност.

Аритметични оператори

Унарни оператори

Записват се пред или след единствения си аргумент.

+, - са префиксни оператори. Потвърждават или променят знака на аргумента си.

Следните означения съдържат унарния оператор + или -:

-i +j -j +i -567

Бинарни оператори

Имат два аргумента. Следните аритметични оператори са инфиксни:

Оператор	Операция
+	събиране
-	изваждане
*	умножение
/	целочислено деление
%	остатък от целочислено деление

Логически оператори

Логическите оператори, реализиращи конюнкция, дизюнкция и отрицание, могат да се прилагат над целочислени константи. Дефинират се по същия начин, както при булевите константи, но целите числа, които са различни от 0 се интерпретират true, а 0 – като false.
Примери:
123 and 0 e false
0 or 15 e true
not 67 e false

Оператори за сравнение

Над цели константи могат да се прилагат следните инфиксни оператори за сравнение:

Оператор	Операция
==	сравнение за равно
!=	сравнение за различно
>	сравнение за по-голямо
>=	сравнение за по-голямо или равно
<	сравнение за по-малко
<=	сравнение за по-малко или равно

Наредбата на целите числа е като в математиката.

Вградени функции

В езика C++ има набор от вградени функции. Обръщението към такива функции има следния синтаксис:

<име_на_функция>(<израз>, <израз>, …, <израз>)

и връща стойност от типа на функцията.

Тук ще разгледаме само едноаргументната целочислена функция abs.

abs(x) – намира |x|, където x е цял израз

Примери:

abs(-1587) = 1587 abs(0) = 0 abs(23) = 23

Въвеждане

Реализира се по стандартния и разгледан вече начин.
Пример: Ако
int i, j;
операторът
cin >> i >> j;
въвежда стойности на целите променливи i и j.

Извеждане

Реализира се чрез оператора
cout << <цяла_константа>;

Други целочислени типове
Други цели типове се получават от int като се използват модификаторите short, long, signed и unsigned. Тези модификатори доопределят някои аспекти на типа int.
За реализацията Visual C++ 6.0 са в сила:

Тип	Диапазон	Необходима памет
short int	-32768 до 32767	2 байта
unsigned shortint	0 до 65535	2 байта
long int	-2147483648 до 2147483647	4 байта
unsigned long int	0 до 4294967295	4 байта
unsigned int	0 до 4294967295	4 байта

Запазената дума int при тези типове се подразбира и може да бъде пропусната. Типовете short int (или само short) и long int (или само long) са съкратен запис на signed short int и signed long int.

Реални типове

Ще разгледаме реалния тип double.
Типът е стандартен, вграден във всички реализации на езика.

Множество от стойности

Множеството от стойности на типа double се състои от реалните числа от -1.74*10308 до 1.7*10308 . Записват се във два формата – като числа с фиксирана и като числа с плаваща запетая (експоненциален формат).

<реално_число> ::= <цяло_число>.<цяло_число_без_знак>|
<цяло_число>E<порядък>|

<цяло_число>.<цяло_число_без_знак>E<порядък>|
<порядък> ::= <цяло_число>
При експоненциалния формат може да се използва и малката латинска буква e.
Примери: Следните числа
123.3454 -10343.034 123E13 -1.23e-4
са коректно записани реални числа. Смисълът на експоненциалния формат е реално число, получено след умножаване на числото пред E (e) с 10 на степен числото след E (e).
Примери: 12.5E4 е реалното число 125000.0, а –1234.025e-3 е реалното число –1.234025.
Дефинира се по обичайния начин. Заделят се по 8 байта оперативна памет за всяка променлива от тип double
Аритметични оператори

Унарни оператори

+, - Префиксни са. Потвърждават или променят знака на аргумента си.

Бинарни оператори

Имат два аргумента. Следните аритметичните оператори са инфиксни:

Оператор	Операция
+	събиране
-	изваждане
*	умножение
/	деление (поне единият аргумент е реален)

Логически оператори

Логическите оператори, реализиращи операциите конюнкция, дизюнкция и отрицание, могат да се прилагат над реални константи. Дефинират се по същия начин, както при булевите константи, като реалните числа, които са различни от 0.0 се интерпретират като true, а 0.0 – като false.

Оператори за сравнение

Над реални данни могат да се прилагат стандартните инфиксни оператори за сравнение (==, !=, >, >=, <, <=).

Допълнение: Сравнението за равно на две реални числа x и y се реализира обикновено чрез релацията: $|x - y| < \varepsilon$, където $\varepsilon = 10\text{-}14$ за тип double. По-добър начин е да се използва релацията:

Вградени функции

$$\frac{|x - y|}{\max\{|x|, |y|\}} \leq \varepsilon$$

При цял или реален аргумент, следните функции връщат реален резултат от тип double:

Функция	Намира
sin(x)	Синус, sin x, x е в радиани
Cos(x)	косинус, cos x, x е в радиани
tan(x)	тангенс, tg x, x е в радиани
asin(x)	аркуссинус, arcsin $x \in [-\pi/2, \pi/2]$, $x \in [-1, 1]$
acos(x)	аркускосинус, arccos $x \in [0, \pi]$, $x \in [-1, 1]$
atan(x)	аркустангенс, arctg $x \in (-\pi/2, \pi/2)$
exp(x)	експонента, ex
log(x)	натурален логаритъм, ln x, x > 0
log10(x)	десетичен логаритъм, lg x, x > 0
sinh(x)	хиперболичен синус, sh x
cosh(x)	хиперболичен косинус, ch x
tanh(x)	хиперболичен тангенс, th x
ceil(x)	най-малкото цяло ≥ x, преобразувано в тип double

Floor(x)	най-голямото цяло ≤ x, преобразувано в тип double
fabs(x)	абсолютна стойност на x, |x|
sqrt(x)	Корен квадратен от x, x ≥ 0
pow(x, n)	степенуване, xn (x и n са реални от тип double).

Въвеждане и Извеждане се Реализира по стандартния начин операторите cin и cout.
Допълнение: за допълнително форматиране на изхода е възможно да се използват манипулаторите: setw (Задава широчината на следващото изходно поле.) и setprecision () задава броя на цифрите, използвани при извеждане на следващите реални числа, а в съчетание с setiosflags(ios::fixed), задава броя на цифрите след десетичната точка на извежданите реални числа.
int x = 21.5632
Операторът

```
cout << setiosflags(ios::fixed)
 << setprecision(3)
 << setw(10) << x << "\n";
```

извежда
****21.563
Други реални типове

В езика C++ има и друг реален тип, наречен float. Различава се от типа double по множеството от стойностите си и заеманата памет.
Множеството от стойности на типа float се състои от реалните числа от диапазона от – 3.4*1038 до 3.4*1038. За записване на константите от този диапазон са необходими 4 байта ОП.

Структура от данни масив

Под структура от данни се разбира организирана информация, която може да бъде описана, създадена и обработена с помощта на програма.

За да се определи една структура от данни е необходимо да се направи:
- логическо описание на структурата, което я описва на базата на декомпозицията ѝ на по-прости структури, а също на декомпозиция на операциите над структурата на по-прости операции.
- физическо представяне на структурата, което дава методи за представяне на структурата в паметта на компютъра.

Елементите на структури, които се състоят от една компонента, се наричат прости, или скаларни. Структури от данни, компонентите на които са редици от елементи, се наричат съставни.
Структури от данни, за които операциите включване и изключване на елемент не са допустими, се наричат статични, в противен случай - динамични.
Логическо описание

Масивът е крайна редица от фиксиран брой елементи от един и същ тип. Към всеки елемент от редицата е възможен пряк достъп, който се осъществява чрез индекс. Операциите включване и изключване на елемент в/от масива са недопустими, т.е. масивът е статична структура от данни.

Физическо представяне

Елементите на масива се записват последователно в паметта на компютъра, като за всеки елемент на редицата се отделя определено количество памет.

В езика C++ структурата масив се реализира чрез типа масив.
Тип масив

В C++ структурата от данни масив е реализирана малко ограничено. Разглежда се като крайна редица от елементи от един и същ тип с пряк достъп до всеки елемент, осъществяващ се чрез индекс с цели стойности, започващи от 0 и нарастващи с 1 до указана горна граница. Дефинира се от програмиста.

Дефиниране на масив

Типът масив се определя чрез задаване на типа и броя на елементите на редицата, определяща масив. Нека T е име или дефиниция на произволен тип, различен от псевдоним, void и функционален. За типа T и константния израз от интегрален или изброен тип с положителна стойност size, T[size] е тип масив от size елемента от тип T. Елементите се индексират от 0 до size–1. T се нарича базов тип за типа масив, а size – горна граница.

Примери:
int[5] дефинира масив от 5 елемента от тип int, индексирани от 0 до 4;
double[10] дефинира масив от 10 елемента от тип double, индексирани от 0 до 9;
bool[4] дефинира масив от 4 елемента от тип bool, индексирани от 0 до 3.
Множество от стойности

Множеството от стойности на типа T[size] се състои от всички редици от по size елемента, които са произволни константи от тип T. Достъпът до елементите на редиците е пряк и се осъществява с помощта на индекс, като достъпът до първия елемент се осъществява с индекс със стойност 0, до последния – с индекс със стойност size-1, а до всеки от останалите елементи – с индекс със стойност с 1 по-голяма от тази на индекса на предишния елемент. Елементите от множеството от стойности на даден тип масив са константите на този тип масив.
Примери:
1. Следните редици {1,2,3,4,5}, {-3, 0, 1, 2, 0}, {12, -14, 8, 23, 1000} са константи от тип int[5].

2. Редиците {1.5, -2.3, 3.4, 4.9, 5.0, -11.6, -123.56, 13.7, -32.12, 0.98}, {-13, 0.5, 11.9, 21.98, 0.03, 1e2, -134.9, 0.09, 12.3, 15.6} са константи от тип double[10].
Дефиниция на масив

<дефиниция_на_променлива_от_тип_масив> ::=
T <променлива>[size] [= {<редица_от_константни_изрази>}]опц
{,<променлива>[size] [= {<редица_от_константни_изрази>}]опц }опц;

където

- T е име или дефиниция на произволен тип, различен от псевдоним, void, функционален;
- <променлива> е идентификатор;
- size е константен израз от интегрален или изброен тип с положителна стойност;
- <редица_от_константни_изрази> се дефинира по следния начин:

<редица_от_константни_изрази> ::= <константен_израз>|
<константен_израз>, <редица_от_константни_изрази>

като константните изрази са от тип T или от тип, съвместим с него.
Примери:

```
int a[5];
double c[10];
bool b[3];
enum {FALSE, TRUE} x[20];
double p[4] = {1.25, 2.5, 9.25, 4.12};
```

Дефиницията
T <променлива>[size] = {<редица_от_константни_изрази>}
се нарича дефиниция на масив с инициализация, а Фрагмента
{<редица_от_константни_изрази>} - инициализация. При нея е възможно size да се пропусне. Тогава за стойност на size се подразбира броят на константните изрази, изброени в инициализацията. Ако size е указано и изброените константни изрази в инициализацията са по-малко от size, останалите се приемат за 0.

Примери:

Дефиницията
int q[5] = {1, 2, 3};
е еквивалентна на
int q[] = {1, 2, 3, 0, 0};
Дефиницията
double r[] = {0, 1, 2, 3};
е еквивалентна на

```
double r[4] = {0, 1, 2, 3};
```

Дефиницията с инициализация е един начин за свързване на променлива от тип масив с конкретна константа от множеството от стойности на този тип масив. Друг начин предоставят т.нар. индексирани променливи. С всяка променлива от тип масив е свързан набор от индексирани променливи. Фиг. 6.2 илюстрира техния синтаксис.

Синтаксис на индексирани променлии
<индексирана_променлива> ::=

```
<променлива_от_тип_масив>[<индекс>]
```

където
<индекс> е израз от интегрален или изброен тип.
Всяка индексирана променлива е от базовия тип.

Синтаксис на индексираните променливи

Примери:
1. С променливата a, дефинирана по-горе, са свързани индексираните променливи a[0], a[1], a[2], a[3] и a[4], които са от тип int.
2. С променливата b са свързани индексираните променливи b[0], b[1],…, b[9], които са от тип double.
3. С променливата x са свързани индексираните променливи x[0], x[1],…, x[19], които са от тип enum {FALSE, TRUE}.

Дефиницията на променлива от тип масив не само свързва променливата с множеството от стойности на указания тип, но и отделя определено количество памет (обикновено 4B), в която записва адреса в паметта на първата индексирана променлива на масива. Останалите индексирани променливи се разполагат последователно след

първата. За всяка индексирана променлива се отделя по толкова памет, колкото базовият тип изисква.
Операции и вградени функции

Не са възможни операции над масиви като цяло, но всички операции и вградени функции, които базовият тип допуска, са възможни за индексираните променливи, свързани с масива.

Пример: Нека

int a[5], b[5];

Недопустими са:

cin >> a >> b;

a = b;

а също a == b или a != b.

Операторът

cout << a;

е допустим и извежда адреса на a[0].
Някои приложения на структурата от данни масив са: Търсене на елемент в редица, Сортиране на редица, Сливане на редици (за повече инфо за писмения изпит - вж. учебника)
Друго приложение на типът масив е за представяне на символни низове в C++.
Разглежданите досега масиви се наричат едномерни. Те реализират крайни редици от елементи от скаларен тип. Възможно е обаче типът на елементите да е масив. В този случай се говори за многомерни масиви.

Многомерни масиви

Масив, базовият тип на който е едномерен масив, се нарича двумерен. Масив, базовият тип на който е двумерен масив, се нарича тримерен и т.н. На практика се използват масиви с размерност най-много 3.

Дефиниране на многомерни масиви

Нека T е име или дефиниция на произволен тип, различен от псевдоним, void и функционален, size1, size2, …, sizen (n>1 е дадено цяло число) са константни изрази от интегрален или изброен тип с положителни стойности. T[size1][size2] … [sizen] е тип n-мерен масив от тип T. T се нарича базов тип за типа масив.

Примери:

int [5][3] дефинира двумерен масив от тип int;

double [4][5][3] дефинира тримерен масив от тип double;

Множество от стойности

Множеството от стойности на типа T[size1][size2] … [sizen] се състои от всички редици от по size1 елемента, които са произволни константи от тип T[size2] … [sizen].
Достъпът до елементите на редиците е пряк и се осъществява с помощта на индекс, като достъпът до първия елемент се осъществява с индекс със стойност 0, до последния – с индекс със стойност size1-1, а до всеки от останалите елементи – с индекс със стойност с 1 по-голяма от тази на индекса на предишния елемент. Елементите от множеството от стойности на даден тип многомерен масив са константите на този тип масив.
Примери:

Множеството от стойности на типа int[5][3] се състои от всички редици от по 5 елемента, които са едномерни масиви от тип int[3]. Достъпът до елементите на редиците се осъществява с индекс със стойности 0, 1, 2, 3 и 4.

Синтаксис на многомерен масив

Примери:

```
int x[10][20];
double y[20][10][5];
int z[3][2] = {{1, 3},
{5, 7},
{2, 9}};
int t[2][3][2] = {{{1, 3}, {5, 7}, {6, 9}},
        {{7, 8}, {1, 8}, {-1, -4}}};
```

Дефиницията на променлива от тип многомерен масив не само свързва променливата с множеството от стойности на указания тип, но и отделя определено количество памет за разполагане на индексираните променливи, свързани с нея. Индексираните променливи се разполагат последователно по по-бързото нарастване на по-далечните си индекси. За всяка индексирана променлива се отделя толкова памет, колкото базовият тип изисква.

Забележка: Двумерните масиви разполагат в ОП индексираните си променливи по по-бързото нарастване на втория индекс. Това физическо представяне се нарича представяне по редове. Тези масиви могат да бъдат използвани за реализация и работа с матрици и др. правоъгълни таблици.

Важно допълнение: При работа с масиви трябва да се има предвид, че повечето реализации не проверяват дали стойностите на индексите са в рамките на границите, зададени при техните дефиниции. Тази особеност крие опасност от допускане на трудноотриваеми грешки.

Тип указател

Променлива, това е място за съхранение на данни, което може да съдържа различни стойности. Идентифицира се с дадено от потребителя име (идентификатор). Има си и тип. Дефинира се като се указват задължително типът и името й. Типът определя броя на байтовете, в които ще се съхранява променливата, а също и множеството от операциите, които могат да се изпълняват над нея. Освен това, с променливата е свързана и стойност – неопределена или константа от типа, от който е тя. Нарича се още rvalue. Мястото в паметта, в което е записана rvalue, се идентифицира с адрес, който се нарича адрес на променливата или lvalue. По-точно адресът е адреса на първия байт от множеството байтове, отделени за променливата.

Пример: Фрагментът

```
int i = 1024;
```

дефинира променлива с име i и тип int. Стойността й (rvalue) е 1024. i именува място от паметта (lvalue) с размери 4 байта, като lvalue е адреса на първия байт на това място.

Намирането на адреса на дефинирана променлива става чрез унарния префиксен дясно-асоциативен оператор & (амперсанд). Приоритетът му е същия като на унарните оператори +, -, !, ++, -- и др.

Оператор &

Синтаксис

```
&<променлива>
```

където <променлива> е вече дефинирана променлива.

Семантика

Намира адреса на <променлива>.

Пример: &i е адреса на променливата i и може да се изведе чрез оператора cout << &i;

Операторът & не може да се прилага върху константи и изрази, т.е. &100 и &(i+5) са недопустими обръщения. Не е възможно също прилагането му и върху променливи от тип масив, тъй като те имат и смисъла на константни указатели.

Адресите могат да се присвояват на специален тип променливи, наречени променливи от тип указател или само указатели.

Дефиниране

Нека T е име или дефиниция на тип. За типа T, T* е тип, наречен указател към T. T се нарича указван тип или тип на указателя.

Примери:

int* е тип указател към тип int;

enum {a, b, c}* е тип указател към тип enum {a, b, c}.

Множество от стойности

Състои се от адресите на данните от тип T, дефинирани в програмата, преди използването на T*. Те са константите на типа T*. Освен тях съществува специална константа с име NULL, наречена нулев указател. Тя може да бъде свързвана с всеки указател независимо от неговия тип. Тази константа се интерпретира като "сочи към никъде", а в позиция на предикат е false.

Променлива величина, множеството от допустимите стойности, на която съвпада с множеството от стойности на типа T*, се нарича променлива от тип T*, променлива от тип указател към тип T или само указател към тип T. Дефинира се по общоприетия начин.

Дефиниция на променлива от тип указател

T* <променлива> [= <стойност>]опц

{,<променлива> [= <стойност>]опц}опц|

T *<променлива> [= <стойност>]опц

{*<променлива> [= <стойност>]опц}опц;

където

- T е име или дефиниция на тип;
- <променлива> е идентификатор;
- <стойност> е шестнадесетично число, представляващо адрес на данна от тип T или NULL.

В първия случай на дефиницията обаче има особеност – операторът * се свързва с T само за първата променлива. Дефиницията

T* a, b;

е еквивалентна на

T* a;

T b;

т.е. само променливата a е от тип указател към тип T.

Примери: Дефиницията

int *pint1, *pint2;

задава два указателя към тип int, а

```
int *pint1, pint2;
```

- указател pint1 към int и променлива pint2 от тип int.

Дефиницията на променлива от тип указател предизвиква в ОП да се отделят 4B, в които се записва някакъв адрес от множеството от стойности на съответния тип, ако дефиницията е с инициализация и неопределено или NULL, ако дефиницията не е с инициализация. Този адрес е стойността на променливата от тип указател, а записаното на този адрес е съдържанието й.

Пример: В резултат от изпълнението на дефинициите

```
int i = 12;
int* p = &i;              // p е инициализирано с адреса на i
double *q = NULL; // q е инициализирано с нулевия указтел
double x = 1.56;
double *r = &x;           // r е инициализирано с адреса на x
```

Операции и вградени функции

Извличане на съдържанието на указател

Осъществява се чрез префиксния, дясноасоциативен унарен оператор *. * има приоритет на унарен оператор.

Оператор *

Синтаксис

*<променлива_от_тип_указател>

Семантика

Извлича стойността на адреса, записан в <променлива_от_тип_ указател>, т.е. съдържанието на <променлива_от_тип_ указател>.

Като използваме дефинициите от примера по-горе, имаме:

*p е 12// 12 е съдържанието на p

```
*r е 1.56        // 1.56 е съдържанието на r
```

Освен, че намира съдържанието на променлива от тип указател, обръщението

*<променлива_от_тип_указател>

е променлива от тип T. (Всички операции, допустими за типа T, са допустими и за нея.

Като използваме дефинициите от примера по-горе, *p и *r са цяла и реална променливи, съответно. След изпълнение на операторите за присвояване

```
*p = 20;
*r = 2.18;
```

стойността на i се променя на 20, а тази на r – на 2.18.

Аритметични и логически операции

Променливите от тип указател могат да участват като операнди в следните аритметични и логически операции +, -, ++, --, ==, !=, >, >=, < и <=. Изпълнението на аритметични операции върху указатели е свързано с някои особености, заради което аритметиката с указатели се нарича още адресна аритметика. Особеността се изразява в т. нар. мащабиране. Ще го изясним чрез пример.

Да разгледаме фрагмента

```
int *p;
double *q;
...
p = p + 1;
q = q + 1;
```

Операторът p = p + 1; свързва p не със стойността на p, увеличена с 1, а с p + 1*4, където 4 е броя на байтовете, необходими за записване на данна от тип int (p е указател към int). Аналогично, q = q + 1; увеличава стойността на q не с 1, а с 8, тъй като q е указател към double (8 байта са необходими за записване на данна от този тип).
Общото правило е следното: Ако p е указател от тип T*, p+i е съкратен запис на p + i*sizeof(T), където sizeof(T) е функция, която намира броя на байтовете, необходими за записване на данна от тип T.

Въвеждане

Не е възможно въвеждане на данни от тип указател чрез оператора cin. Свързването на указател със стойност става чрез инициализация или оператора за присвояване.

Извеждане

Осъществява се по стандартния начин - чрез оператора cout.

Допълнение

Типът, който се задава в дефиницията на променлива от тип указател, е информация за компилатора относно начина, по който да се интерпретира съдържанието на указателя. В контекста на последния пример *p са четири байта, които ще се интерпретират като цяло число от тип int. Аналогично, *q са осем байта, които ще се интерпретират като реално число от тип double.
В някои случаи е важна стойността на променливата от тип указател (адресът), а не нейното съдържание. Тогава тя се дефинира като указател към тип void. Този тип указатели са предвидени с цел една и съща променлива - указател да може в различни моменти да сочи към данни от различен тип. В този случай, при опит да се използва съдържанието на променливата от тип указател, ще се предизвика грешка.
Съдържанието на променлива - указател към тип void може да се извлече само след привеждане на типа на указателя (void*) до типа на съдържанието. Това може да се осъществи чрез операторите за преобразуване на типове.
Пример:

```
int a = 100;
void* p;                                    // дефинира указател към void
p = &a;                                     // инициализира p
cout << *p;                                 // грешка
cout << *((int*) p);            // преобразува p в указател към int
// и тогава извлича съдържанието му.
```

В C++ е възможно да се дефинират указатели, които са константи (T* const <идентификатор>), а също и указатели, които сочат към константи (const T* <идентификатор>). И в двата случая се използва запазената дума const, която се поставя пред съответните елементи от дефинициите на указателите. Стойността на елемента, дефиниран като const (указателя или обекта, към който сочи) не може да бъде променяна. В дефиницията:

T* const <идентификатор>;

<идентификатор> е константен указател към тип T и не може да бъде променяна стойността му. В дефиницията:

const T* <идентификатор>;

<идентификатор> е указател към константа от тип T и не може да бъде променяно съдържанието му.
Пример:

```
int i, j = 5;
int *pi;                                    // pi е указател към int
```

```
int * const b = &i;      // b е константен указател към int
const int *c = &j;       // c е указател към цяла константа.
b = &j;                                    // грешка, b е константен указател
*c = 15;                                   // грешка, *c е константа
```

Указатели и масиви

В C++ има интересна и полезна връзка между указателите и масивите. Изразява се в това, че имената на масивите са указатели към техните "първи" елементи. Последното позволява указателите да се разглеждат като алтернативен начин за обхождане на елементите на даден масив.

Указатели и едномерни масиви

Нека a е масив, дефиниран по следния начин:
int a[100];
Тъй като a е указател към a[0], *a е стойността на a[0], т.е. *a и a[0] са два различни записа на стойността на първия елемент на масива. Тъй като елементите на масива са разположени последователно в паметта, a + 1 е адреса на a[1], a + 2 е адреса на a[2] и т.н. a + n-1 е адреса на a[n-1]. Тогава *(a+i) е друг запис на a[i] (i = 0, 1, ..., n-1).
Има обаче една особеност. Имената на масивите са константни указатели. Заради това, някои от аритметичните оператори, приложими над указатели, не могат да се приложат над масиви. Такива са ++, -- и присвояването на стойност. Използването на указатели е по-бърз начин за достъп до елементите на масива и заради това се предпочита. Индексираните променливи правят кода по-ясен и разбираем. В процеса на компилация всички конструкции от вида a[i] се преобразуват в *(a+i), т.е. операторът за индексиране [] се обработва от компилатора чрез адресна аритметика. Полезно е да отбележим, че операторът [] е ляво-асоциативен и с по-висок приоритет от унарните оператори (в частност от оператора за извличане на съдържание *).

Указатели и двумерни масиви

Името на двумерен масив е константен указател към първия елемент на едномерен масив от константни указатели. Ще изясним с пример казаното.
Нека a е двумерен масив, дефиниран по следния начин: int a[10][20];
Променливата a е константен указател към първия елемент на едномерния масив a[0], a[1], ..., a[9], като всяко a[i] е константен указател към a[i][0] (i = 0,1, ..., 9), т.е.
a

a[0] a[0][0] a[0][1] … a[0][19]

a[1] a[1][0] a[1][1] … a[1][19]

…

…

a[9] a[9][0] a[9][1] … a[9][19]

Тогава

**a == a[0][0]
a[0] == *a a[1] == *(a + 1) ... a[9] == *(a + 9),
т.е.
a[i] == *(a + i)
Като използваме, че операторът за индексиране е лявоасоциативен и с по-висок приоритет от оператора *, получаваме:
a[i][j] == (*(a+i))[j] == *(*(a+i)+j).
(вж. задачи за писмения в учебника)

Указатели и низове

Низовете са масиви от символи. Името на променлива от тип низ е константен указател, както и името на всеки друг масив. Така всичко, което казахме за връзката между масив и указател е в сила и за низ – указател.

Следващият пример илюстрира обхождане на низ чрез използване на указател към char. Обхождането продължава до достигане на знака за край на низ.

```
#include <iostream.h>
int main()
{char str[] = "C++Language"; // str е константен указател
 char* pstr = str;                  // pstr е указател към низа str
 while (*pstr)
 {cout << *pstr << '\n';
  pstr++;
 }                  // pstr вече не е свързан с низа “C++Language”.
 return 0;
}
```

Тъй като низът е зададен чрез масива от символи str, str е константен указател и не може да бъде променяна стойността му. Затова се налага използването на помощната променлива pstr. Ако низът е зададен чрез указател към char, както е в следващата програма, не се налага използването на такава.

```
#include <iostream.h>
int main()
{char* str = "C++Language";  // str е променлива
 while (*str)
 {cout << *str << '\n';
  str++;
 }
 return 0;
}
```

Примерите показват, че задаването на низ като указател към char има предимство пред задаването като масив от символи. Ще отбележим обаче, че дефиницията
char* str = "C++Language";
не може да бъде заменена с
char* str;
cin >> str;
следвани с въвеждане на низа “C++Language”(ще напомним, че не е възможно въвеждане на стойност на променлива от тип указател чрез оператора cin), докато дефиницията
char str[20];
позволява въвеждането му чрез cin, т.е.

cin >> str;
е допустимо.

Функции

Добавянето на нови оператори и функции в приложенията, реализирани на езика C++, се осъществява чрез функциите. Те са основни структурни единици, от които се изграждат програмите на езика. Всяка функция се състои от множество от оператори, оформени подходящо за да се използват като обобщено действие или операция. След като една функция бъде дефинирана, тя може да бъде изпълнявана многократно за различни входни данни.

Програмите на езика C++ се състоят от една или повече функции. Сред тях задължително трябва да има точно една с име main и наречена главна функция. Тя е първата функция, която се изпълнява при стартиране на програмата. Главната функция от своя страна може да се обръща към други функции. Нормалното изпълнение на програмата завършва с изпълнението на главната функция (Възможно е изпълнението да завърши принудително с изпълнението на функция, различна от главната).

Използването на функции има следните предимства:
- Програмите стават ясни и лесни за тестване и модифициране.
- Избягва се многократното повтаряне на едни и същи програмни фрагменти. Те се дефинират еднократно като функции, след което могат да бъдат изпълнявани произволен брой пъти.
- Постига се икономия на памет, тъй като кодът на функцията се съхранява само на едно място в паметта, независимо от броя на нейните изпълнения.

8.3 Дефиниране на функции

Синтаксис

Дефиницията на функция се състои от две части: заглавие (прототип) и тяло. Синтаксисът й е показан на Фиг. 8.4.

Дефиниране на функция

[<модификатор>]опц[<тип_на_функция>] опц <име_на_функция>
(<формални_параметри>)
{<тяло>
}
където
<модификатор>::= inline|static| ...
<тип_на_функцията> ::= <име_на_тип> | <дефиниция_на_тип>
<име_на_функция> ::= <идентификатор>
<формални_параметри> :: <празно> | void |
<параметър> {,
<параметър>}
<параметър> ::= <тип>[& |opc * [const]opc] opc <име_на_параметър>
<тип> ::= <име_на_тип>
<име_на_параметър> ::= <идентификатор>
<тяло> ::= <редица_от_оператори_и_дефиниции>

Дефиниция на функция

Модификаторите са спецификатори, които задават препоръка за компилатора (inline), класа памет (extern или static) и др. характеристики. Ще дадем примери в следващите разглеждания. Ако <модификатор> е пропуснат, подразбира се extern.
Типът на функцията е произволен тип без масив и функционален, но се допуска да е указател към такива обекти (в широкия смисъл на думата). Ако е пропуснат, подразбира се int.
Името на функцията е произволен идентификатор. Допуска се неоднозначност.
Списъкът от формални параметри (нарича се още сигнатура) може да е празен или void.
Например, следната функция извежда текст:

```
void printtext(void)
{cout << "C++ Programming Language \n"
        cout << "B. Stroustrup \n";
        return;
}
```

В случай, че списъкът е непразен, имената на параметрите трябва да са различни. Те заедно с името определят еднозначно функцията. Формалните параметри са: параметри – стойности, параметри – указатели и параметри – псевдоними. Името на параметъра се предшества от тип.

Примери:

```
int a, int const& b, double& x, int const * y, const int* a
```

Засега няма да използваме параметри, специфицирани със const.
Тялото на функцията е редица от дефиниции и оператори. Тя описва алгоритъма, реализиращ функцията. Може да съдържа един или повече оператора return.
Операторът return (Фиг. 8.5) връща резултата на функцията в мястото на извикването.

Оператор return

Синтаксис

return [<израз>]опц

където

- return е запазена дума;
- <израз> е произволен израз от тип <тип_на_функцията> или съвместим с него. Ако типът на функцията е void, <израз> се пропуска. В този случай е възможно и return да се пропусне.

Семантика

Пресмята се стойността на <израз>, конвертира се до типа на функцията (ако е възможно) и връщайки получената стойност в мястото на извикването на функцията, прекратява изпълнението ѝ.

Оператор return

Забележка: Ако функцията не е от тип void, тя задължително трябва да върне стойност. Това означава, че операторът return трябва да се намира във всички разклонения на тялото. В противен случай, повечето компилатори ще изведат съобщение или предупреждение за грешка. Възможно е обаче функцията да върне случайна стойност, което е лошо. По-добре е функцията да върне някаква безобидна стойност, отколкото случайна.

(„**Декларация на функция" - невключено в анотацията на въпроса**)

```
/*********************************************************/
```

Функциите могат да се дефинират в произволно място на програмата, но не и в други функции. Преди да се извика една функция, тя трябва да е "позната" на компилатора. Това става, като дефиницията на функцията се постави пред main или когато функцията се дефинира на произволно място в частта за дефиниране на функции, а преди дефинициите на функциите се постави само нейната декларация (Фиг. 8.6).

Декларация на функция

<декларация_на_функция> ::=
[<модификатор>][<тип_на_резултата>]<име_на_функция>
([<формални_параметри>]);

Декларация на функция

Възможно е имената на параметрите във <формални_параметри> да се пропуснат.

Семантика

Описанието на функция задава параметрите, които носят входа и изхода, типа на резултата, а също и алгоритъма, за реализиране на действията, което функцията дефинира. Параметрите-стойности най-често задават входа на функцията. Параметрите-указатели и псевдоними са входно-изходните параметри за нея. Алгоритъмът се описва в тялото на функцията. Изпълнението на функцията завършва при достигане на края на тялото или след изпълнение на оператор return [<израз>]опц;.
/**/

Обръщение към функция

Синтаксис
<обръщение_към_функция> ::=
<име_на_функция>()|
<име_на_функция>(void)|
<име_на_функция>(<фактически_параметри>)
където <фактически_параметри> са толкова на брой, колкото са формалните параметри. Освен по брой, формалните и фактическите параметри трябва да си съответстват по тип, по вид и по смисъл.

Съответствието по тип означава, че типът на i-тия фактически параметър трябва да съвпада (да е съвместим) с типа на i-тия формален параметър. Съответствието по вид се състои в следното: ако формалният параметър е параметър-указател, съответният му фактически параметър задължително е променлива или адрес на променлива, ако е параметър-псевдоним, съответният му фактически параметър задължително е променлива (за реализацията Visual C++, 6.0 от същия тип) и ако е параметър-стойност – съответният му фактически параметър е израз.

Семантика
Обръщението към функция е унарна операция с най-висок приоритет и с операнд - името на функцията. Последното пък е указател със стойност адреса на мястото в паметта където е записан програмният код на функцията. Ако функцията определя процедура, обръщението към нея се оформя като оператор (завършва с ;). Опитът за използването й като израз предизвиква грешка. Ако функцията връща резултат както

чрез return, така и чрез някой от формалните си параметри, обръщението към нея може да се разглежда и като оператор, и като израз. И ако функцията връща резултат единствено чрез оператора return, обръщението към нея има единствено смисъла на израз. Използването му като оператор не води до грешка, но не предизвиква видим резултат.
Обръщението към функция предизвиква генериране на нова стекова рамка и се осъществява на следните два етапа:

Свързване на формалните с фактическите параметри

За целта първият формален параметър се свързва с първия фактически, вторият формален параметър се свързва с втория фактически и т.н. последният формален параметър се свързва с последния фактически параметър. Свързването се реализира по различни начини в зависимост от вида на формалния параметър.

а) формален параметър – стойност

В този случай се намира стойността на съответния му фактически параметър. В стековата рамка на функцията за формалния параметър се отделя толкова памет, колкото типът му изисква и в нея се откопирва стойността на фактическия параметър.

б) формален параметър – указател

В този случай в стековата рамка на функцията за формалния параметър се отделят 4B, в които се записва стойността на фактическия параметър, която е адрес на променлива. Действията, описани в тялото се изпълняват със съдържанието на формалния параметър - указател. По такъв начин е възможна промяна на стойността на променливата, чийто адреа е предаден като фактически параметър.

в) формален параметър – псевдоним

Формалният параметър-псевдоним се свързва с адреса на фактическия. За него в стековата рамка на функцията памет не се отделя. Той просто “прелита” и се “закачва” за фактическия си параметър. Действията с него се извършват над фактическия параметър.

Изпълнение на тялото на функцията

Аналогично е на изпълнението на блок.
При всяко обръщение към функция в програмния стек се включва нов “блок” от данни. В него се съхраняват формалните параметри на функцията, нейните локални променливи, а също и някои “вътрешни” данни като return-адреса и др. Този блок се нарича стекова рамка на функцията.
В дъното на стека е стековата рамка на main. На върха на стека е стековата рамка на функцията, която се обработва в момента. Под нея е стековата рамка на функцията, извикала функцията, обработваща се в момента. Ако изпълнението на една функция завършва, нейната стекова рамка се отстранява от стека.
Видът на стековата рамка зависи от реализацията. С точност до наредба, тя има вида:

```
Формални параметри
Адрес за връщане
Адрес на предходна рамка на стека
Локални параметри
```

Стекова рамка

Масивите като формални параметри

Едномерни масиви

Съществуват различни начини за задаване на формални параметри от тип едномерен масив.

а) традиционен

Дефиницията

T a[]

където T е скаларен тип, задава параметър a от тип едномерен масив с базов тип T. Може да се укаже горна граница на масива, но компилаторът я пренебрегва.

Примери:

int a[] - a е параметър от тип масив от цели числа,

int a[10] - еквивалентна е на int a[],

double b[] - b е параметър от тип масив от реални числа,

char c[] - c е параметър от тип масив от символи.

б) чрез указател

Дефиницията

T* p

където T е скаларен тип, задава параметър p от тип указател към тип T. От връзката между масив и указател следва, че тази дефиниция може да се използва и за дефиниране на формален параметър от тип масив.

Примери: Следните дефиниции на формални параметри са еквивалентни на тези от примера по-горе:

int* a - a е параметър от тип указател към int

double* b - b е параметър от тип указател към double.

char* c - c е параметър от тип указател към char.

И в двата случая фактическият параметър се указва с името на едномерен масив от същия тип. Необходимо е също на функцията да се подаде като параметър и размерът на масива.

Забележки:

Функциите работят с направо с масива a, а не с негови копия. Промените на елементите на масива се запазват след излизане от функцията.
Размерът на масивът не може да се разбере от неговото описание. Затова се налага използването на допълнителния параметър m в списъка от аргументи на функциите. Последното не се отнася за масивите, представляващи символни низове, тъй като те завършват със знака за край на низ '\0'.

Многомерни масиви

Когато многомерен масив трябва да е формален параметър на функция, в описанието му трябва да присъстват като константи всички размери с изключение на първият. Например, декларацията

```
void readarr2(int n, int matr[][20]);
```

определя matr като двумерен масив (редица от двадесеторки от цели числа).
Описанието

```
int (*matr)[20]
```

е еквивалентно на
int matr[][20]
Скобите, ограждащи *matr, са задължителни. В противен случай, тъй като [] е с по-висок приоритет от *, int *matr[20] ще се интерпретира като "matr е масив с 20 елемента от тип *int".
(вж. допълнителни задачи за писмения в учебника)

10. Процедурно програмиране – основни информационни и алгоритмични структури (C++).

Основните принципи на структурното програмиране са принципа за **модулност** и принципа за **абстракция на данните**.Съгласно принципа за модулност, програмата се разделя наподходящи взаимосвързани части, всяка от които се реализирачрез определени средства. Целта е промените в представянето наданните да не променят голям брой от модулите на програмата. Съгласно принципа за абстракция на данните, методите за използване на данните се отделят от методите за тяхното конкретно представяне. Програмите се конструират така, че да работят с абстрактни данни – данни с неуточнено представяне. След това представянето се конкретизира с помощта на множество функции, наречени конструктори, мутатори и функции за достъп (селектори), които реализират абстрактните данни по конкретен начин. Така при решаването на даден проблем се оформят следните нива на абстракция:
1. Приложения в проблемната област.
2. Модули, реализиращи основните операции над данните.
3. Примитивни операции – конструктори, мутатори, селектори.
4. Избор на представянето на данните. Реализацията на подхода трябва да е такава, че всяко ниво използва единствено средствата на непосредствено следващото го ниво. По този начин, промените, които възникват на едно ниво ще се отразят само на предходното ниво. При изпълнение на компютърна програма се извършват определени действия над данните, дефинирани в програмата. Тези данни се съхраняват в **информационните структури**, допустими в съответния език за програмиране. Най-общо типовете информационни структури могат да бъдат разделени на два вида: вградени и абстрактни. Вградените типове са предварително дефинирани и се поддържат от самия език, а абстрактните типове се дефинират от програмиста. Друга класификация на типовете е следната: скаларни и съставни типове.

1.) Скаларните типове представят данни, които се състоят само от една компонента. При съставните типове данни, данните представляват редица от компоненти. Скаларните типове, поддържани в C++ са следните: булев, цял, реален, символен, изброен, указател, псевдоним. Съставните типове, поддържани в C++ са следните: масив, вектор, запис.

Логически тип

Нарича се още булев тип. Типът е стандартен, вграден в реализацията на C++. За означаването му се използва запазената дума bool (съкращение от boolean).

Множество от стойности

Дефинира се по обичайния начин.

Примери:

```
bool b1, b2;
bool b3 = false;
```

Дефиницията свързва булевите променливи с множеството от стойности на типа булев или с конкретна стойност от това множество като отделя по 1 байт оперативна памет за всяка от тях. Стойността на тази памет е неопределена или е булевата константа, свързана с дефинираната прменлива, в случай, че тя е инициализирана.

Оператори и вградени функции

Логически оператори

Оператор за логическо умножение (конюнкция)
Той е двуаргументен (бинарен) оператор. Означава се с and или && (за Visual C++)
Операторът се поставя между двата си аргумента. Такива оператори се наричат инфиксни.

Оператор за логическо събиране (дизюнкция)
Той също е бинарен, инфиксен оператор. Означава се с or или || (за Visual C++ 6.0)
Оператор за логическо отрицание

Той е едноаргументен (унарен) оператор. Означава се с not или ! (за Visual C++ 6.0)
Поставя се пред единствения си аргумент. Такива оператори се наричат префиксни.
Смисълът на операторите and, or и not е разширен чрез разширяване смисъла на булевите константи. Прието е, че true е всяка стойност, различна от 0 и че false е стойността 0.

Оператори за сравнение

Над булевите данни могат да се извършват следните инфиксни оператори за сравнение:
==, !=, >, >=, <, <=
Сравняват се кодовете.
Примери:
false < true e true
false > false e false
true >= false e true

Въвеждане

Не е възможно въвеждане на стойност на булева променлива чрез оператора >>, т.е. операторът
cin >> b1;
където b1 е булевата променлива, дефинирана по-горе, е недопустим.

Извеждане

Осъществява се чрез оператора
cout << <булева_константа>;
или по-общо
cout << <булев_израз>;
където синтактичната категория <булев_израз> е определена по-долу.
Извежда се кодът на булевата константа или кодът на булевата константа, която е стойност на <булев_израз>.
Булеви изрази

Булевите изрази са правила за получаване на булева стойност. Дефинират се рекурсивно по следния начин:
Булевите константи са булеви изрази.
Булевите променливи са булеви изрази.
Прилагането на булевите оператори not (!), and (&&), or (||) над булеви изрази е булев израз.

Прилагането на операторите за сравнение ==, !=, >, >=, <, <= към булеви изрази е булев израз.

Числени типове

Целочислени типове

Ще разгледаме целочисления тип int.

Типът е стандартен, вграден в реализацията на езика. За означаването му се използва запазената дума int (съкращение от integer).

Допълнение: Целите числа могат да са в десетична, осмична и шестнадесетична позиционна система. Осмичните числа започват с 0 (нула), а шестнадесетичните - с 0x (нула, x).

Елементите от множеството от стойности на типа int се наричат константи от тип int. Променлива величина, множеството от допустимите стойности, на която съвпада с множеството от стойности на типа int, се нарича цяла променлива или променлива от тип int. Дефинира се по обичайния начин. Дефиницията свързва променливата с множеството от стойности на типа int или с конкретна стойност от това множество като отделя по 4 байта (1 дума) оперативна памет за всяка от тях. Ако променливата не е била инициализирана, стойността на свързната с нея памет е неопределена, а в противен случай е указаната при инициализацията стойност.

Аритметични оператори

Унарни оператори

Записват се пред или след единствения си аргумент.

+, - са префиксни оператори. Потвърждават или променят знака на аргумента си.

Следните означения съдържат унарния оператор + или -:

-i +j -j +i -567

Бинарни оператори

Имат два аргумента. Следните аритметични оператори са инфиксни:

Оператор	Операция
+	събиране
-	изваждане
*	умножение
/	целочислено деление
%	остатък от целочислено деление

Логически оператори

Логическите оператори, реализиращи конюнкция, дизюнкция и отрицание, могат да се прилагат над целочислени константи. Дефинират се по същия начин, както при булевите константи, но целите числа, които са различни от 0 се интерпретират true, а 0 – като false.

Примери:

123 and 0 е false

0 or 15 e true
not 67 e false

Оператори за сравнение

Над цели константи могат да се прилагат следните инфиксни оператори за сравнение:

Оператор	Операция
==	сравнение за равно
!=	сравнение за различно
>	сравнение за по-голямо
>=	сравнение за по-голямо или равно
<	сравнение за по-малко
<=	сравнение за по-малко или равно

Наредбата на целите числа е като в математиката.

Вградени функции

В езика C++ има набор от вградени функции. Обръщението към такива функции има следния синтаксис:

<име_на_функция>(<израз>, <израз>, …, <израз>)

и връща стойност от типа на функцията.

Тук ще разгледаме само едноаргументната целочислена функция abs.

abs(x) – намира |x|, където x е цял израз

Примери:

abs(-1587) = 1587 abs(0) = 0 abs(23) = 23

Въвеждане

Реализира се по стандартния и разгледан вече начин.

Пример: Ако

int i, j;

операторът

cin >> i >> j;

въвежда стойности на целите променливи i и j.

Извеждане

Реализира се чрез оператора

cout << <цяла_константа>;

Други целочислени типове

Други цели типове се получават от int като се използват модификаторите short, long, signed и unsigned. Тези модификатори доопределят някои аспекти на типа int.

За реализацията Visual C++ 6.0 са в сила:

Тип	Диапазон	Необходима памет
short int	-32768 до 32767	2 байта
unsigned shortint	0 до 65535	2 байта
long int	-2147483648 до 2147483647	4 байта
unsigned long int	0 до 4294967295	4 байта
unsigned int	0 до 4294967295	4 байта

Запазената дума int при тези типове се подразбира и може да бъде пропусната. Типовете short int (или само short) и long int (или само long) са съкратен запис на signed short int и signed long int.

Реални типове

Ще разгледаме реалния тип double.

Типът е стандартен, вграден във всички реализации на езика.

Аритметични оператори

Унарни оператори

+, - Префиксни са. Потвърждават или променят знака на аргумента си.

Бинарни оператори

Имат два аргумента. Следните аритметичните оператори са инфиксни:

Оператор	Операция
+	събиране
-	изваждане
*	умножение
/	деление (поне единият аргумент е реален)

Логически оператори

Логическите оператори, реализиращи операциите конюнкция, дизюнкция и отрицание, могат да се прилагат над реални константи. Дефинират се по същия начин, както при булевите константи, като реалните числа, които са различни от 0.0 се интерпретират като true, а 0.0 – като false.

Оператори за сравнение

Над реални данни могат да се прилагат стандартните инфиксни оператори за сравнение (==, !=, >, >=, <, <=).

Допълнение: Сравнението за равно на две реални числа x и y се реализира обикновено чрез релацията: $|x - y| < \varepsilon$, където $\varepsilon = 10\text{-}14$ за тип double. По-добър начин е да се използва релацията:

Вградени функции

При цял или реален аргумент, следните функции връщат реален резултат от тип double:

Функция	Намира
sin(x)	Синус, sin x, x е в радиани
Cos(x)	косинус, cos x, x е в радиани
tan(x)	тангенс, tg x, x е в радиани
asin(x)	аркуссинус, arcsin $x \in [-\pi/2, \pi/2]$, $x \in [-1, 1]$

acos(x)	аркускосинус, arccos x ∈ [0, π], x ∈ [-1, 1]
atan(x)	аркустангенс, arctg x ∈ (-π/2, π/2)
exp(x)	експонента, ex
log(x)	натурален логаритъм, ln x, x > 0
log10(x)	десетичен логаритъм, lg x, x > 0
sinh(x)	хиперболичен синус, sh x
cosh(x)	хиперболичен косинус, ch x
tanh(x)	хиперболичен тангенс, th x
ceil(x)	най-малкото цяло ≥ x, преобразувано в тип double
Floor(x)	най-голямото цяло ≤ x, преобразувано в тип double
fabs(x)	абсолютна стойност на x, |x|
sqrt(x)	Корен квадратен от x, x ≥ 0
pow(x, n)	степенуване, xn (x и n са реални от тип double).

Въвеждане и Извеждане се Реализира по стандартния начин операторите cin и cout.
Други реални типове

В езика C++ има и друг реален тип, наречен float. Различава се от типа double по множеството от стойностите си и заеманата памет.
Множеството от стойности на типа float се състои от реалните числа от диапазона от –3.4*1038 до 3.4*1038. За записване на константите от този диапазон са необходими 4 байта ОП.

Структура от данни масив

Под структура от данни се разбира организирана информация, която може да бъде описана, създадена и обработена с помощта на програма.

За да се определи една структура от данни е необходимо да се направи:
- логическо описание на структурата, което я описва на базата на декомпозицията ѝ на по-прости структури, а също на декомпозиция на операциите над структурата на по-прости операции.
- физическо представяне на структурата, което дава методи за представяне на структурата в паметта на компютъра.

Елементите на структури, които се състоят от една компонента, се наричат прости, или скаларни. Структури от данни, компонентите на които са редици от елементи, се наричат съставни.
Структури от данни, за които операциите включване и изключване на елемент не са допустими, се наричат статични, в противен случай - динамични.
Логическо описание

Масивът е крайна редица от фиксиран брой елементи от един и същ тип. Към всеки елемент от редицата е възможен пряк достъп, който се осъществява чрез индекс. Операциите включване и изключване на елемент в/от масива са недопустими, т.е. масивът е статична структура от данни.

Физическо представяне

Елементите на масива се записват последователно в паметта на компютъра, като за всеки елемент на редицата се отделя определено количество памет.

В езика C++ структурата масив се реализира чрез типа масив.
Тип масив

В C++ структурата от данни масив е реализирана малко ограничено. Разглежда се като крайна редица от елементи от един и същ тип с пряк достъп до всеки елемент, осъществяващ се чрез индекс с цели стойности, започващи от 0 и нарастващи с 1 до указана горна граница. Дефинира се от програмиста.

Дефиниране на масив

Типът масив се определя чрез задаване на типа и броя на елементите на редицата, определяща масив. Нека T е име или дефиниция на произволен тип, различен от псевдоним, void и функционален. За типа T и константния израз от интегрален или изброен тип с положителна стойност size, T[size] е тип масив от size елемента от тип T. Елементите се индексират от 0 до size–1. T се нарича базов тип за типа масив, а size – горна граница.

Примери:

int[5] дефинира масив от 5 елемента от тип int, индексирани от 0 до 4;
double[10] дефинира масив от 10 елемента от тип double, индексирани от 0 до 9;
bool[4] дефинира масив от 4 елемента от тип bool, индексирани от 0 до 3.
Множество от стойности

Множеството от стойности на типа T[size] се състои от всички редици от по size елемента, които са произволни константи от тип T. Достъпът до елементите на редиците е пряк и се осъществява с помощта на индекс, като достъпът до първия елемент се осъществява с индекс със стойност 0, до последния – с индекс със стойност size-1, а до всеки от останалите елементи – с индекс със стойност с 1 по-голяма от тази на индекса на предишния елемент. Елементите от множеството от стойности на даден тип масив са константите на този тип масив.
Примери:
1. Следните редици {1,2,3,4,5}, {-3, 0, 1, 2, 0}, {12, -14, 8, 23, 1000} са константи от тип int[5].

2. Редиците {1.5, -2.3, 3.4, 4.9, 5.0, -11.6, -123.56, 13.7, -32.12, 0.98}, {-13, 0.5, 11.9, 21.98, 0.03, 1e2, -134.9, 0.09, 12.3, 15.6} са константи от тип double[10].
Примери:

```
int a[5];
double c[10];
bool b[3];
enum {FALSE, TRUE} x[20];
double p[4] = {1.25, 2.5, 9.25, 4.12};
```

Дефиницията

Примери:

Дефиницията
int q[5] = {1, 2, 3};
е еквивалентна на
int q[] = {1, 2, 3, 0, 0};
Дефиницията
double r[] = {0, 1, 2, 3};
е еквивалентна на

```
double r[4] = {0, 1, 2, 3};
```

Дефиницията с инициализация е един начин за свързване на променлива от тип масив с конкретна константа от множеството от стойности на този тип масив. Друг начин предоставят т.нар. индексирани променливи.
Всяка индексирана променлива е от базовия тип.

Синтаксис на индексираните променливи

Примери:
1. С променливата a, дефинирана по-горе, са свързани индексираните променливи a[0], a[1], a[2], a[3] и a[4], които са от тип int.
2. С променливата b са свързани индексираните променливи b[0], b[1],…, b[9], които са от тип double.
3. С променливата x са свързани индексираните променливи x[0], x[1],…, x[19], които са от тип enum {FALSE, TRUE}.

Дефиницията на променлива от тип масив не само свързва променливата с множеството от стойности на указания тип, но и отделя определено количество памет (обикновено 4B), в която записва адреса в паметта на първата индексирана променлива на масива. Останалите индексирани променливи се разполагат последователно след първата. За всяка индексирана променлива се отделя по толкова памет, колкото базовият тип изисква.
Операции и вградени функции

Не са възможни операции над масиви като цяло, но всички операции и вградени функции, които базовият тип допуска, са възможни за индексираните променливи, свързани с масива.

Пример: Нека

```
int a[5], b[5];
```

Недопустими са:

```
cin >> a >> b;
a = b;
```

а също a == b или a != b.

Операторът

```
cout << a;
```

е допустим и извежда адреса на a[0].
Някои приложения на структурата от данни масив са: Търсене на елемент в редица, Сортиране на редица, Сливане на редици (за повече инфо за писмения изпит - вж. учебника)
Друго приложение на типът масив е за представяне на символни низове в C++.
Разглежданите досега масиви се наричат едномерни. Те реализират крайни редици от елементи от скаларен тип. Възможно е обаче типът на елементите да е масив. В този случай се говори за многомерни масиви.

Многомерни масиви

Масив, базовият тип на който е едномерен масив, се нарича двумерен. Масив, базовият тип на който е двумерен масив, се нарича тримерен и т.н. На практика се използват масиви с размерност най-много 3.

Дефиниране на многомерни масиви

Нека T е име или дефиниция на произволен тип, различен от псевдоним, void и функционален, size1, size2, …, sizen (n>1 е дадено цяло число) са константни изрази от интегрален или изброен тип с положителни стойности. T[size1][size2] … [sizen] е тип n-мерен масив от тип T. T се нарича базов тип за типа масив.

Примери:

int [5][3] дефинира двумерен масив от тип int;

double [4][5][3] дефинира тримерен масив от тип double;

Синтаксис на многомерен масив

Примери:

```
int x[10][20];
double y[20][10][5];
int z[3][2] = {{1, 3},
{5, 7},
{2, 9}};
int t[2][3][2] = {{{1, 3}, {5, 7}, {6, 9}},
        {{7, 8}, {1, 8}, {-1, -4}}};
```

Дефиницията на променлива от тип многомерен масив не само свързва променливата с множеството от стойности на указания тип, но и отделя определено количество памет за разполагане на индексираните променливи, свързани с нея. Индексираните променливи се разполагат последователно по по-бързото нарастване на по-далечните си индекси. За всяка индексирана променлива се отделя толкова памет, колкото базовият тип изисква.

Забележка: Двумерните масиви разполагат в ОП индексираните си променливи по по-бързото нарастване на втория индекс. Това физическо представяне се нарича представяне по редове. Тези масиви могат да бъдат използвани за реализация и работа с матрици и др. правоъгълни таблици.

Важно допълнение: При работа с масиви трябва да се има предвид, че повечето реализации не проверяват дали стойностите на индексите са в рамките на границите, зададени при техните дефиниции. Тази особеност крие опасност от допускане на трудноотриваеми грешки.

Тип указател

Променлива, това е място за съхранение на данни, което може да съдържа различни стойности. Идентифицира се с дадено от потребителя име (идентификатор). Има си и тип. Дефинира се като се указват задължително типът и името й. Типът определя броя на байтовете, в които ще се съхранява променливата, а също и множеството от операциите, които могат да се изпълняват над нея. Освен това, с променливата е свързана и стойност – неопределена или константа от типа, от който е тя. Нарича се още rvalue. Мястото в паметта, в което е записана rvalue, се идентифицира с адрес, който се нарича адрес на променливата или lvalue. По-точно адресът е адреса на първия байт от множеството байтове, отделени за променливата.

Пример: Фрагментът

```
int i = 1024;
```

дефинира променлива с име i и тип int. Стойността й (rvalue) е 1024. i именува място от паметта (lvalue) с размери 4 байта, като lvalue е адреса на първия байт на това място.

Намирането на адреса на дефинирана променлива става чрез унарния префиксен дясно-асоциативен оператор & (амперсанд). Приоритетът му е същия като на унарните оператори +, -, !, ++, -- и др.

Оператор &

Синтаксис

&<променлива>

където <променлива> е вече дефинирана променлива.

Семантика

Намира адреса на <променлива>.

Пример: &i е адреса на променливата i и може да се изведе чрез оператора cout << &i;

Операторът & не може да се прилага върху константи и изрази, т.е. &100 и &(i+5) са недопустими обръщения. Не е възможно също прилагането му и върху променливи от тип масив, тъй като те имат и смисъла на константни указатели.

Адресите могат да се присвояват на специален тип променливи, наречени променливи от тип указател или само указатели.

Дефиниране

Нека T е име или дефиниция на тип. За типа T, T* е тип, наречен указател към T. T се нарича указван тип или тип на указателя.

Примери:

int* е тип указател към тип int;

enum {a, b, c}* е тип указател към тип enum {a, b, c}.

Множество от стойности

Състои се от адресите на данните от тип T, дефинирани в програмата, преди използването на T*. Те са константите на типа T*. Освен тях съществува специална константа с име NULL, наречена нулев указател. Тя може да бъде свързвана с всеки указател независимо от неговия тип. Тази константа се интерпретира като “сочи към никъде”, а в позиция на предикат е false.

Дефиниция на променлива от тип указател

Дефиницията

T* a, b;

е еквивалентна на

T* a;

T b;

т.е. само променливата a е от тип указател към тип T.

Примери: Дефиницията

int *pint1, *pint2;

задава два указателя към тип int, а

int *pint1, pint2;

- указател pint1 към int и променлива pint2 от тип int.

Дефиницията на променлива от тип указател предизвиква в ОП да се отделят 4B, в които се записва някакъв адрес от множеството от стойности на съответния тип, ако дефиницията е с инициализация и неопределено или NULL, ако дефиницията не е с инициализация. Този адрес е стойността на променливата от тип указател, а записаното на този адрес е съдържанието ѝ.

Пример: В резултат от изпълнението на дефинициите

```
int i = 12;
int* p = &i;              // p е инициализирано с адреса на i
double *q = NULL; // q е инициализирано с нулевия указтел
double x = 1.56;
double *r = &x;         // r е инициализирано с адреса на x
```

Операции и вградени функции

Извличане на съдържанието на указател

Осъществява се чрез префиксния, дясноасоциативен унарен оператор *. * има приоритет на унарен оператор.

Оператор *
Синтаксис
*<променлива_от_тип_указател>
Семантика
Извлича стойността на адреса, записан в <променлива_от_тип_ указател>, т.е. съдържанието на <променлива_от_тип_ указател>.
Като използваме дефинициите от примера по-горе, имаме:
*p е 12// 12 е съдържанието на p
*r е 1.56 // 1.56 е съдържанието на r

Освен, че намира съдържанието на променлива от тип указател, обръщението *<променлива_от_тип_указател>
е променлива от тип T. (Всички операции, допустими за типа T, са допустими и за нея.

Като използваме дефинициите от примера по-горе, *p и *r са цяла и реална променливи, съответно. След изпълнение на операторите за присвояване

```
*p = 20;
*r = 2.18;
```

стойността на i се променя на 20, а тази на r – на 2.18.
Аритметични и логически операции

Променливите от тип указател могат да участват като операнди в следните аритметични и логически операции +, -, ++, --, ==, !=, >, >=, < и <=. Изпълнението на аритметични операции върху указатели е свързано с някои особености, заради което аритметиката с указатели се нарича още адресна аритметика. Особеността се изразява в т. нар. мащабиране. Ще го изясним чрез пример.

Да разгледаме фрагмента

```
int *p;
double *q;
...
p = p + 1;
q = q + 1;
```

Операторът p = p + 1; свързва p не със стойността на p, увеличена с 1, а с p + 1*4, където 4 е броя на байтовете, необходими за записване на данна от тип int (p е указател към int). Аналогично, q = q + 1; увеличава стойността на q не с 1, а с 8, тъй като q е указател към double (8 байта са необходими за записване на данна от този тип).
Общото правило е следното: Ако p е указател от тип T*, p+i е съкратен запис на p + i*sizeof(T), където sizeof(T) е функция, която намира броя на байтовете, необходими за записване на данна от тип T.
Въвеждане

Не е възможно въвеждане на данни от тип указател чрез оператора cin. Свързването на указател със стойност става чрез инициализация или оператора за присвояване.

Извеждане
Осъществява се по стандартния начин - чрез оператора cout.

Указатели и масиви

В C++ има интересна и полезна връзка между указателите и масивите. Изразява се в това, че имената на масивите са указатели към техните "първи" елементи. Последното позволява указателите да се разглеждат като алтернативен начин за обхождане на елементите на даден масив.

Указатели и едномерни масиви

Нека a е масив, дефиниран по следния начин:
int a[100];
Тъй като a е указател към a[0], *a е стойността на a[0], т.е. *a и a[0] са два различни записа на стойността на първия елемент на масива. Тъй като елементите на масива са разположени последователно в паметта, a + 1 е адреса на a[1], a + 2 е адреса на a[2] и т.н. a + n-1 е адреса на a[n-1]. Тогава *(a+i) е друг запис на a[i] (i = 0, 1, ..., n-1).
Има обаче една особеност. Имената на масивите са константни указатели. Заради това, някои от аритметичните оператори, приложими над указатели, не могат да се приложат над масиви. Такива са ++, -- и присвояването на стойност. Използването на указатели е по-бърз начин за достъп до елементите на масива и заради това се предпочита. Индексираните променливи правят кода по-ясен и разбираем. В процеса на компилация всички конструкции от вида a[i] се преобразуват в *(a+i), т.е. операторът за индексиране [] се обработва от компилатора чрез адресна аритметика. Полезно е да отбележим, че операторът [] е ляво-асоциативен и с по-висок приоритет от унарните оператори (в частност от оператора за извличане на съдържание *).

Указатели и двумерни масиви

Името на двумерен масив е константен указател към първия елемент на едномерен масив от константни указатели. Ще изясним с пример казаното.
Нека a е двумерен масив, дефиниран по следния начин: int a[10][20];
Променливата a е константен указател към първия елемент на едномерния масив a[0], a[1], ..., a[9], като всяко a[i] е константен указател към a[i][0] (i = 0,1, ..., 9), т.е.

a[i] == *(a + i)

Като използваме, че операторът за индексиране е лявоасоциативен и с по-висок приоритет от оператора *, получаваме:

a[i][j] == (*(a+i))[j] == *(*(a+i)+j).

(вж. задачи за писмения в учебника)

Указатели и низове

Низовете са масиви от символи. Името на променлива от тип низ е константен указател, както и името на всеки друг масив. Така всичко, което казахме за връзката между масив и указател е в сила и за низ – указател.

Следващият пример илюстрира обхождане на низ чрез използване на указател към char. Обхождането продължава до достигане на знака за край на низ.

```
#include <iostream.h>
int main()
{char str[] = "C++Language"; // str е константен указател
 char* pstr = str;                    // pstr е указател към низа str
 while (*pstr)
 {cout << *pstr << '\n';
  pstr++;
 }                    // pstr вече не е свързан с низа “C++Language”.
 return 0;
}
```

Тъй като низът е зададен чрез масива от символи str, str е константен указател и не може да бъде променяна стойността му. Затова се налага използването на помощната променлива pstr. Ако низът е зададен чрез указател към char, както е в следващата програма, не се налага използването на такава.

```
#include <iostream.h>
int main()
{char* str = "C++Language";  // str е променлива
 while (*str)
 {cout << *str << '\n';
  str++;
 }
 return 0;
}
```

Примерите показват, че задаването на низ като указател към char има предимство пред задаването като масив от символи. Ще отбележим обаче, че дефиницията

```
char* str = "C++Language";
```

не може да бъде заменена с

```
char* str;
cin >> str;
```

следвани с въвеждане на низа “C++Language”(ще напомним, че не е възможно въвеждане на стойност на променлива от тип указател чрез оператора cin), докато дефиницията

```
char str[20];
```

позволява въвеждането му чрез cin, т.е.

```
cin >> str;
```

е допустимо.

Функции

Добавянето на нови оператори и функции в приложенията, реализирани на езика C++, се осъществява чрез функциите. Те са основни структурни единици, от които се изграждат програмите на езика. Всяка функция се състои от множество от оператори, оформени подходящо за да се използват като обобщено действие или операция. След като една функция бъде дефинирана, тя може да бъде изпълнявана многократно за различни входни данни.

Програмите на езика C++ се състоят от една или повече функции. Сред тях задължително трябва да има точно една с име main и наречена главна функция. Тя е първата функция, която се изпълнява при стартиране на програмата. Главната функция от своя страна може да се обръща към други функции. Нормалното изпълнение на

програмата завършва с изпълнението на главната функция (Възможно е изпълнението да завърши принудително с изпълнението на функция, различна от главната).

Използването на функции има следните предимства:

- Програмите стават ясни и лесни за тестване и модифициране.
- Избягва се многократното повтаряне на едни и същи програмни фрагменти. Те се дефинират еднократно като функции, след което могат да бъдат изпълнявани произволен брой пъти.
- Постига се икономия на памет, тъй като кодът на функцията се съхранява само на едно място в паметта, независимо от броя на нейните изпълнения.

8.3 Дефиниране на функции

Синтаксис

Дефиницията на функция се състои от две части: заглавие (прототип) и тяло.

Дефиниция на функция

Модификаторите са спецификатори, които задават препоръка за компилатора (inline), класа памет (extern или static) и др. характеристики. Ще дадем примери в следващите разглеждания. Ако <модификатор> е пропуснат, подразбира се extern.
Типът на функцията е произволен тип без масив и функционален, но се допуска да е указател към такива обекти (в широкия смисъл на думата). Ако е пропуснат, подразбира се int.
Името на функцията е произволен идентификатор. Допуска се неоднозначност.
Списъкът от формални параметри (нарича се още сигнатура) може да е празен или void.
Например, следната функция извежда текст:

```
void printtext(void)
{cout << “C++ Programming Language \n”
        cout << “B. Stroustrup \n”;
        return;
}
```

В случай, че списъкът е непразен, имената на параметрите трябва да са различни. Те заедно с името определят еднозначно функцията. Формалните параметри са: параметри – стойности, параметри – указатели и параметри – псевдоними. Името на параметъра се предшества от тип.

Примери:

int a, int const& b, double& x, int const * y, const int* a

Засега няма да използваме параметри, специфицирани със const.
Тялото на функцията е редица от дефиниции и оператори. Тя описва алгоритъма, реализиращ функцията. Може да съдържа един или повече оператора return.
Операторът return (Фиг. 8.5) връща резултата на функцията в мястото на извикването.

Оператор return

Синтаксис

return [<израз>]опц

където

- return е запазена дума;
- <израз> е произволен израз от тип <тип_на_функцията> или съвместим с него. Ако типът на функцията е void, <израз> се пропуска. В този случай е възможно и return да се пропусне.

Семантика

Пресмята се стойността на <израз>, конвертира се до типа на функцията (ако е възможно) и връщайки получената стойност в мястото на извикването на функцията, прекратява изпълнението ѝ.

Оператор return

Забележка: Ако функцията не е от тип void, тя задължително трябва да върне стойност. Това означава, че операторът return трябва да се намира във всички разклонения на тялото. В противен случай, повечето компилатори ще изведат съобщение или предупреждение за грешка. Възможно е обаче функцията да върне случайна стойност, което е лошо. По-добре е функцията да върне някаква безобидна стойност, отколкото случайна.

Обръщение към функция

Синтаксис

<обръщение_към_функция> ::=
<име_на_функция>()|
<име_на_функция>(void)|
<име_на_функция>(<фактически_параметри>)
където <фактически_параметри> са толкова на брой, колкото са формалните параметри. Освен по брой, формалните и фактическите параметри трябва да си съответстват по тип, по вид и по смисъл.

Съответствието по тип означава, че типът на i-тия фактически параметър трябва да съвпада (да е съвместим) с типа на i-тия формален параметър. Съответствието по вид се състои в следното: ако формалният параметър е параметър-указател, съответният му фактически параметър задължително е променлива или адрес на променлива, ако е параметър-псевдоним, съответният му фактически параметър задължително е променлива (за реализацията Visual C++, 6.0 от същия тип) и ако е параметър-стойност – съответният му фактически параметър е израз.

Семантика

Обръщението към функция е унарна операция с най-висок приоритет и с операнд - името на функцията. Последното пък е указател със стойност адреса на мястото в паметта където е записан програмният код на функцията. Ако функцията определя процедура, обръщението към нея се оформя като оператор (завършва с ;). Опитът за използването ѝ като израз предизвиква грешка. Ако функцията връща резултат както чрез return, така и чрез някой от формалните си параметри, обръщението към нея може да се разглежда и като оператор, и като израз. И ако функцията връща резултат единствено чрез оператора return, обръщението към нея има единствено смисъла на израз. Използването му като оператор не води до грешка, но не предизвиква видим резултат.

Обръщението към функция предизвиква генериране на нова стекова рамка и се осъществява на следните два етапа:

Свързване на формалните с фактическите параметри

За целта първият формален параметър се свързва с първия фактически, вторият формален параметър се свързва с втория фактически и т.н. последният формален

параметър се свързва с последния фактически параметър. Свързването се реализира по различни начини в зависимост от вида на формалния параметър.

а) формален параметър – стойност

В този случай се намира стойността на съответния му фактически параметър. В стековата рамка на функцията за формалния параметър се отделя толкова памет, колкото типът му изисква и в нея се откопирва стойността на фактическия параметър.

б) формален параметър – указател

В този случай в стековата рамка на функцията за формалния параметър се отделят 4B, в които се записва стойността на фактическия параметър, която е адрес на променлива. Действията, описани в тялото се изпълняват със съдържанието на формалния параметър - указател. По такъв начин е възможна промяна на стойността на променливата, чийто адреа е предаден като фактически параметър.

в) формален параметър – псевдоним

Формалният параметър-псевдоним се свързва с адреса на фактическия. За него в стековата рамка на функцията памет не се отделя. Той просто “прелита” и се “закачва” за фактическия си параметър. Действията с него се извършват над фактическия параметър.

Изпълнение на тялото на функцията

Аналогично е на изпълнението на блок.

При всяко обръщение към функция в програмния стек се включва нов “блок” от данни. В него се съхраняват формалните параметри на функцията, нейните локални променливи, а също и някои “вътрешни” данни като return-адреса и др. Този блок се нарича стекова рамка на функцията.

В дъното на стека е стековата рамка на main. На върха на стека е стековата рамка на функцията, която се обработва в момента. Под нея е стековата рамка на функцията, извикала функцията, обработваща се в момента. Ако изпълнението на една функция завършва, нейната стекова рамка се отстранява от стека.

```
Формални параметри
Адрес за връщане
Адрес на предходна рамка на стека
Локални параметри
```

Видът на стековата рамка зависи от реализацията. С точност до наредба, тя има вида:

Стекова рамка

Масивите като формални параметри

Едномерни масиви

Съществуват различни начини за задаване на формални параметри от тип едномерен масив.

а) традиционен

Дефиницията

T a[]

където T е скаларен тип, задава параметър a от тип едномерен масив с базов тип T. Може да се укаже горна граница на масива, но компилаторът я пренебрегва.

Примери:

int a[] - a е параметър от тип масив от цели числа,

int a[10] - еквивалентна е на int a[],

double b[] - b е параметър от тип масив от реални числа,

char c[] - c е параметър от тип масив от символи.

б) чрез указател

Дефиницията

T* p

където T е скаларен тип, задава параметър p от тип указател към тип T. От връзката между масив и указател следва, че тази дефиниция може да се използва и за дефиниране на формален параметър от тип масив.

Примери: Следните дефиниции на формални параметри са еквивалентни на тези от примера по-горе:

int* a - a е параметър от тип указател към int

double* b - b е параметър от тип указател към double.

char* c - c е параметър от тип указател към char.

И в двата случая фактическият параметър се указва с името на едномерен масив от същия тип. Необходимо е също на функцията да се подаде като параметър и размерът на масива.

Забележки:

Функциите работят с направо с масива a, а не с негови копия. Промените на елементите на масива се запазват след излизане от функцията.

Размерът на масивът не може да се разбере от неговото описание. Затова се налага използването на допълнителния параметър m в списъка от аргументи на функциите. Последното не се отнася за масивите, представляващи символни низове, тъй като те завършват със знака за край на низ ‘\0’.

Многомерни масиви

Когато многомерен масив трябва да е формален параметър на функция, в описанието му трябва да присъстват като константи всички размери с изключение на първият. Например, декларацията

void readarr2(int n, int matr[][20]);

определя matr като двумерен масив (редица от двадесеторки от цели числа).

Описанието

int (*matr)[20]

е еквивалентно на

int matr[][20]

Скобите, ограждащи *matr, са задължителни. В противен случай, тъй като [] е с по-висок приоритет от *, int *matr[20] ще се интерпретира като “matr е масив с 20 елемента от тип *int”.

(вж. допълнителни задачи за писмения в учебника)

11. Обектно ориентирано програмиране (на базата на C++ или Java): Основни принципи. Класове и обекти. Конструктори и деструктори. Оператори. Производни класове и наследяване.

(уводни бози)
В днешно време най-разпространените езици за програмиране са т.нар, *езици от високо ниво*. При тях програмистът описва схематично основната идея за решаване на задачата, а специалния програмен транслатор(компилатор или интепретатор) превежда това описание в машинни инструкции за конкретния процесор. Тези езици не зависят от хардуера. Програмирането на високо ниво се свързва с обектно-ориентирания подход – програмиране с помоща на обекти, създването и използването на абстрактен тип данни, наследяването и полиморфизма. Основните характеристики на обектно ориентираното програмиране са:

- Всичко е обект
- Една програма е съвкупснот от обекти, указващи си един на друг чрез изпраяане на съобщения какво да правят
- Обектите притежават собствена памет, суставена от други обекти.
- Всеки обект има тип
- Всички обекти от определен тип могат да получават едни и същи съобщения.

Обектно-ориентираното програмиране се е наложило като стандарт при почти всички съвременни езици за програмиране. То предоставя мощно средство за моделиране на обектите от реалния свят и взаимоотношенията между тях, позволява добро структуриране на програмния код и улеснява неговото преизползване. Благодарение на капсулацията на данните, както и на възможностите за наследяване и работа с абстрактни данни операции, ООП се е утвърдило като предпочитан подход при създаване на големи приложения и библиотеки.

Четирите основни принципа на ООП са капсулация на данните, абстракция, наследяване и полиморфизъм. Те са основните характеристики, които определят един език за програмиране като обектно-ориентиран.

Капсулация на данните
Основна концепция в ООП е обектът да се разглежда като "черна кутия" – използващите обекта "виждат" само атрибутите и операциите, които са присъщи на обекта от реалния свят, без да се интересуват от конкретната им реализация – клиентът на обекта трябва да знае само какво може да прави обектът, а не как го прави. В такъв смисъл "капсулация" означава скриване на ненужните детайли за обектите и откриване към външния свят само на важните техни характеристики и свойства. Обектите в ООП съдържат своите данни и средствата за тяхната обработката, капсулирани като едно цяло.

Абстракция
Абстракцията опростява сложната реалност чрез моделиране на класове подходящи за разрешаването на определен проблем. Тя представлява модел, изглед или някакво друго общо представяне на действителен обект. Абстракцията е разработване на програмен обект представящ обект от реалния свят, а капсулацията му скрива детайлите на имплеметацията. Абстракцията на данните налага ясно разделяне между абстрактните свойства на съответния тип и конкретните детайли на представянето. Идеята е, че промени в това представяне не трябва да имат влияние върху код използващ абстракцията. Нека разгледаме един прост пример: имаме клас Person съответно със

ссвойства Height, Weight, Age и т.н. и действия, които може да се извършват с него: Run(), Sleep(), Cry(). Абстрактното представяне на Person се определя от свойствата които притежава и от действията, които могат да се извършват върху него и от информацията, която може да получим или предадем на него. Конкретната имплементация на действията и свойствата на обекта остава скрита от външния свят, който няма достъп до нея.

Наследяване

Ако един обект съдържа всички свойства и действия на друг, първият може да го наследи. По този начин наследеният обект освен собствените си атрибути и операции приема и тези на "родителя" си (базовия клас), като така се избягва повторното им дефиниране и се позволява създаването на йерархии от класове, моделиращи по естествен начин зависимостите от реалността.

За да изясним това понятие, ще си послужим с класическият в ООП пример за класа от обекти Animal, който представлява абстракция за множеството от всички животни. Всички обекти от този клас имат общи характеристики (например цвят и възраст) и обща функционалност, например операциите Eat и Sleep, докато за класът Dog, представляващ множеството от всички кучета, които също са животни, би могъл да предоставя операциите Eat, Sleep и Bark. Удачно е класът Dog да наследи Animal – тогава той ще съдържа описание само на собственото си действие Bark, докато тези на Eat и Sleep ще получи от базовия си клас. Чрез наследяването се постига специализация, или конкретизация на класовете, тъй като базовият клас представлява категория от обекти по-обща от тази на наследяващите го. Ако си послужим с горния пример, множествата на кучетата и котките са подмножества на множеството от всички животни.

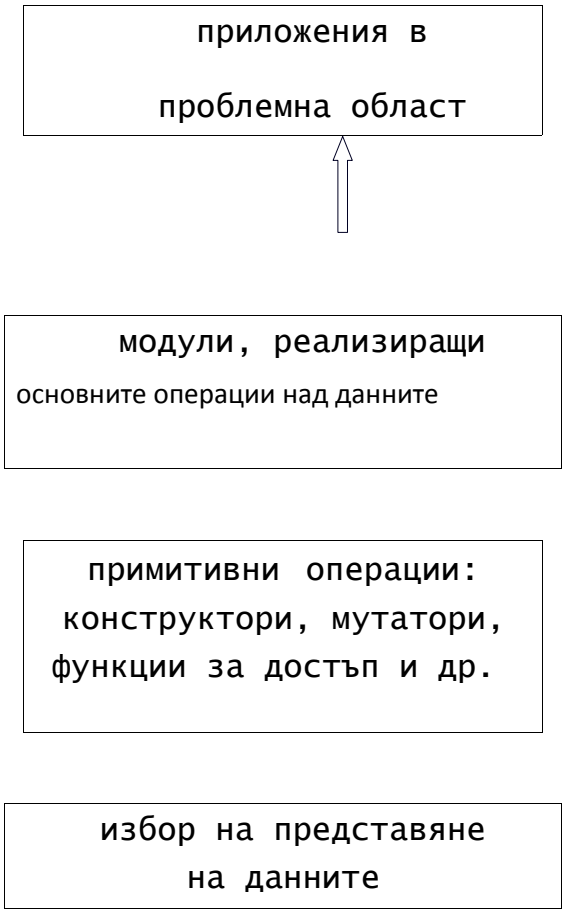
Полиморфизъм

Полиморфизъм буквално означава приемането на различни форми от един обект. Нека е даден базов клас, представящ категория от обекти, които реализират общо действие, което се наследява от множество класове, описващи по-тесни категории. Въпреки, че те всички споделят това действие, те могат да го реализират по различен начин. Когато разполагаме с обекти от базовия клас, знаем че всички те реализират това действие, независимо на кой наследен клас принадлежат. Поради това можем да го използваме без да се интересуваме от конкретната му реализация. Например, ако класът Animal предоставя действието Talk и разгледаме наследяващите го класове Dog и Cat, всеки от тях го реализира по конкрете начин. И ако имаме животно, което издава звук и то е куче – ще лае, а ако е котка – ще мяучи. Полиморфизмът позволява унифицираното извършване на действие над различни обекти, които го реализират. В този случай издаването на звук от животно е полиморфно действие – такова, което се реализира по различен начин в различните наследници на базовия клас.

Класове

Класовете са нови типове данни, дефинирани от потребителя. Те могат да обогатяват възможностите на вече съществуващ тип или да представят напълно нов тип данни. Класовете са подобни на структурите, даже може да се каже, че в някои отношения са почти идентични. В C++ класът може да се разглежда като структура, на която са наложени някои ограничения по отношение на правата на достъп. Отначало ще разгледаме основното, което е приложимо както за класовете, така и за структурите.

Основен принцип на процедурното програмиране е модулния. Програмата се разделя на "подходящи" взаимносвързани части (функции, модули), всяка от които се реализира чрез определени средства. Важен е обаче начинът, по който да става определянето на частите и връзките помежду им. Целта е, следващи промени в представянето на данните да не променят голям брой от модулите на програмата. Разсъждения в тази посока довеждат до подхода абстракция със структури от данни. При него методите за използване на данните се разделят от методите за тяхното конкретно представяне. Програмите се конструират така, че да работят с "абстрактни данни" – данни с неуточнено представяне. След това представянето се конкретизира с помощта на множество функции, наречени конструктори, мутатори и функции за достъп, които реализират "абстрактните данни" по конкретен начин. Така при решаването на даден проблем се оформят следните нива на абстракцията:



Добра реализация на подхода е тази, при която всяко ниво използва единствено средствата на предходното. Предимствата са, че възникнали промени на едно ниво ще се отразят само на следващото над него. Например, промяна на представянето на данните ще доведе до промени единствено на реализацията на някои от конструкторите, мутаторите или функциите за достъп.

Дефиниране на класове

Класовете осигуряват механизми за създаване на напълно нови типове данни, които могат да бъдат интегрирани в езика,

а също за обогатяване възможностите на вече съществуващи типове. Дефинирането на един клас се състои от две части:

- декларация на класа и
- дефиниция на неговите член-функции (методи).

14.2.1. Декларация на клас

Декларацията на клас се състои от заглавие и тяло. Заглавието започва със запазената дума class, следвано от името на класа. Тялото е заградено във фигурни скоби. След скобите стои знакът “;” или списък от обекти. В тялото на класа са декларирани членовете на класа (член-данни и член-функции) със съответните им нива на достъп.

(това изглежда на индиански първоначално, ама като го погледате 10-15 мин се разбира :Д)

Декларация на клас

```
<декларация_на_клас> ::= <заглавие> <тяло>
<заглавие> ::= class [<име_на_клас>]опц
<тяло> ::= {<декларация_на_член>;
            {<декларация_на_член>;}опц
           }[<списък_от_обекти>]опц;
<декларация_на_член> ::=
                              <декларация_на_конструктор>|
<декларация_на_мутатор>|
                         <декларация_на_функция_за_достъп>|
<декларация_на_данна>
<декларация_на_конструктор> ::=
      [<спецификатор_на_достъп>:]опц<име_на_клас>(<парамет
ри>)
<декларация_на_мутатор> ::=
     [<спецификатор_на_достъп>:]опц <тип>
                          <име_на_мутатор>(<параметри>)
<декларация_на_функция_за_достъп> ::=
    [<спецификатор_на_достъп>:]опц <тип>
                   <име_на_функция_за_достъп>(<параметри>)
const;
<спецификатор_на_достъп>  ::=  private  |  public  |
protected
```

```
   <параметри> :: <празно> | void |
                          <параметър> {, <параметър>}опц
   <параметър> ::= <тип> [ &|опц * [const]опц] опц
   <декларация_на_данна>    ::=    <тип>    <име_на_данна>{,
<име_на_данна>}опц
   <тип> ::= <име_на_тип>|<дефиниция_на_тип>
   <списък_от_обекти> ::=
       <обект> [= <име_на_клас>(<фактически_параметри>)]опц
               {,<обект>[=<име_на_клас>(<фактически_парамет
ри>)]опц }опц
              {, <обект>(<фактически_параметри>)}опц
              {, <обект> = <вече_дефиниран_обект>}опц
където  <име_на_клас>,  <име_на_мутатор>,  <име_на_данна>,
<обект> и <име_на_функция_за_достъп> са идентификатори, а
<фактически_ параметри> е определено в Глава 8.
```

Фиг. 14.1 Декларация на клас

За имената на класовете важат същите правила, които се прилагат за имената на всички останали типове и променливи. Също като при структурите името на класа може да бъде пропуснато.

Имената на членовете на класа са локални за него, т.е. в различни класове в рамките на една програма могат да се дефинират членове с еднакви имена. Член-данни от един и същ тип могат да се изредят, разделени със запетая и предшествани от типа им.

Пример:

```
class point
{private:
  double x, y;      // x и y са член-данни
 public:
  point(double, double);
  void read();
  int get_x() const;
  int get_y() const;
  void print() const;
}p=point(2,7), q(-2,3), r=q;
```

В тялото, някои декларации на членове могат да бъдат предшествани от **спецификаторите на достъп** private, public или protected. Областта на един спецификатор на достъп започва от спецификатора и продължава до следващия спецификатор. Подразбиращ се спецификатор за достъп е private. Един и същ спецификатор на достъп може да се използва повече от веднъж в декларация на клас.

Препоръчва се, ако секция public съществува, да бъде първа в декларацията, а секцията private да бъде последна в тялото на класа.

Достъпът до членовете на класовете може да се разгледа на следните две нива:

- По отношение на *член-функциите в класа* е в сила, че те имат достъп до всички членове на класа. При това не е необходимо тези компоненти да се предават като параметри. Този режим на достъп се нарича **режим на пряк достъп**.

- По отношение на *функциите, които са външни за класа*, режимът на достъп са определя от начина на деклариране на членовете.

Членовете на даден клас, декларирани като private (декларирани след запазената дума private) са видими (достъпни) само в рамките на класа. Външните функции нямат достъп до тях. По подразбиране членовете на класовете са private. Чрез използването на членове, обявени като private, се постига скриване на членове за външната за класа среда. Процесът на скриване се нарича още **капсулиране на информацията**.

Членовете на клас, които трябва да бъдат видими извън класа (да бъдат достъпни за функции, които не са методи на дадения клас) трябва да бъдат декларирани като public (декларирани след запазената дума public).

14.2.2 Дефиниране на методите на клас

След декларирането на клас, трябва да се дефинират неговите методи. Дефинициите са аналогични на дефинициите на функции, но името на метода се предшества от името на класа, на който принадлежи метода, следвано от оператора за

принадлежност :: (Нарича се още оператор за област на действие). Такива имена се наричат **пълни.**

Дефиниция на метод на клас
<дефиниция_на_метод_на_клас> ::=
[<тип>]$_{опц}$ <име_на_клас>::<име_на_функция>(<параметри>) [const]$_{опц}$
{<тяло>}
<тяло> ::= <редица_от_оператори_и_дефиниции>
където <име_на_клас> и <име_на_функция> са идентификатори, а <параметри> се определя както в дефиниция на функция.

Дефиниция на метод на клас

Ще отбележим, че дефиницията на конструктор **не започва** с <тип>, а запазената дума const може да присъства само в дефинициите на функциите за достъп. Добрият стил на програмиране изисква използването на const в дефинициите на функциите за достъп и също в техните декларации. Ако се пренебрегне това изискване, могат да се създадат класове, които да не могат да се използват от други програмисти.

Обикновено дефинициите на методите са разположени веднага след декларирането на класа, на който те са членове. Възможно е обаче, дефинициите на методите на един клас да бъдат част от декларациите на този клас, т.е. в декларациите на член-функциите на класа могат да се зададат не само прототипите им, но и техните тела.

Пример: Класът rat може да бъде дефиниран и по следния начин:

```
class rat
{private:
  int numer;
  int denom;
 public:
  rat()
  {numer = 0;
   denom = 1;
  }
  rat(int a, int b)
```

```
 {if (a == 0 || b==0)
{numer = 0;
 denom = 1;
}
  else
  {int g = gcd(abs(a), abs(b));
   if (a>0 && b>0 || a<0 && b<0)
     {numer = abs(a)/g;
      denom = abs(b)/g;}
    else
 {numer = - abs(a)/g;
  denom = abs(b)/g;
 }
  }
 }
void read()
{cout << "numer: ";
 cin >> numer;
 do
 {cout << "denom: ";
  cin >> denom;
 } while (denom == 0);
}
int get_numer() const
{return numer;
}
int get_denom() const
{return denom;
}
void print() const
{cout << numer << "/" << denom << endl;
}
};
```

В този случай обаче член-функциите се третират като **вградени (inline) функции**.

Често член-функциите се реализират като вградени функции. Това увеличава ефективността на програмата, използваща класа.

Ще отбележим също, че в тялото на дефиницията на член-функция явно не се указва обектът, върху който тя ще се приложи. Този обект участва неявно - чрез член-данните на класа. Заради това се нарича **неявен параметър,** а член-данните – **абстрактни данни**.Параметри, които участват явно в дефиницията на член-функция се наричат **явни**. *Всяка член-функция има точно един неявен параметър и нула или повече явни.*

14.2.3 Област на класовете

За разлика от функциите, класовете могат да се декларират на различни нива в програмата: *глобално* (ниво функция) и *локално* (вътре във функция или в тялото на клас).

Областта на глобално деклариран клас започва от декларацията и продължава до края на програмата. Примерите досега бяха с такива класове.

Ако клас е деклариран във функция, всички негови член-функции трябва да са вградени (inline). В противен случай ще се получат функции, дефинирани във функция, което не е възможно.

Пример:

```
void f(int i, int* p)
{int k;
 class CL
 {public:
     // всички методи са дефинирани в тялото на класа
  ...
  private:
  ...
 };
// тяло на функцията f
 CL x;
 ...
}
```

Областта на клас, дефиниран във функция, е функцията. Обектите на такъв клас са видими само в тялото на функцията.

Възможно е използването на обекти (в широкия смисъл на думата) с еднакви имена. В сила е правилото, че в областта си локалният обект скрива нелокалния.

Не е възможно в тялото на локално дефиниран клас да се използва функцията, в която класът е дефиниран.

Пример: Нека сме в означенията на горния пример.

```
void f(...)
{...
 class cl
 {
     // не може да се използва функцията f
 };
 ...
}
```

14.3 Обекти

След като даден клас е дефиниран, могат да бъдат създавани негови екземпляри, които се наричат **обекти**. Връзката между клас и обект в езика C++ е подобна на връзката между тип данни и променлива, но за разлика от обикновените променливи, обектите се състоят от множество **компоненти** (член-данни и член-функции). На Фиг. 14.3 е даден синтаксисът на дефиниция на обект на клас.

Дефиниция на обект на клас

<дефиниция-на_обект_на_клас> ::=
<име_на_клас> <обект>
[=<име_на_клас>(<фактически_параметри>)]$_{опц}$
{,
<обект>[=<име_на_клас>(<фактически_параметри>)]$_{опц}$ }$_{опц}$
{, <обект>(<фактически_параметри>)}$_{опц}$
{, <обект> = <вече_дефиниран_обект>}$_{опц}$;
<обект> ::= <идентификатор>
където <фактически_параметри> е определено в Глава 8.

Фиг. 14.3 Дефиниция на обект на клас

Декларацията на клас не заделя памет за него. Памет се заделя едва при дефинирането на обект от класа. Дефиницията

rat p, q(2, 3), r = rat(3, 8);

заделя за обектите p, q и r по 8 байта ОП (по 4В за всяка от данните им numer и denom).

Достъпът до компонентите на обектите (ако е възможен) се осъществява чрез задаване на името на обекта и името на данната или метода, разделени с точка. Изключение от това правило правят конструкторите.

Достъп до компонента на обект

<компонента_на_обект> ::= <обект>.<данна>|
<обект>.<име_на_член_функция>()|

<обект>.<име_на_член_функция>(<параметри>)

<име_на_член_функция> е <идентификатор>, означаващ име на мутатор или име на функция за достъп.

Фиг. 14.4 Достъп до компонента на обект

Пример:

```
rat p(1,2), q;
p.get_numer()  // достъп до член-функцията get_numer() за обекта p
q.get_numer()  // достъп до член-функцията get_numer() за обекта q
```

Ще отбележим също, че на практика обектите p и q нямат свои копия на метода get_numer(). И двете обръщения се отнасят за един и същ метод, но при първото обръщение се работи с данните за обекта p, а при второто – с данните за обекта q.

При създаването на обекти на един клас кодът на методите на този клас не се копира във всеки обект, а се намира само на едно място в паметта.

Естествено възниква въпросът *по какъв начин методите на един клас "разбират" за кой обект на този клас са били извикани*. Отговорът на този въпрос дава указателят this. Всяка член-функция на клас поддържа допълнителен формален параметър – указател с име this и от тип <име_на_клас>*.

Обекти от един и същ клас могат да се присвояват един на друг. Присвояването може да е и на ниво инициализация (фиг. 14.3).

Пример: Допустими са дефинициите

```
rat p, q(4,5), r=q;
p = q;
...
r = p;
```

При присвояването се копират всички член-данни на обекта. Така присвояването

```
r = p;
```

е еквивалентно на

```
r.numer = p.numer;
r.denom = p.denom;
```

17.4 Конструктори

Създаването на обекти е свързано с отделяне на памет, запомняне на текущо състояние, задаване на начални стойности и др. дейности, които се наричат **инициализация на обекта.** В езика C++ тези дейности се изпълняват от специален вид член-функции на класовете – конструкторите.

14.4.1. Дефиниране на конструктор

На Фиг. 14.5 е дадена най-често използваната форма за дефиниране на конструктор.

Дефиниране на конструктор (най-често използвана форма)

<дефиниция_на_конструктор> ::=
<име_на_клас>::<име_на_клас>(<параметри>)
{<тяло>}
<тяло> ::= <редица_от_оператори_и_дефиниции>
<параметри> се определя както формални параметри на функция.

Фиг. 14.5 Дефиниране на конструктор (най-често използвана форма)

Ще напомним, че конструкторът е член-функция, която притежава повечето характеристики на другите член-функции, но има и редица особености, като:

- Името на конструктора съвпада с името на класа.
- Типът на резултата е указателят this и явно не се указва.
- Изпълнява се автоматично при създаване на обекти.
- Не може да се извиква явно (обръщение от вида r.rat(1,4) е недопустимо).

Освен това в един клас може явно да не е дефиниран конструктор, но може да са дефинирани и няколко конструктора.

Типична дефиниция на конструктор е дадена в следващия пример.

Пример:

```
class CL
{public:
  CL(int, int, int);
  void print();
  ...
 private:
  int a, b, c;
};
CL::CL(int x, int y, int z) // конструктор с три параметъра
{a = x;
 b = y;
 c = z;
}
```

Класът CL в този пример има един триаргументен конструктор. При създаване на обект на класа, този конструктор ще се изпълнява автоматично, стига да е коректно извикан, в резултат на което член-данните a, b и c на създадения обект ще се свържат със стойностите, които се подадат като фактически параметри на конструктора.

На Фиг. 14.6 е дадена по-обща дефиниция на конструктор.

Дефиниране на конструктор

```
<дефиниция_на_конструктор> ::=
<име_на_клас>::<име_на_клас>(<параметри>):
    <член_данна>(<израз>){,<член_данна>(<израз>)}опц
{<тяло>}
<тяло> ::= <редица_от_оператори_и_дефиниции>
```

Фиг. 14.6 Дефиниране на конструктор

Забелязваме, че е възможно член_данна да се свърже с инициализираща стойност в заглавието на конструктора.

Пример: Конструкторът на класа CL може да се дефинира и по следния начин

```
CL::CL(int x, int y, int z): a(x), b(y), c(z)
{}
```

Ще отбележим, че не е задължително всички член-данни да са инициализирани само пред тялото или само вътре в тялото на конструктора.

Пример: Допустима е дефиницията

```
CL::CL(int x, int y, int z): a(x)
  {b = y;
   c = z;
  }
```

Смисълът на обобщената синтактична конструкция е, че инициализацията на член-данните в заглавието предшества изпълнението на тялото на конструктора. Това я прави изключително полезна. Използването й увеличава ефективността на класа.

14.4.2 Предефинирани конструктори

Обект (в общия смисъл) е предефиниран, ако за него има няколко различни дефиниции, задаващи различни негови интерпретации. За да бъдат използвани такива конструкции е необходим критерии, по който те да се различат.

В рамките на една програма може да се извършва предефиниране на функции. Възможно е:

а) да се използват функции с едно и също име с различни области на видимост

В този случай не възниква проблем с различаването.

б) да се използват функции с едно и също име в една и съща област на видимост

В един клас може да са дефинирани няколко конструктора. Всички те имат едно име (името на класа), но трябва да се различават по броя и/или типа на параметрите си. Наричат се **предефинирани конструктори**. При създаването на обект на класа се изпълнява само един от тях. Определя се съгласно критерия за най-добро съвпадане.

Пример: В класа rat дефинирахме два конструктора rat() и rat(int, int), които се различават по броя на параметрите си.

14.4.3 Подразбиращ се конструктор

В клас може явно да е дефиниран, но може и да не е дефиниран конструктор. Ако явно не е дефиниран конструктор, автоматично се създава един т. нар. **подразбиращ се конструктор**. Този конструктор реализира множество от действия като: заделяне на памет за данните на обект, инициализиране на някои системни променливи и др.

Подразбиращият се конструктор може да бъде предефиниран. За целта е необходимо в класа да бъде дефиниран конструктор без параметри.

Пример: В класа rat, подразбиращият се конструктор беше предефиниран от конструктора

```
rat::rat()
{numer = 0;
 denom = 1;
}
```

14.4.4 Конструктори с подразбиращи се параметри

Функциите в езика C++ могат да имат подразбиращи се параметри. За тези параметри се задават подразбиращи се стойности, които се използват само ако при извикването на функцията не бъде зададена стойност за съответния параметър.

Задаването на подразбираща се стойност се извършва чрез задаване на конкретна стойност в прототипа на функцията или в нейната дефиниция.

При използване на подразбиращи се параметри, важна роля играе редът на параметрите. Прието е, че ако параметър на функция е подразбиращ се, всички параметри след него също са подразбиращи се.

Всичко казано за подразбиращите се параметри на функции се отнася и за конструкторите.

Ако променим дефиницията на класа rat по следния начин:

```
class rat
{private:
  int numer;
  int denom;
 public:
  rat(int=0, int=1);
  void read();
  int get_numer() const;
  int get_denom() const;
  void print() const;
};
```

където конструкторът rat(int, int); се дефинира по същия начин, са допустими следните дефиниции на обекти:

```
rat p,                          // p се инициализира с 0/1
      q = rat(),                // q се инициализира с 0/1
      r = rat(5),           // r се инициализира с 5/1
      s = rat(13,21),       // s се инициализира с 13/21
      t(2);                     // t се инициализира с 2/1
```

14.4.5 Конструктори за присвояване и копиране

Инициализацията на новосъздаден обект на даден клас може да зависи от друг обект на същия клас. За да се укаже такава зависимост се използва знакът за присвояване или кръгли скоби.

Пример: Нека

```
rat p = rat(1,4);
```

Чрез еквивалентните конструкции

```
rat q = p;
rat q(p);
```

се създава обектът q от клас rat, като инициализацията на q зависи от p. Тази инициализация се създава от специален конструктор, наречен **конструктор за присвояване.**

Конструкторът за присвояване е конструктор, поддържащ формален параметър от тип <име_на_клас> const &.

- Ако в един клас явно не е дефиниран конструктор за присвояване, компилаторът автоматично създава такъв, в момента когато новосъздаден обект се инициализира с обекта, намиращ се от дясната страна на знака за присвояване или в кръглите скоби. Този конструктор за присвояване се нарича конструктор за копиране.

Дефиницията rat p=q създава нов обект p *(без викане на конструктор)*, в който се копират съответните стойности на обекта q. Това е резултат от изпълнението на обръщението към rat(&p, q).

- Ако в класа е дефиниран конструктор за присвояване, компилаторът го използва.

Предефиниране на служебния конструктор за копиране се налага когато обектите имат член-данни, указващи динамична памет.

Освен в горните случаи, конструкторът за присвояване (копиране) се използва и при предаване на обект като аргумент на функция, когато предаването е по стойност, а също при връщане на обект като резултат от изпълнение на функция.

14.5 Указатели към обекти на класове

Дефинират се по същия начин както се дефинират указатели към променливи от стандартните типове данни.

Пример:

```
rat p;
rat * ptr = &p;
```

В резултат ще се отделят 4В ОП, които ще се именуват с ptr и ще се инициализират с адреса на обекта p.

Достъпът до компонентите на рационалното число, сочено от ptr, се осъществява по общоприетия начин:

```
(*ptr).get_numer()

(*ptr).get_denom()
```

Синтактичната конструкция (*ptr). е еквивалентна на ptr ->. Така горните обръщения могат да се запишат и по следния начин:

```
ptr -> get_numer()

ptr -> get_denom()
```

14.6 Масиви и обекти

Елементите на масив могат да са обекти, но разбира се от един и същ клас. Дефинират се по общоприетия начин (Фиг. 14.7).

Дефиниция на масив от обекти

<дефиниция_на_променлива_от_тип_масив_от_обекти> ::=

T <променлива>[size] [= {<инициализиращ_списък>}]$_{опц}$;

където

- T е име или декларация на клас;
- <променлива> е идентификатор;
- size е константен израз от интегрален или изброен тип с *положителна* стойност;
- <инициализиращ_списък> се дефинира по следния начин:

<инициализиращ_списък> ::= <стойност>{, <стойност>}$_{опц}$

```
                {,
<име_на_конструктор>(<фактически_параметри>)}опц
```

Фиг. 14.7 Дефиниция на масив от обекти

Пример:

```
rat table[10];
```

определя масив от 10 обекта от клас rat.

Достъпът до елементите на масива е пряк и се осъществява по стандартния начин – чрез индексирани променливи.

Пример: Чрез индексираните променливи

```
table[0], table[1], ..., table[9]
```

се осъществява достъп до първия, втория и т.н. до десетия елемент на table.

Тъй като table[i] (i = 0, 1, ..., 9) са обекти, възможни са следните обръщения към техни компоненти:

```
table[i].read();              // въвежда стойност на table[i]
table[i].print();             // извежда стойността на table[i]
table[i].get_numer(); // намира числителя на table[i]
table[i].get_denom();    // намира знаменателя на table[i].
```

Връзката между масиви и указатели е в сила и в случая когато елементите на масива са обекти. Името на масива е указател към първия му елемент, т.е. ако

```
rat * p = table;         // p сочи към table[0]
                                    // т.е. p==&table[0]
```

*(p+i) == table[i], i = 0, 1,...,9

Тогава

(*(p+i)).print(); // е еквивалентно на table[i].print();

Масивът може да е член-данна на клас. Конструкторите (в частност конструкторът по подразбиране) играят важна роля при дефинирането и инициализирането на масиви от обекти. Масив от обекти, дефиниран в програма, се инициализира по два начина:

- *неявно* (чрез извикване на системния конструктор по подразбиране за всеки обект – елемент на масива);
- *явно* (чрез инициализиращ списък).

Примери:

а) Класът

```
const NUM = 5;
class student
{public:
 void read_student();
 void print_student() const;
 bool is_better(student const &) const;
 double average() const;
 private:
 int facnom;
 char name[26];
 double marks[NUM];
};
```

няма явно дефиниран конструктор. Дефиницията

```
student table[30];
```

на масива table от 30 обекта от клас student е правилна.

Инициализацията се осъществява чрез извикване на “системния” конструктор по подразбиране за всеки обект – елемент на масива.

б) Класът rat, дефиниран по-долу

```
class rat
{private:
  int numer;
  int denom;
 public:
  rat(int=0, int=1);
  void read();
  int get_numer() const;
  int get_denom() const;
  void print() const;
};
```

притежава явно дефиниран конструктор с два подразбиращи се параметъра. В този случай са допустими дефиниции от вида:

```
rat x[10]; // x[i] се инициализира с 0/1, за всяко i=0,1,...9.
rat x[10] = {1,2,3,4,5,6,7,8,9,10}; //x[i] == i/1
rat x[10] = {rat(1,21),rat(2),rat(3, 5),4,5,6,7,8,9,10};
// x[0] == 1/21; x[1] == 2/1; x[2] == 3/5, x[3]== 4/1, ...
```

Ако променим конструктора на класа rat от

```
rat(int=0, int=1);
```

в

```
rat(int, int);
```

т.е. без подразбиращи се параметри и трите дефиниции от по-горе ще съобщят за грешка. Единствено допустима дефиниция на x[10] е с инициализация с 10 обръщения към двуаргументния конструктор rat с явно указани два аргумента.

Динамични обекти

Вече разгледахме в най-общ план разпределението на ОП по време на изпълнението на програма. От Фиг. 8.1 на Глава 8 се вижда, че всяка програма има две "места" за памет: *програмен стек (стек)* и *област за динамичните данни (динамична памет, хийп или куп).*

Стекът е област за временно съхранение на информация. Той е кратковременна памет. C++ използва стека основно за реализиране на обръщения към функции. Всяко обръщение към функция предизвиква конструиране на нова стекова рамка, която се установява на върха на стека. По такъв начин когато функция A извика функция B, която от своя страна вика функция C, стекът нараства. Когато пък всяка от тези функции завършва, стековите рамки на тези функции автоматично се разрушава. Така стекът се свива.

Хийпът е по-постоянна област за съхранение на данни. Той е един вид дълготрайна памет. Особеност на тази памет е, че тя не се свързва с имена на променливи. С разположените в нея обекти се работи косвено – чрез указатели. Обикновено се използва при работа с т. нар. **динамични структури от данни**. *Динамичните данни са такива обекти (в широкия смисъл на думата), чийто брой не е известен в момента на проектирането на програмата. Те се създават и разрушават по време на изпълнението на програмата. След разрушаването им, заетата от тях памет се освобождава и може да се използва отново. Така паметта се използва по-ефективно.*

Създаването и разрушаването на динамични обекти в C++ се осъществява чрез операторите new и delete. Извикването на new заделя в хийпа необходимата памет и връща указател към нея. Този указател може да се съхрани в някаква променлива и да се пази докато е необходимо. За разлика от стека,

заделянето на памет в хийпа е явно – чрез new. Освобождаването на паметта от хийпа също става явно, чрез delete. Всяко извикване на new трябва да бъде балансирано чрез извикване на delete. Последното се налага, тъй като за разлика от стека, хийпът не се изчиства автоматично. В C++ няма система за "събиране на боклуци" (автоматично премахване на обекти, които вече не са необходими). Затова трябва явно да бъдат изтрити създадените в хийпа обекти. Описанието на оператора new е дадено на Фиг. 14.8.

Оператор new

Синтаксис

new <име_на_тип> [[size]]$_{опц}$;|

new <име_на_тип> (<инициализация>);

където

- *<име_на_тип>* е име на някой от стандартните типове int, double, char и др. или е име на клас;

- *size* е израз с произволна сложност, но трябва да може да се преобразува до цял. Показва броя на компонентите от тип <име_на_тип>, за които да се задели памет в хийпа и се нарича **размерност**;

- *<инициализация>* е израз от тип <име_на_тип> или инициализация на обект според синтаксиса на конструктора на класа, ако <име_на_тип> е име на клас.

Семантика

Заделя в хийпа (ако е възможно):

- sizeof(<име_на_тип>) байта, ако не са зададени size и <инициализация> или

- sizeof(<име_на_тип>)*size байта, ако явно е указан size или

- sizeof(<име_на_тип>) байта, ако е специфицирана <инициализация>, която памет се инициализира с <инициализация>

и връща указател към заделената памет.

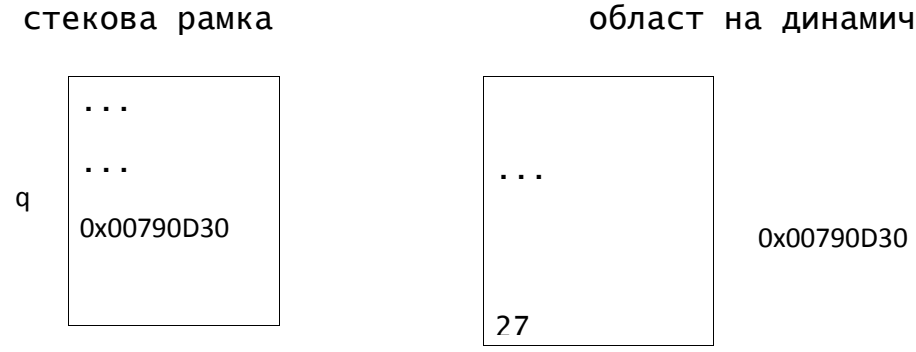
Ако няма достатъчно памет в хийпа, операторът new връща NULL.

Фиг. 14.8 Оператор new

Примери:

a) int* q = new int(2+5*5);

отделя (ако е възможно) 4B памет в хийпа, инициализира я с 27 - стойността на израза 2+5*5 и свързва q с адреса на тази памет, т.е.



Заделянето на памет по време на компилация се нарича **статично** заделяне на памет, заделянето на памет по време на изпълнение на програмата - **динамично разпределение на паметта**.

Под период на **активност на една променлива** се разбира частта от времето за изпълнение на програмата, през което променливата е свързана с конкретно място в паметта. Паметта за глобалните променливи се заделя в началото и остава свързана с тях до завършването на изпълнението на програмата. Паметта за локалните променливи се заделя при влизане в локалната област и се освобождава при напускането ѝ. Паметта на динамичните променливи се заделя от оператора new. Заделената по този начин памет остава свързана със съответната променлива докато не се освободи явно от програмиста. Явното освобождаване на динамична променлива се осъществява чрез оператора delete, приложен към указателя, който адресира съответната променлива. Едно непълно негово описание е дадено на Фиг. 14.9.

Оператор delete

Синтаксис

delete <указател_към_динамичен_обект>;

където <указател_към_динамичен_обект> е указател към динамичен обект (в широкия смисъл на думата), създаден чрез оператора new.

Семантика

Разрушава обекта, адресиран от указателя, като паметта, която заема този обект, се освобождава. Ако обектът, адресиран от указателя, е обект на клас, отначало се извиква деструкторът на класа (т.15.9) и след това се освобождава паметта.

Фиг. 14.9 Оператор delete

Ако в хийпа е заделена памет, след което тази памет не е освободена чрез delete, се получава загуба на памет. Парчето памет, което не е освободено, е като остров в хийпа, заемащо пространство, което иначе би могло да се използва за други цели.

За да се разруши масив, създаден чрез new по следния начин:

```
int* arr = new int[5];
```

трябва да се запише:

```
delete [] arr;
```

Ако обаче масивът съдържа в себе си указатели, първо трябва да бъде обходен и да бъде извикан операторът delete за всеки негов елемент. Едва след това може да се извика delete за масива по показания по-горе начин. Операторът delete трябва да се използва само за освобождаване на динамична памет, заделена с new. В противен случай действието му е непредсказуемо.

Динамичната памет не е неограничена. Тя може да се изчерпи по време на изпълнение на програмата. Неуспешното завършване на delete ускорява изчерпването. Ако наличната в момента динамична памет е недостатъчна, new връща нулев указател и програмата (функцията) няма да работи. Затова се препоръчва след всяко извикване на new да се прави проверка за успешността ѝ.

Чрез оператора new могат да се създават т. нар. **динамични масиви** – масиви с променлива дължина. Динамичните масиви се създават в динамичната памет. Следващата програма илюстрира този процес.

Задача 123. Да се напише програма, която създава динамичен масив от цели числа. Да се изведе масивът.

```
// Program 123. cpp
#include <iostream.h>
int main()
{int size; // дължина на масива
 do
 {cout << "size of array: ";
  cin >> size;
 } while (size<1);
 // създаване на динамичен масив arr от size елемента от
тип int
 int* arr = new int[size];
 int i;
 for (i = 0; i <= size-1; i++)
  arr[i] = i;
 // извеждане на елементите на arr
 for (i = 0; i <= size-1; i++)
  cout << arr[i] << " ";
 cout << endl;
 // освобождаване на заетата динамична памет
 delete [] arr;
 return 0;
}
```

Чрез оператора

```
delete [] arr;
```

е разрушен динамичният масив arr.

Методите на класовете също могат да използват динамична памет, която се заделя и освобождава по време на изпълнението им, чрез операторите new и delete.

Ще отбележим също, че заделената от член-функциите динамична памет не се освобождава автоматично при разрушаване на обектите на класове. Освобождаването на тази памет трябва да стане явно чрез оператора delete, който трябва да се изпълни преди разрушаването на обекта. Този процес може да бъде автоматизиран чрез използване на специален вид методи, наречени **деструктори**.

Деструктори

Разрушаването на обекти на класове в някои случаи е свързано с извършване на определени действия, които се наричат **заключителни**. Най-често тези действия са свързани с освобождаване на заделена преди това динамична памет, възстановяване на състояние на програмата и др. Ефектът от заключителните действия е противоположен на ефекта на инициализацията. Естествено е да се даде възможност заключителните действия да се извършат автоматично при разрушаването на обекта. Това се осъществява от деструкторите.

Деструкторът е член-функция, която се извиква при:
- разрушаването на обект чрез оператора delete,
- излизане от блок, в който е бил създаден обект от класа.

Един клас може да има явно дефиниран точно един деструктор. Името му съвпада с името на класа, предшествано от символа '~' (тилда), типът му е void и явно не се задава в заглавието. Деструкторът няма формални параметри.

Забележка: Използването на явно дефинирани деструктори не винаги е належащо, тъй като всички член-променливи се разрушават при разрушаването на обекта и без използването на деструктор. Ако конструкторът или някоя член-функция реализира динамично заделяне на памет за някоя член-данна, използването на деструктор е задължително, тъй като в този случай той трябва да освободи заетата памет.

Забележка: Ако се освобождава памет, заета от динамичен масив, чийто елементи са обекти на клас, трябва явно да се посочи дължината на масива. Тя е необходима за да се определи броят на извикванията на деструктора.

Създаване и разрушаване на обекти на класове

Съществуват два начина за създаване на обекти:

- чрез дефиниция;
- чрез функциите за динамично управление на паметта.

В първия случай обектът се създава при срещане на дефиницията (във функция или блок) и се разрушава при

завършване на изпълнението на функцията или при излизане от блока. Дефиницията може да бъде поставена където и да е в тялото на функцията или блока, като пред и след нея може да има изпълними оператори. Дефиницията, чрез която се създава обект, може да бъде допълнена с инициализация, която може да се основава на извикване на обикновен конструктор или на конструктора за присвояване.

Разрушаването на обекта е свързано с извикването на деструктора на класа, ако такъв явно е дефиниран или с извикването на "системния" деструктор (деструктора по подразбиране), ако в класа явно не е дефиниран деструктор.

Във втория случай създаването и разрушаването на обекти се управлява от програмиста. Създаването става с new, а разрушаването чрез delete. Операциите new се включват в конструкторите, а операциите delete – в деструктора на класа.

Пример:

```
rat *p = new rat(3,7); // търси в хийпа 8В, свързва адреса
                                    // им с p, извиква конструктора
                                       //  rat(9,13)  и инициализира паметта
(*p).print();
delete p;              //  извиква  деструктора,  след което
                    // разрушава обекта
...
```

В този случай деструкторът само регистрира присъствието си. Получаваме:

```
3/7
destruct number: 3/7
```

Оператори. Предефиниране на оператори

Езикът C++ има богат набор от оператори. В него са дадени също средства за предефиниране на оператори.

Всеки оператор се характеризира с:

- позиция на оператора спрямо аргументите му;

- приори тет;
- асоциативност.

Позицията на оператора спрямо аргументите му го определя като: префиксен (операторът е пред единствения му аргумент), инфиксен (операторът е между аргументите си) и постфиксен (операторът е след аргумента си).

Пример: Операторът / е инфиксен (4/8), операторът + е както инфиксен, така и префиксен (2+8, +78), а операторът ++ е както постфиксен, така и префиксен.

Приоритетът определя реда на изпълнение на операторите в операторен терм. Оператор с по-висок приоритет се изпълнява преди оператор с по-нисък приоритет.

Пример: Приоритетът на * и / е по-висок от този на + и -.

Асоциативността определя реда на изпълнение на оператори с еднакъв приоритет в операторен терм. В C++ има лявоасоциативни и дясноасоциативни оператори. Лявоасоциативните оператори се изпълняват отляво надясно, а дясноасоциативните – отдясно наляво.

В C++ не могат да се дефинират нови оператори, но всеки съществуващ едноаргументен или двуаргументен оператор с изключение на ::, ?:, ., *, # и ## може да бъде предефиниран от програмиста, стига поне един операнд на оператора да е обект на някакъв клас.

Например, възможно е да се предефинират операторите +, -, * и /, така че да могат да събират, изваждат, умножават и делят рационални числа. Тогава вместо sum(p, q), sub(p, q), mult(p, q) и quot(p, q) ще можем да пишем p+q, p-q, p*q и p/q, което безспорно е много по-удобно.

Предефинирането се осъществява чрез дефиниране на специален вид функции, наречени **операторни функции**. Последните имат синтаксис като на обикновените функции, но името им се състои от запазената дума operator, следвана от мнемоничното означение на предефинирания оператор. Когато предефинирането на оператор изисква достъп до компонентите на класове, обявени като private или protected, операторната дефиниция трябва да е член-функция или функция-приятел на

тези класове. Предефинираният оператор запазва всички характеристики на оригиналния.

Предефинирането може да стане по два начина:

- чрез функция-приятел;
- чрез член-функция.

Предефиниране чрез функция-приятел

В public частта на класа rat са включени декларациите на предефинираните оператори, предшествани от запазената дума friend:

```
friend rat operator+(rat const &, rat const &);
friend rat operator-(rat const &, rat const &);
friend rat operator*(rat const &, rat const &);
friend rat operator/(rat const &, rat const &);
```

Предефиниране чрез член-функция

В този случай първият аргумент на член-функцията трябва да е обект на класа и при дефинирането на операторната функция не се задава като параметър. Ако това не е така, операцията не може да се предефинира като член-функция.

Пример:

```
rat operator+(rat const &) const;
rat operator-(rat const &) const;
rat operator*(rat const &) const;
rat operator/(rat const &) const;
```

Наследяване. Производни класове.

Производните класове и наследяването са една от най-важните характеристики на обектно-ориентираното програмиране (ООП). Чрез механизма на наследяване от съществуващ клас се създава нов клас. Класът от който се създава се нарича **базов (основен) клас**, а този, който е създаден - **производен**. Понятията основен и производен клас са относителни, тъй като производен клас може да е основен за други класове, а основен – да е производен от други основни класове. Производният клас може да наследи компонентите на един или няколко базови класа. В първия случай наследяването се нарича **единично (просто)**, а във втория – **множествено**.

Дефинирането на производни класове е еквивалентно на конструирането на йерархии от класове. Ако множество от класове имат общи данни и методи, тези общи части могат да се обособят като основни класове, а всяка от останалите части да се дефинира като производен клас на съответния основен клас. Така се прави икономия на памет, тъй като се избягва многократното описание на едни и същи програмни фрагменти.

При конструирането на производни класове е достатъчно да се разполага само с обектните модули на основните класове, а не с техния програмен код. Това позволява да бъдат създавани библиотеки от класове, които да бъдат използвани при създаването на производни класове.

17.1 Дефиниране на производни класове

Подобно на обикновените, производните класове се дефинират като се *декларира класът* и се *дефинират неговите методи*.

Деклариране на производен клас

<декларация_на_производен_клас> ::=

class <име_на-производен_клас> :

[<атрибут_за_област>]$_{опц}$ <име_на_базов_клас>

{**,** [<атрибут_за_област>]$_{опц}$ <име_на_базов_клас>}$_{опц}$

{<декларация_на_компоненти>

};

<име_на-производен_клас> ::= <идентификатор>

<атрибут_за_област> ::= public | private | protected

<име_на_базов_клас> ::= <идентификатор>

Пред всяко име на базов клас *може* да се постави запазената дума public, private или protected. Нарича се **атрибут за област**, тъй като определя областта на наследените членове. Употребата на атрибутите за област е различна от тази за обявяване на секции в тялото на класа. Ако атрибут за област е пропуснат, подразбира се private. Атрибутът protected е включен в новите версии на езика и не се използва много често.

Примери:

1. Декларацията: class der : base1, base2, base3

{...

};

определя производен клас der с три основни класа base1, base2 и base3. Тъй като атрибутът за област е пропуснат и за трите базови класа, подразбира се private, т.е. декларацията е еквивалентна на

class der : private base1, private base2, private base3

{...

};

2. Декларацията: class der : public base1, base2, base3

{...

};

е еквивалентна на

class der : public base1, private base2, private base3

{...

};

3. Декларацията: class der : protected base1, base2, public base3

{...

};

е еквивалентна на

class der : protected base1, private base2, public base3

{...

};

Декларациите на компонентите на производен клас, а също дефинициите на неговите методи не се различават от съответните при обикновените класове.

Множеството от компонентите на един производен клас се състои от компонентите на неговите базови класове и компонентите, декларирани в самия производен клас. Оттук произлиза и терминът наследяване. Механизмът, чрез който производният клас получава компонентите на базовия, се нарича наследяване. Когато производният клас има няколко базови класа, той наследява компонентите на всеки от тях. Наследяването в този случай е множествено.

Процесът на наследяване се изразява в следното:

- наследяват се данните и методите на основния клас;
- получава се достъп до някои от наследените членове на основния клас;
- производният клас "познава" реализацията само на основния клас, от който произлиза;
- производният клас може да е основен за други класове.

Производният клас може да дефинира допълнително:

- свои член-данни;
- методи, аналогични на тези на основния клас, а също и нови.

Дефинираните в производния клас данни и методи се наричат **собствени**.

Производният клас е дефиниран след като вече е дефиниран базовият клас, от който той произлиза. Чрез него се разширява декларацията на съществуващ клас. Разширяемостта на класовете е една от важните характеристики на ООП.

17. 2 Наследяване и достъп до наследените компоненти

Ще напомним, че в рамките на един клас (без наследяване), protected частта има аналогична роля като тази на private частта. До компоненти от тип protected имат пряк достъп само член-функции и приятелски функции на класа.

Атрибутът за област на базовия клас в декларацията на производния клас (public, private или protected) управлява механизма на наследяване и определя какъв да бъде режимът на достъп до наследените членове. Таблицата от Фиг. 17.3 показва наследяванията на компоненти на основен клас в зависимост от атрибута за област.

Атрибут за област	Компонента на основен клас, определена като	Наследява се като
public	private public protected	Private public protected
Private	private public protected	Private private private
Protected	private public protected	Private protected protected

Фиг. 17.3 Наследявания на компоненти на основен клас в производен

От таблицата се вижда, че:

· Ако базовият клас е деклариран като public в производния клас, всички private, public и protected компоненти на базовия клас се наследяват съответно като private, public и protected компоненти на производния клас.

Пример: Ако

```
class base                                  class der1 : public base
{private: int b1;                           {private: int d1;
 protected: int b2;                          protected: int d2;
 public: int b3();                          public: int d3();
```

```
};                                                    };
```

можем да си мислим, че der1 е клас от вида:

```
    class der1
{private:
 int b1;
 int d1;
protected:
 int b2;
 int d2;
public:
 int b3();
 int d3();
};
```

· Ако базовият клас е деклариран като private в производния клас, всички негови компоненти се наследяват като private.

Пример: Ако

```
class base                                    class der2 : private base
{private: int b1;                          {private: int d1;
 protected: int b2;                         protected: int d2;
 public: int b3();                      public: int d3();
};                                                    };
```

можем да си мислим, че der2 е клас от вида:

```
    class der2
{private:
 int b1;
 int b2;
 int b3();
 int d1;
 protected:
 int d2;
 public:
 int d3();
};
```

· Ако базовият клас е деклариран като protected в производния клас, private компонентите му се наследяват като private, а public и protected – като protected.

Пример: Ако

```
class base                                    class der3 : protected base
{private: int b1;                             {private: int d1;
 protected: int b2;                            protected: int d2;
 public: int b3();                             public: int d3();
};                                                                    };
```

можем да си мислим, че der3 е клас от вида:

```
    class der3
{private:
```

```
 int b1;
 int d1;
protected:
 int b2;
 int d2;
 int b3();
public:
 int d3();
};
```

Наследените компоненти обаче се различават от декларираните в производния клас по правата за достъп. Производният клас има пряк достъп до компонентите, декларирани като public и protected, но няма пряк достъп до декларираните като private в базовия клас. Достъпът до private компонентите на базовия клас се извършва чрез неговия интерфейс.

Таблицата от Фиг. 17.4 показва прекия достъп на член-функции на производния клас (ПД) и външния достъп на производния клас (ВД) до компонентите на базовия клас.

компонента на базов клас	производен клас с атрибут public		производен клас с атрибут private		производен клас с атрибут protected	
	ПД	ВД	ПД	ВД	ПД	ВД
Public	да	да	да	не	да	не
Protected	да	не	да	не	да	не
Private	не	не	не	не	не	не

Фиг. 17.4 Достъп на член-функции и на обекти на производния клас до компонентите на базовия клас

Ще се спрем на някои от често срещаните случаи за достъп до членове на производен и основен клас, а също на достъпа на външни функции до наследен компонент. Ще изкажем и някои правила за достъп до компоненти на базови и производни класове, които ще подкрепим с още примери.

· Достъп до членове на основен клас чрез дефиниции на методи на производен клас

В сила са следните правила за достъп:

- Методите на производен клас (без значение на атрибута за област) нямат директен достъп до членовете от private–секцията на основния му клас

- В дефинициите на собствени методи на производния клас могат да се използват методите от секциите public и protected на основния му клас

- В дефинициите на собствени методи на производния клас може директно да се използват член-данните на секцията protected на основния му клас

· Достъп до методи чрез обекти на основния и производния клас

Обект на основен клас има пряк достъп до всички свои компоненти, обявени като public и няма пряк достъп до компонентите, обявени като private и protected. Обект на производен клас има пряк достъп до public компонентите на собствения си и компонентите на основния клас, наследени в производния клас като public.

· Достъп на основен клас до членове на производен клас и обратно

Методите на основния клас нямат достъп до членове на производен клас. Причината е, че когато основният клас се дефинира, не е ясно какви производни класове ще произхождат от него.

Производният клас също няма привилигирован достъп до членове на основния клас. От фиг. 17.3 се вижда, че производните класове нямат достъп до методите, обявени като private в основния клас.

Допустими са редица присвоявания между обекти на основния и производния клас. Ще ги разгледаме подробно в следващи части на главата. Засега ще отбележим само, че за реализирането им се извършват редица преобразувания.

· Достъп на функции приятели на производен клас до компоненти на основния му клас

Ще напомним, че функциите-приятели на клас не са елементи на класа, на който са приятели. Те са външни функции, получили привилигирован достъп до компонентите на класа.

Функциите приятели на производен клас имат същите права на достъп като член-функциите на производния клас. *Имат пряк достъп до всички компоненти, декларирани в класа и до public и protected компонентите на основния клас. Декларацията за приятелство не се наследява. Функция приятел на базовия клас не е приятел (освен ако не е декларирана като такава) на производния клас*.

11. Обектно ориентирано програмиране (на базата на C++ или Java): Основни принципи. Класове и обекти. Конструктори и деструктори. Оператори. Производни класове и наследяване.

Основните характеристики на обектно ориентираното програмиране са:

- Всичко е обект
- Една програма е съвкупснот от обекти, указващи си един на друг чрез изпраяане на съобщения какво да правят
- Обектите притежават собствена памет, суставена от други обекти.
- Всеки обект има тип
- Всички обекти от определен тип могат да получават едни и същи съобщения.

Обектно-ориентираното програмиране се е наложило като стандарт при почти всички съвременни езици за програмиране. То предоставя мощно средство за моделиране на обектите от реалния свят и взаимоотношенията между тях, позволява добро структуриране на програмния код и улеснява неговото преизползване. Благодарение на капсулацията на данните, както и на възможностите за наследяване и работа с абстрактни данни операции, ООП се е утвърдило като предпочитан подход при създаване на големи приложения и библиотеки.

Четирите основни принципа на ООП са капсулация на данните, абстракция, наследяване и полиморфизъм. Те са основните характеристики, които определят един език за програмиране като обектно-ориентиран.

Капсулация на данните

Основна концепция в ООП е обектът да се разглежда като "черна кутия" – използващите обекта "виждат" само атрибутите и операциите, които са присъщи на обекта от реалния свят, без да се интересуват от конкретната им реализация – клиентът на обекта трябва да знае само какво може да прави обектът, а не как го прави. В такъв смисъл "капсулация" означава скриване на ненужните детайли за обектите и откриване към външния свят само на важните техни характеристики и свойства. Обектите в ООП съдържат своите данни и средствата за тяхната обработката, капсулирани като едно цяло.

Абстракция

Абстракцията опростява сложната реалност чрез моделиране на класове подходящи за разрешаването на определен проблем. Тя представлява модел, изглед или някакво друго общо представяне на действителен обект. Абстракцията е разработване на програмен обект представящ обект от реалния свят, а капсулацията му скрива детайлите на имплеметацията. Абстракцията на данните налага ясно разделяне между абстрактните свойства на съответния тип и конкретните детайли на представянето. Идеята е, че промени в това представяне не трябва да имат влияние върху код използващ абстракцията. Нека разгледаме един прост пример: имаме клас Person съответно със ссвойства Height, Weight, Age и т.н. и действия, които може да се извършват с него: Run(), Sleep(), Cry(). Абстрактното представяне на Person се определя от свойствата които притежава и от действията, които могат да се извършват върху него и от информацията, която може да получим или предадем на него. Конкретната имплементация на действията и свойствата на обекта остава скрита от външния свят, който няма достъп до нея.

Наследяване

Ако един обект съдържа всички свойства и действия на друг, първият може да го наследи. По този начин наследеният обект освен собствените си атрибути и операции приема и тези на "родителя" си (базовия клас), като така се избягва повторното им дефиниране и се позволява създаването на йерархии от класове, моделиращи по естествен начин зависимостите от реалността.
За да изясним това понятие, ще си послужим с класическият в ООП пример за класа от обекти Animal, който представлява абстракция за множеството от всички животни. Всички обекти от този клас имат общи характеристики (например цвят и възраст) и обща функционалност, например операциите Eat и Sleep, докато за класът Dog, представляващ множеството от всички кучета, които също са животни, би могъл да предоставя операциите Eat, Sleep и Bark. Удачно е класът Dog да наследи Animal – тогава той ще съдържа описание само на собственото си действие Bark, докато тези на Eat и Sleep ще получи от базовия си клас. Чрез наследяването се постига специализация, или конкретизация на класовете, тъй като базовият клас представлява категория от обекти по-обща от тази на наследяващите го. Ако си послужим с горния пример, множествата на кучетата и котките са подмножества на множеството от всички животни.

Полиморфизъм

Полиморфизъм буквално означава приемането на различни форми от един обект. Нека е даден базов клас, представящ категория от обекти, които реализират общо действие, което се наследява от множество класове, описващи по-тесни категории. Въпреки, че те всички споделят това действие, те могат да го реализират по различен начин. Когато разполагаме с обекти от базовия клас, знаем че всички те реализират това действие, независимо на кой наследен клас принадлежат. Поради това можем да го използваме без да се интересуваме от конкретната му реализация. Например, ако класът Animal предоставя действието Talk и разгледаме наследяващите го класове Dog и Cat, всеки от тях го реализира по конкрете начин. И ако имаме животно, което издава звук и то е куче – ще лае, а ако е котка – ще мяучи. Полиморфизмът позволява унифицираното извършване на действие над различни обекти, които го реализират. В този случай издаването на звук от животно е полиморфно действие – такова, което се реализира по различен начин в различните наследници на базовия клас.

Класове

Класовете са нови типове данни, дефинирани от потребителя. Те могат да обогатяват възможностите на вече съществуващ тип или да представят напълно нов тип данни. Класовете са подобни на структурите, даже може да се каже, че в някои отношения са почти идентични. В C++ класът може да се разглежда като структура, на която са наложени някои ограничения по отношение на правата на достъп. Отначало ще разгледаме основното, което е приложимо както за класовете, така и за структурите. Основен принцип на процедурното програмиране е модулния. Програмата се разделя на “подходящи” взаимносвързани части (функции, модули), всяка от които се реализира чрез определени средства. Важен е обаче начинът, по който да става определянето на частите и връзките помежду им. Целта е, следващи промени в представянето на данните да не променят голям брой от модулите на програмата. Разсъждения в тази посока довеждат до подхода абстракция със структури от данни. При него методите за използване на данните се разделят от методите за тяхното конкретно представяне.

Програмите се конструират така, че да работят с “абстрактни данни” – данни с неуточнено представяне.

Дефиниране на класове

Класовете осигуряват механизми за създаване на напълно нови типове данни, които могат да бъдат интегрирани в езика, а също за обогатяване възможностите на вече съществуващи типове. Дефинирането на един клас се състои от две части:
декларация на класа и
дефиниция на неговите член-функции (методи).

Декларация на клас

Декларацията на клас се състои от заглавие и тяло. Заглавието започва със запазената дума class, следвано от името на класа. Тялото е заградено във фигурни скоби. След скобите стои знакът “;” или списък от обекти. В тялото на класа са декларирани членовете на класа (член-данни и член-функции) със съответните им нива на достъп.
За имената на класовете важат същите правила, които се прилагат за имената на всички останали типове и променливи. Също като при структурите името на класа може да бъде пропуснато.
Имената на членовете на класа са локални за него, т.е. в различни класове в рамките на една програма могат да се дефинират членове с еднакви имена. Член-данни от един и същ тип могат да се изредят, разделени със запетая и предшествани от типа им.

Пример:

```
class point
{private:
  double x, y;              // x и y са член-данни
 public:
  point(double, double);
  void read();
  int get_x() const;
  int get_y() const;
  void print() const;
}p=point(2,7), q(-2,3), r=q;
```

В тялото, някои декларации на членове могат да бъдат предшествани от спецификаторите на достъп private, public или protected. Областта на един спецификатор на достъп започва от спецификатора и продължава до следващия спецификатор. Подразбиращ се спецификатор за достъп е private. Един и същ спецификатор на достъп може да се използва повече от веднъж в декларация на клас.
Препоръчва се, ако секция public съществува, да бъде първа в декларацията, а секцията private да бъде последна в тялото на класа.

Достъпът до членовете на класовете може да се разгледа на следните две нива:
- По отношение на член-функциите в класа е в сила, че те имат достъп до всички членове на класа. При това не е необходимо тези компоненти да се предават като параметри. Този режим на достъп се нарича режим на пряк достъп.
- По отношение на функциите, които са външни за класа, режимът на достъп са определя от начина на деклариране на членовете.

Членовете на даден клас, декларирани като private (декларирани след запазената дума private) са видими (достъпни) само в рамките на класа. Външните функции нямат достъп до тях. По подразбиране членовете на класовете са private. Чрез използването на членове, обявени като private, се постига скриване на членове за външната за класа среда. Процесът на скриване се нарича още капсулиране на информацията.
Членовете на клас, които трябва да бъдат видими извън класа (да бъдат достъпни за функции, които не са методи на дадения клас) трябва да бъдат декларирани като public (декларирани след запазената дума public).
Дефиниране на методите на клас

След декларирането на клас, трябва да се дефинират неговите методи. Дефинициите са аналогични на дефинициите на функции, но името на метода се предшества от името на класа, на който принадлежи метода, следвано от оператора за принадлежност :: (Нарича се още оператор за област на действие). Такива имена се наричат пълни.

Дефиниция на метод на клас

Ще отбележим, че дефиницията на конструктор не започва с <тип>, а запазената дума const може да присъства само в дефинициите на функциите за достъп. Добрият стил на програмиране изисква използването на const в дефинициите на функциите за достъп и също в техните декларации.Ако се пренебрегне това изискване, могат да се създадат класове, които да не могат да се използват от други програмисти.
Обикновено дефинициите на методите са разположени веднага след декларирането на класа, на който те са членове. Възможно е обаче, дефинициите на методите на един клас да бъдат част от декларациите на този клас, т.е. в декларациите на член-функциите на класа могат да се зададат не само прототипите им, но и техните тела.

Пример: Класът rat може да бъде дефиниран и по следния начин:

```
class rat
{private:
 int numer;
 int denom;
public:
 rat()
 {numer = 0;
  denom = 1;
 }
 rat(int a, int b)
 {if (a == 0 || b==0)
 {numer = 0;
```

```
 denom = 1;
}
else
{int g = gcd(abs(a), abs(b));
 if (a>0 && b>0 || a<0 && b<0)
        {numer = abs(a)/g;
         denom = abs(b)/g;}
 else
 {numer = - abs(a)/g;
  denom = abs(b)/g;
 }
 }
}
void read()
{cout << "numer: ";
 cin >> numer;
 do
 {cout << "denom: ";
  cin >> denom;
 } while (denom == 0);
}
int get_numer() const
{return numer;
}
int get_denom() const
{return denom;
}
void print() const
{cout << numer << "/" << denom << endl;
}
};
```

В този случай обаче член-функциите се третират като вградени (inline) функции.
Често член-функциите се реализират като вградени функции. Това увеличава ефективността на програмата, използваща класа.

Ще отбележим също, че в тялото на дефиницията на член-функция явно не се указва обектът, върху който тя ще се приложи. Този обект участва неявно - чрез член-данните на класа. Заради това се нарича неявен параметър, а член-данните – абстрактни данни.Параметри, които участват явно в дефиницията на член-функция се наричат явни. Всяка член-функция има точно един неявен параметър и нула или повече явни.

Област на класовете

За разлика от функциите, класовете могат да се декларират на различни нива в програмата: глобално (ниво функция) и локално (вътре във функция или в тялото на клас).
Областта на глобално деклариран клас започва от декларацията и продължава до края на програмата. Примерите досега бяха с такива класове.
Ако клас е деклариран във функция, всички негови член-функции трябва да са вградени (inline). В противен случай ще се получат функции, дефинирани във функция, което не е възможно.

Пример:

```
void f(int i, int* p)
{int k;
 class CL
 {public:
        // всички методи са дефинирани в тялото на класа
  ...
  private:
  ...
 };
// тяло на функцията f
 CL x;
 ...
}
```

Областта на клас, дефиниран във функция, е функцията. Обектите на такъв клас са видими само в тялото на функцията.
Възможно е използването на обекти (в широкия смисъл на думата) с еднакви имена. В сила е правилото, че в областта си локалният обект скрива нелокалния.
Не е възможно в тялото на локално дефиниран клас да се използва функцията, в която класът е дефиниран.

Обекти

След като даден клас е дефиниран, могат да бъдат създавани негови екземпляри, които се наричат обекти. Връзката между клас и обект в езика C++ е подобна на връзката между тип данни и променлива, но за разлика от обикновените променливи, обектите се състоят от множество компоненти (член-данни и член-функции). На Фиг. 14.3 е даден синтаксисът на дефиниция на обект на клас.

Декларацията на клас не заделя памет за него. Памет се заделя едва при дефинирането на обект от класа. Дефиницията

```
rat p, q(2, 3), r = rat(3, 8);
```

заделя за обектите p, q и r по 8 байта ОП (по 4B за всяка от данните им numer и denom).

Достъпът до компонентите на обектите (ако е възможен) се осъществява чрез задаване на името на обекта и името на данната или метода, разделени с точка. Изключение от това правило правят конструкторите.

Пример:

```
rat p(1,2), q;
p.get_numer() // достъп до член-функцията get_numer() за обекта p
q.get_numer()          // достъп до член-функцията get_numer() за обекта q
```

Ще отбележим също, че на практика обектите p и q нямат свои копия на метода get_numer(). И двете обръщения се отнасят за един и същ метод, но при първото обръщение се работи с данните за обекта p, а при второто – с данните за обекта q.
При създаването на обекти на един клас кодът на методите на този клас не се копира във всеки обект, а се намира само на едно място в паметта.
Естествено възниква въпросът по какъв начин методите на един клас "разбират" за кой обект на този клас са били извикани. Отговорът на този въпрос дава указателят this. Всяка член-функция на клас поддържа допълнителен формален параметър - указател с име this и от тип <име_на_клас>*.
Обекти от един и същ клас могат да се присвояват един на друг. Присвояването може да е и на ниво инициализация .

Пример: Допустими са дефинициите

```
rat p, q(4,5), r=q;
p = q;
...
r = p;
```

При присвояването се копират всички член-данни на обекта. Така присвояването

```
r = p;
```

е еквивалентно на

```
r.numer = p.numer;
r.denom = p.denom;
```

Конструктори

Създаването на обекти е свързано с отделяне на памет, запомняне на текущо състояние, задаване на начални стойности и др. дейности, които се наричат инициализация на обекта. В езика C++ тези дейности се изпълняват от специален вид член-функции на класовете – конструкторите.

Дефиниране на конструктор

Ще напомним, че конструкторът е член-функция, която притежава повечето характеристики на другите член-функции, но има и редица особености, като:
Името на конструктора съвпада с името на класа.
Типът на резултата е указателят this и явно не се указва.
Изпълнява се автоматично при създаване на обекти.
Не може да се извиква явно (обръщение от вида r.rat(1,4) е недопустимо).

Освен това в един клас може явно да не е дефиниран конструктор, но може да са дефинирани и няколко конструктора.
Типична дефиниция на конструктор е дадена в следващия пример.

Пример:

```
class CL
{public:
  CL(int, int, int);
  void print();
  ...
 private:
  int a, b, c;
};
CL::CL(int x, int y, int z) // конструктор с три параметъра
{a = x;
 b = y;
 c = z;
}
```

Класът CL в този пример има един триаргументен конструктор. При създаване на обект на класа, този конструктор ще се изпълнява автоматично, стига да е коректно извикан, в резултат на което член-данните a, b и c на създадения обект ще се свържат със стойностите, които се подадат като фактически параметри на конструктора.

Забелязваме, че е възможно член_данна да се свърже с инициализираща стойност в заглавието на конструктора.

Пример: Конструкторът на класа CL може да се дефинира и по следния начин

```
CL::CL(int x, int y, int z): a(x), b(y), c(z)
{}
```

Ще отбележим, че не е задължително всички член-данни да са инициализирани само пред тялото или само вътре в тялото на конструктора.

Пример: Допустима е дефиницията

```
CL::CL(int x, int y, int z): a(x)
    {b = y;
     c = z;
    }
```

Смисълът на обобщената синтактична конструкция е, че инициализацията на член-данните в заглавието предшества изпълнението на тялото на конструктора. Това я прави изключително полезна. Използването й увеличава ефективността на класа.

Предефинирани конструктори

Обект (в общия смисъл) е предефиниран, ако за него има няколко различни дефиниции, задаващи различни негови интерпретации. За да бъдат използвани такива конструкции е необходим критерии, по който те да се различат.
В рамките на една програма може да се извършва предефиниране на функции. Възможно е:

а) да се използват функции с едно и също име с различни области на видимост
В този случай не възниква проблем с различаването.
б) да се използват функции с едно и също име в една и съща област на видимост
В един клас може да са дефинирани няколко конструктора. Всички те имат едно име (името на класа), но трябва да се различават по броя и/или типа на параметрите си. Наричат се предефинирани конструктори. При създаването на обект на класа се изпълнява само един от тях. Определя се съгласно критерия за най-добро съвпадане.
Пример: В класа rat дефинирахме два конструктора rat() и rat(int, int), които се различават по броя на параметрите си.

Подразбиращ се конструктор

В клас може явно да е дефиниран, но може и да не е дефиниран конструктор. Ако явно не е дефиниран конструктор, автоматично се създава един т. нар. подразбиращ се конструктор. Този конструктор реализира множество от действия като: заделяне на памет за данните на обект, инициализиране на някои системни променливи и др.

Подразбиращият се конструктор може да бъде предефиниран. За целта е необходимо в класа да бъде дефиниран конструктор без параметри.

Пример: В класа rat, подразбиращият се конструктор беше предефиниран от конструктора

```
rat::rat()
{numer = 0;
 denom = 1;
}
```

Конструктори с подразбиращи се параметри

Функциите в езика C++ могат да имат подразбиращи се параметри. За тези параметри се задават подразбиращи се стойности, които се използват само ако при извикването на функцията не бъде зададена стойност за съответния параметър.
Задаването на подразбираща се стойност се извършва чрез задаване на конкретна стойност в прототипа на функцията или в нейната дефиниция.
При използване на подразбиращи се параметри, важна роля играе редът на параметрите. Прието е, че ако параметър на функция е подразбиращ се, всички параметри след него също са подразбиращи се.

Всичко казано за подразбиращите се параметри на функции се отнася и за конструкторите.
Ако променим дефиницията на класа rat по следния начин:

```
class rat
{private:
  int numer;
  int denom;
 public:
  rat(int=0, int=1);
  void read();
```

```
    int get_numer() const;
    int get_denom() const;
    void print() const;
};
```

където конструкторът rat(int, int); се дефинира по същия начин, са допустими следните дефиниции на обекти:

```
rat p,                          // p се инициализира с 0/1
      q = rat(),                // q се инициализира с 0/1
      r = rat(5),               // r се инициализира с 5/1
      s = rat(13,21),           // s се инициализира с 13/21
      t(2);                     // t се инициализира с 2/1
```

Конструктори за присвояване и копиране

Инициализацията на новосъздаден обект на даден клас може да зависи от друг обект на същия клас. За да се укаже такава зависимост се използва знакът за присвояване или кръгли скоби.
Пример: Нека
rat p = rat(1,4);
Чрез еквивалентните конструкции

```
rat q = p;
rat q(p);
```

се създава обектът q от клас rat, като инициализацията на q зависи от p. Тази инициализация се създава от специален конструктор, наречен конструктор за присвояване.
Конструкторът за присвояване е конструктор, поддържащ формален параметър от тип <име_на_клас> const &.
- Ако в един клас явно не е дефиниран конструктор за присвояване, компилаторът автоматично създава такъв, в момента когато новосъздаден обект се инициализира с обекта, намиращ се от дясната страна на знака за присвояване или в кръглите скоби. Този конструктор за присвояване се нарича конструктор за копиране.

Дефиницията rat p=q създава нов обект p (без викане на конструктор), в който се копират съответните стойности на обекта q. Това е резултат от изпълнението на обръщението към rat(&p, q).
- Ако в класа е дефиниран конструктор за присвояване, компилаторът го използва.

Предефиниране на служебния конструктор за копиране се налага когато обектите имат член-данни, указващи динамична памет.

Освен в горните случаи, конструкторът за присвояване (копиране) се използва и при предаване на обект като аргумент на функция, когато предаването е по стойност, а също при връщане на обект като резултат от изпълнение на функция.

Указатели към обекти на класове

Дефинират се по същия начин както се дефинират указатели към променливи от стандартните типове данни.

Пример:

rat p;

rat * ptr = &p;

В резултат ще се отделят 4В ОП, които ще се именуват с ptr и ще се инициализират с адреса на обекта p.

Достъпът до компонентите на рационалното число, сочено от ptr, се осъществява по общоприетия начин:

(*ptr).get_numer()

(*ptr).get_denom()

Синтактичната конструкция (*ptr). е еквивалентна на ptr ->. Така горните обръщения могат да се запишат и по следния начин:

ptr -> get_numer()

ptr -> get_denom()

Масиви и обекти

Елементите на масив могат да са обекти, но разбира се от един и същ клас. Дефинират се по общоприетия начин (Фиг. 14.7).

Пример:

rat table[10];

определя масив от 10 обекта от клас rat.

Достъпът до елементите на масива е пряк и се осъществява по стандартния начин – чрез индексирани променливи.

Пример: Чрез индексираните променливи

table[0], table[1], ..., table[9]

се осъществява достъп до първия, втория и т.н. до десетия елемент на table.

Тъй като table[i] (i = 0, 1, ..., 9) са обекти, възможни са следните обръщения към техни компоненти:

table[i].read(); // въвежда стойност на table[i]

table[i].print(); // извежда стойността на table[i]

table[i].get_numer(); // намира числителя на table[i]

table[i].get_denom(); // намира знаменателя на table[i].

Връзката между масиви и указатели е в сила и в случая когато елементите на масива са обекти. Името на масива е указател към първия му елемент, т.е. ако

```
rat * p = table;          // p сочи към table[0]
                                                    // т.е. p==&table[0]
*(p+i) == table[i], i = 0, 1,...,9
```

Тогава

```
(*(p+i)).print();                  // е еквивалентно на table[i].print();
```

Масивът може да е член-данна на клас.
Конструкторите (в частност конструкторът по подразбиране) играят важна роля при дефинирането и инициализирането на масиви от обекти. Масив от обекти, дефиниран в програма, се инициализира по два начина:
неявно (чрез извикване на системния конструктор по подразбиране за всеки обект – елемент на масива);
явно (чрез инициализиращ списък).

Примери:

а) Класът

```
const NUM = 5;
class student
{public:
 void read_student();
 void print_student() const;
 bool is_better(student const &) const;
 double average() const;
 private:
 int facnom;
 char name[26];
 double marks[NUM];
```

```
};
```

няма явно дефиниран конструктор. Дефиницията
student table[30];
на масива table от 30 обекта от клас student е правилна.
Инициализацията се осъществява чрез извикване на "системния" конструктор по подразбиране за всеки обект – елемент на масива.
б) Класът rat, дефиниран по-долу

```
class rat

{private:

 int numer;

 int denom;

public:

 rat(int=0, int=1);

 void read();

 int get_numer() const;

 int get_denom() const;

 void print() const;

};
```

притежава явно дефиниран конструктор с два подразбиращи се параметъра. В този случай са допустими дефиниции от вида:

```
rat x[10]; // x[i] се инициализира с 0/1, за всяко i=0,1,...9.

rat x[10] = {1,2,3,4,5,6,7,8,9,10}; //x[i] == i/1

rat x[10] = {rat(1,21),rat(2),rat(3, 5),4,5,6,7,8,9,10};

// x[0] == 1/21; x[1] == 2/1; x[2] == 3/5, x[3]== 4/1, ...
```

Ако променим конструктора на класа rat от

```
rat(int=0, int=1);
```

в

```
rat(int, int);
```

т.е. без подразбиращи се параметри и трите дефиниции от по-горе ще съобщят за грешка. Единствено допустима дефиниция на x[10] е с инициализация с 10 обръщения към двуаргументния конструктор rat с явно указани два аргумента.

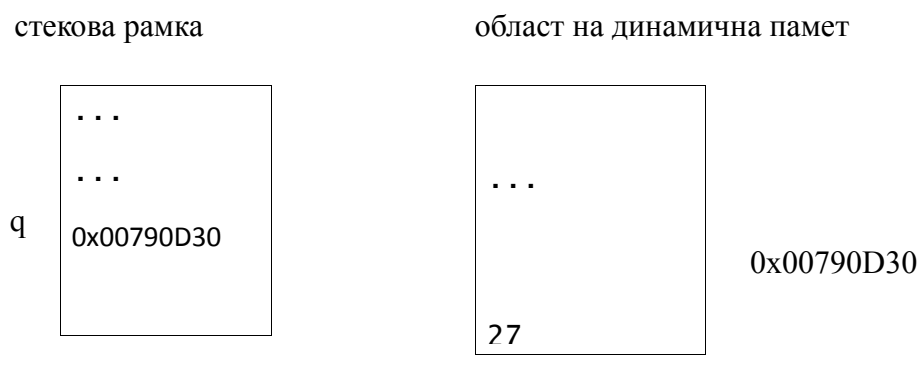
Динамични обекти

Вече разгледахме в най-общ план разпределението на ОП по време на изпълнението на програма. От Фиг. 8.1 на Глава 8 се вижда, че всяка програма има две "места" за памет: програмен стек (стек) и област за динамичните данни (динамична памет, хийп или куп). Стекът е област за временно съхранение на информация. Той е кратковременна памет. C++ използва стека основно за реализиране на обръщения към функции. Всяко обръщение към функция предизвиква конструиране на нова стекова рамка, която се установява на върха на стека. По такъв начин когато функция A извика функция B, която от своя страна вика функция C, стекът нараства. Когато пък всяка от тези функции завършва, стековите рамки на тези функции автоматично се разрушава. Така стекът се свива.
Хийпът е по-постоянна област за съхранение на данни. Той е един вид дълготрайна памет. Особеност на тази памет е, че тя не се свързва с имена на променливи. С разположените в нея обекти се работи косвено – чрез указатели. Обикновено се използва при работа с т. нар. динамични структури от данни. Динамичните данни са такива обекти (в широкия смисъл на думата), чийто брой не е известен в момента на проектирането на програмата. Те се създават и разрушават по време на изпълнението на програмата. След разрушаването им, заетата от тях памет се освобождава и може да се използва отново. Така паметта се използва по-ефективно. Създаването и разрушаването на динамични обекти в C++ се осъществява чрез операторите new и delete. Извикването на new заделя в хийпа необходимата памет и връща указател към нея. Този указател може да се съхрани в някаква променлива и да се пази докато е необходимо. За разлика от стека, заделянето на памет в хийпа е явно – чрез new. Освобождаването на паметта от хийпа също става явно, чрез delete. Всяко извикване на new трябва да бъде балансирано чрез извикване на delete. Последното се налага, тъй като за разлика от стека, хийпът не се изчиства автоматично. В C++ няма система за "събиране на боклуци" (автоматично премахване на обекти, които вече не са необходими). Затова трябва явно да бъдат изтрити създадените в хийпа обекти.

Примери:

a) int* q = new int(2+5*5);

отделя (ако е възможно) 4B памет в хийпа, инициализира я с 27 - стойността на израза 2+5*5 и свързва q с адреса на тази памет, т.е.



Заделянето на памет по време на компилация се нарича статично заделяне на памет, заделянето на памет по време на изпълнение на програмата - динамично разпределение на паметта.

Под период на активност на една променлива се разбира частта от времето за изпълнение на програмата, през което променливата е свързана с конкретно място в паметта. Паметта за глобалните променливи се заделя в началото и остава свързана с тях до завършването на изпълнението на програмата. Паметта за локалните променливи се заделя при влизане в локалната област и се освобождава при напускането й. Паметта на динамичните променливи се заделя от оператора new. Заделената по този начин памет остава свързана със съответната променлива докато не се освободи явно от програмиста. Явното освобождаване на динамична променлива се осъществява чрез оператора delete, приложен към указателя, който адресира съответната променлива.

Ако в хийпа е заделена памет, след което тази памет не е освободена чрез delete, се получава загуба на памет. Парчето памет, което не е освободено, е като остров в хийпа, заемащо пространство, което иначе би могло да се използва за други цели.

За да се разруши масив, създаден чрез new по следния начин:

```
int* arr = new int[5];
```

трябва да се запише:

```
delete [] arr;
```

Ако обаче масивът съдържа в себе си указатели, първо трябва да бъде обходен и да бъде извикан операторът delete за всеки негов елемент. Едва след това може да се извика delete за масива по показания по-горе начин. Операторът delete трябва да се използва само за освобождаване на динамична памет, заделена с new. В противен случай действието му е непредсказуемо.

Динамичната памет не е неограничена. Тя може да се изчерпи по време на изпълнение на програмата. Неуспешното завършване на delete ускорява изчерпването. Ако наличната в момента динамична памет е недостатъчна, new връща нулев указател и програмата (функцията) няма да работи. Затова се препоръчва след всяко извикване на new да се прави проверка за успешността й.

Чрез оператора new могат да се създават т. нар. динамични масиви – масиви с променлива дължина. Динамичните масиви се създават в динамичната памет. Следващата програма илюстрира този процес.

(за примерни задачи вж. учебника)

Методите на класовете също могат да използват динамична памет, която се заделя и освобождава по време на изпълнението им, чрез операторите new и delete.

Ще отбележим също, че заделената от член-функциите динамична памет не се освобождава автоматично при разрушаване на обектите на класове. Освобождаването на тази памет трябва да стане явно чрез оператора delete, който трябва да се изпълни преди разрушаването на обекта. Този процес може да бъде автоматизиран чрез използване на специален вид методи, наречени деструктори.

Деструктори

Разрушаването на обекти на класове в някои случаи е свързано с извършване на определени действия, които се наричат заключителни. Най-често тези действия са свързани с освобождаване на заделена преди това динамична памет, възстановяване на състояние на програмата и др. Ефектът от заключителните действия е противоположен на ефекта на инициализацията. Естествено е да се даде възможност заключителните действия да се извършат автоматично при разрушаването на обекта. Това се осъществява от деструкторите.

Деструкторът е член-функция, която се извиква при:

- разрушаването на обект чрез оператора delete,
- излизане от блок, в който е бил създаден обект от класа.

Един клас може да има явно дефиниран точно един деструктор. Името му съвпада с името на класа, предшествано от символа ‘~’ (тилда), типът му е void и явно не се задава в заглавието. Деструкторът няма формални параметри.

Забележка: Използването на явно дефинирани деструктори не винаги е належащо, тъй като всички член-променливи се разрушават при разрушаването на обекта и без използването на деструктор. Ако конструкторът или някоя член-функция реализира динамично заделяне на памет за някоя член-данна, използването на деструктор е задължително, тъй като в този случай той трябва да освободи заетата памет.
Забележка: Ако се освобождава памет, заета от динамичен масив, чийто елементи са обекти на клас, трябва явно да се посочи дължината на масива. Тя е необходима за да се определи броят на извикванията на деструктора.

Създаване и разрушаване на обекти на класове

Съществуват два начина за създаване на обекти:
чрез дефиниция;
чрез функциите за динамично управление на паметта.
В първия случай обектът се създава при срещане на дефиницията (във функция или блок) и се разрушава при завършване на изпълнението на функцията или при излизане от блока. Дефиницията може да бъде поставена където и да е в тялото на функцията или блока, като пред и след нея може да има изпълними оператори. Дефиницията, чрез която се създава обект, може да бъде допълнена с инициализация, която може да се основава на извикване на обикновен конструктор или на конструктора за присвояване.
Разрушаването на обекта е свързано с извикването на деструктора на класа, ако такъв явно е дефиниран или с извикването на "системния" деструктор (деструктора по подразбиране), ако в класа явно не е дефиниран деструктор.
Във втория случай създаването и разрушаването на обекти се управлява от програмиста. Създаването става с new, а разрушаването чрез delete. Операциите new се включват в конструкторите, а операциите delete – в деструктора на класа.

Пример:

```
rat *p = new rat(3,7); // търси в хийпа 8B, свързва адреса
                                                    // им с p, извиква конструктора
                                                    // rat(9,13) и инициализира
паметта
(*p).print();
delete p;                                   // извиква деструктора, след което
                                            // разрушава обекта
...
```

В този случай деструкторът само регистрира присъствието си. Получаваме:

```
3/7
destruct number: 3/7
```

Оператори. Предефиниране на оператори

Езикът C++ има богат набор от оператори. В него са дадени също средства за предефиниране на оператори.

Всеки оператор се характеризира с:
позиция на оператора спрямо аргументите му;
приоритет;
асоциативност.

Позицията на оператора спрямо аргументите му го определя като: префиксен (операторът е пред единствения му аргумент), инфиксен (операторът е между аргументите си) и постфиксен (операторът е след аргумента си).
Пример: Операторът / е инфиксен (4/8), операторът + е както инфиксен, така и префиксен (2+8, +78), а операторът ++ е както постфиксен, така и префиксен.
Приоритетът определя реда на изпълнение на операторите в операторен терм. Оператор с по-висок приоритет се изпълнява преди оператор с по-нисък приоритет.
Пример: Приоритетът на * и / е по-висок от този на + и -.
Асоциативността определя реда на изпълнение на оператори с еднакъв приоритет в операторен терм. В C++ има лявоасоциативни и дясноасоциативни оператори. Лявоасоциативните оператори се изпълняват отляво надясно, а дясноасоциативните – отдясно наляво.
В C++ не могат да се дефинират нови оператори, но всеки съществуващ едноаргументен или двуаргументен оператор с изключение на ::, ?:, ., *, # и ## може да бъде предефиниран от програмиста, стига поне един операнд на оператора да е обект на някакъв клас.
Например, възможно е да се предефинират операторите +, -, * и /, така че да могат да събират, изваждат, умножават и делят рационални числа. Тогава вместо sum(p, q), sub(p, q), mult(p, q) и quot(p, q) ще можем да пишем p+q, p-q, p*q и p/q, което безспорно е много по-удобно.
Предефинирането се осъществява чрез дефиниране на специален вид функции, наречени операторни функции. Последните имат синтаксис като на обикновените функции, но името им се състои от запазената дума operator, следвана от мнемоничното означение на предефинирания оператор. Когато предефинирането на оператор изисква достъп до компонентите на класове, обявени като private или protected, операторната дефиниция трябва да е член-функция или функция-приятел на тези класове. Предефинираният оператор запазва всички характеристики на оригиналния.
Предефинирането може да стане по два начина:
чрез функция–приятел;
чрез член-функция.

Предефиниране чрез функция-приятел

В public частта на класа rat са включени декларациите на предефинираните оператори, предшествани от запазената дума friend:

```
friend rat operator+(rat const &, rat const &);
friend rat operator-(rat const &, rat const &);
friend rat operator*(rat const &, rat const &);
friend rat operator/(rat const &, rat const &);
```

Предефиниране чрез член-функция

В този случай първият аргумент на член-функцията трябва да е обект на класа и при дефинирането на операторната функция не се задава като параметър. Ако това не е така, операцията не може да се предефинира като член-функция.

Пример:

```
rat operator+(rat const &) const;
rat operator-(rat const &) const;
rat operator*(rat const &) const;
rat operator/(rat const &) const;
```

Наследяване. Производни класове.

Производните класове и наследяването са една от най-важните характеристики на обектно-ориентираното програмиране (ООП). Чрез механизма на наследяване от съществуващ клас се създава нов клас. Класът от който се създава се нарича **базов (основен) клас**, а този, който е създаден - **производен**. Понятията основен и производен клас са относителни, тъй като производен клас може да е основен за други класове, а основен – да е производен от други основни класове. Производният клас може да наследи компонентите на един или няколко базови класа. В първия случай наследяването се нарича **единично (просто)**, а във втория – **множествено**.

Дефинирането на производни класове е еквивалентно на конструирането на йерархии от класове. Ако множество от класове имат общи данни и методи, тези общи части могат да се обособят като основни класове, а всяка от останалите части да се дефинира като производен клас на съответния основен клас. Така се прави икономия на памет, тъй като се избягва многократното описание на едни и същи програмни фрагменти.

При конструирането на производни класове е достатъчно да се разполага само с обектните модули на основните класове, а не с техния програмен код. Това позволява да бъдат създавани библиотеки от класове, които да бъдат използвани при създаването на производни класове.

17.1 Дефиниране на производни класове

Подобно на обикновените, производните класове се дефинират като се *декларира класът* и се *дефинират неговите методи*.

Деклариране на производен клас

```
<декларация_на_производен_клас> ::=

class <име_на-производен_клас> :

        [<атрибут_за_област>]опц <име_на_базов_клас>

        {, [<атрибут_за_област>]опц <име_на_базов_клас>}опц

{<декларация_на_компоненти>

};

<име_на-производен_клас> ::= <идентификатор>

<атрибут_за_област> ::= public | private | protected

<име_на_базов_клас> ::= <идентификатор>
```

Пред всяко име на базов клас *може* да се постави запазената дума public, private или protected. Нарича се **атрибут за област**, тъй като определя областта на наследените членове. Употребата на атрибутите за област е различна от тази за обявяване на секции в тялото на класа. Ако атрибут за област е пропуснат, подразбира се private. Атрибутът protected е включен в новите версии на езика и не се използва много често.

Примери:

1. Декларацията: class der : base1, base2, base3

{...

};

определя производен клас der с три основни класа base1, base2 и base3. Тъй като атрибутът за област е пропуснат и за трите базови класа, подразбира се private, т.е. декларацията е еквивалентна на

class der : private base1, private base2, private base3

{...

};

2. Декларацията: class der : public base1, base2, base3

{...

};

е еквивалентна на

class der : public base1, private base2, private base3

{...

};

3. Декларацията: class der : protected base1, base2, public base3

{...

};

е еквивалентна на

class der : protected base1, private base2, public base3

{...

};

Декларациите на компонентите на производен клас, а също дефинициите на неговите методи не се различават от съответните при обикновените класове.

Множеството от компонентите на един производен клас се състои от компонентите на неговите базови класове и компонентите, декларирани в самия производен клас. Оттук произлиза и терминът наследяване. Механизмът, чрез който производният клас получава компонентите на базовия, се нарича наследяване. Когато производният клас има няколко базови класа, той наследява компонентите на всеки от тях. Наследяването в този случай е множествено.

Процесът на наследяване се изразява в следното:

- наследяват се данните и методите на основния клас;
- получава се достъп до някои от наследените членове на основния клас;
- производният клас "познава" реализацията само на основния клас, от който произлиза;
- производният клас може да е основен за други класове.

Производният клас може да дефинира допълнително:

- свои член-данни;
- методи, аналогични на тези на основния клас, а също и нови.

Дефинираните в производния клас данни и методи се наричат **собствени**.

Производният клас е дефиниран след като вече е дефиниран базовият клас, от който той произлиза. Чрез него се разширява декларацията на съществуващ клас. Разширяемостта на класовете е една от важните характеристики на ООП.

17. 2 Наследяване и достъп до наследените компоненти

Ще напомним, че в рамките на един клас (без наследяване), protected частта има аналогична роля като тази на private частта. До компоненти от тип protected имат пряк достъп само член-функции и приятелски функции на класа.

Атрибутът за област на базовия клас в декларацията на производния клас (public, private или protected) управлява механизма на наследяване и определя какъв да бъде режимът на достъп до наследените членове. Таблицата от Фиг. 17.3 показва наследяванията на компоненти на основен клас в зависимост от атрибута за област.

Атрибут за област	**Компонента на основен клас, определена като**	**Наследява се като**
public	private public protected	Private public protected
Private	private public protected	Private private private
Protected	private public protected	Private protected protected

Фиг. 17.3 Наследявания на компоненти на основен клас в производен

От таблицата се вижда, че:

· Ако базовият клас е деклариран като public в производния клас, всички private, public и protected компоненти на базовия клас се наследяват съответно като private, public и protected компоненти на производния клас.

Пример: Ако

```
class base                                  class der1 : public base
{private: int b1;                           {private: int d1;
 protected: int b2;                          protected: int d2;
 public: int b3();                          public: int d3();
};                                                                  };
```

можем да си мислим, че der1 е клас от вида:

```
    class der1
{private:
 int b1;
 int d1;
protected:
 int b2;
 int d2;
public:
 int b3();
 int d3();
};
```

· Ако базовият клас е деклариран като private в производния клас, всички негови компоненти се наследяват като private.

Пример: Ако

```
class base                                  class der2 : private base
{private: int b1;                           {private: int d1;
 protected: int b2;                          protected: int d2;
 public: int b3();                          public: int d3();
};                                                                  };
```

можем да си мислим, че der2 е клас от вида:

```
    class der2
{private:
 int b1;
 int b2;
 int b3();
 int d1;
 protected:
 int d2;
 public:
 int d3();
};
```

· Ако базовият клас е деклариран като protected в производния клас, private компонентите му се наследяват като private, а public и protected – като protected.

Пример: Ако

```
class base                                  class der3 : protected base
```

```
{private: int b1;                    {private: int d1;
 protected: int b2;                   protected: int d2;
 public: int b3();                  public: int d3();
};                                                          };
```

можем да си мислим, че der3 е клас от вида:

```
   class der3
{private:
 int b1;
 int d1;
 protected:
 int b2;
 int d2;
 int b3();
 public:
 int d3();
};
```

Наследените компоненти обаче се различават от декларираните в производния клас по правата за достъп. Производният клас има пряк достъп до компонентите, декларирани като public и protected, но няма пряк достъп до декларираните като private в базовия клас. Достъпът до private компонентите на базовия клас се извършва чрез неговия интерфейс.

Таблицата от Фиг. 17.4 показва прекия достъп на член-функции на производния клас (ПД) и външния достъп на производния клас (ВД) до компонентите на базовия клас.

компонента на базов клас	производен клас с атрибут public ПД ВД	производен клас с атрибут private ПД ВД	производен клас с атрибут protected ПД ВД
Public	да да	да не	да не
Protected	да не	да не	да не
Private	не не	не не	не не

Фиг. 17.4 Достъп на член-функции и на обекти на производния клас до компонентите на базовия клас

Ще се спрем на някои от често срещаните случаи за достъп до членове на производен и основен клас, а също на достъпа на външни функции до наследен компонент. Ще изкажем и някои правила за достъп до компоненти на базови и производни класове, които ще подкрепим с още примери.

· Достъп до членове на основен клас чрез дефиниции на методи на производен клас

В сила са следните правила за достъп:

- Методите на производен клас (без значение на атрибута за област) нямат директен достъп до членовете от private–секцията на основния му клас

- В дефинициите на собствени методи на производния клас могат да се използват методите от секциите public и protected на основния му клас

- В дефинициите на собствени методи на производния клас може директно да се използват член-данните на секцията protected на основния му клас

· Достъп до методи чрез обекти на основния и производния клас

Обект на основен клас има пряк достъп до всички свои компоненти, обявени като public и няма пряк достъп до компонентите, обявени като private и protected. Обект на производен клас има пряк достъп до public компонентите на собствения си и компонентите на основния клас, наследени в производния клас като public.

· Достъп на основен клас до членове на производен клас и обратно

Методите на основния клас нямат достъп до членове на производен клас. Причината е, че когато основният клас се дефинира, не е ясно какви производни класове ще произхождат от него.

Производният клас също няма привилигирован достъп до членове на основния клас. От фиг. 17.3 се вижда, че производните класове нямат достъп до методите, обявени като private в основния клас.

Допустими са редица присвоявания между обекти на основния и производния клас. Ще ги разгледаме подробно в следващи части на главата. Засега ще отбележим само, че за реализирането им се извършват редица преобразувания.

· Достъп на функции приятели на производен клас до компоненти на основния му клас

Ще напомним, че функциите-приятели на клас не са елементи на класа, на който са приятели. Те са външни функции, получили привилигирован достъп до компонентите на класа.

Функциите приятели на производен клас имат същите права на достъп като член-функциите на производния клас. *Имат пряк достъп до всички компоненти, декларирани в класа и до public и protected компонентите на основния клас. Декларацията за приятелство не се наследява. Функция приятел на базовия клас не е приятел (освен ако не е декларирана като такава) на производния клас*.

Структури от данни и алгоритми. Анализ на алгоритми. Абстрактни типове от данни. Стек, опашка, списък, дърво. Сортиране.

1. **Основи на анализа на алгоритми: Асимптотична нотация на сложността. Основни рекурентни формули. Примери за анализ на алгоритми.**

Бързодействието на даден алгоритъм може да зависи от скоростта на даденият компютър, върху който се изпълнява. Различаваме относително бързодействие (върху даден компютър) и абсолютно бързодействие (върху различни компютри). При формалното оценяване на сложността на алгоритмите се интересуваме от поведението им при N (размер на входа), клонящо към безкрайност, като игнорираме машинно зависимите константи, тоест ще ни интересува как се държи алгоритъма при достатъчно голям размер N на входните данни, без да ни интересува машината, на която се изпълнява алгоритъма. Сложността на алгоритъм ще измерваме спрямо времето или паметта, необходима на алгоритъма за да реши даден проблем спрямо размера на подаденият вход (съответно сложност по време и сложност по памет).

Разпознаваме три вида анализ:

- Най-лош случай: (обикновено)
  T(n) = максималното време за алгоритъм върху всеки вход с размер n.

- Средно време: (понякога)
  T(n) = очакваното време за алгоритъм за всеки вход с размер n.
  Необходимо е предположение за статистическото разпределение на входа.

- Най-добър случай: (погрешно)
  Измама с бавен алгоритъм, който работи бързо върху някакъв вход.

За анализа на алгоритми ще използваме следните 'асимптотична нотации':

- **O-нотация (горна граница):**

Дефиниция: $O(g(n))= \{ f(n)$: за които съществуват константи $c> 0, n_0> 0$, такива че $0 \le f(n) \le cg(n)$ за всяко $n \ge n_0\}$

Записваме $f(n) = O(g(n))$ ако съществуват константи $c> 0, n_0> 0$, такива че $0 \le f(n) \le cg(n)$ за всяко $n \ge n_0$.

Пример: $2n^2= O(n^3)$:$(c= 1, n_0= 2)$

- **Ω-нотация (долна граница)**

Дефиниция: $\Omega(g(n))= \{ f(n)$: за които съществуват константи $c > 0, n_0> 0$, такива че $0 \le cg(n) \le f(n)$ за всяко $n \ge n_0\}$)

Записваме $f(n) = \Omega(g(n))$ ако съществуват константи $c> 0, n_0> 0$, такива че $0 \le cg(n) \le f(n)$ за всяко $n \ge n_0$.

Пример: n $\in \Omega(\lg n)$ $(c=1, n_0= 16)$

- **Θ-нотация (tight bounds)**

Дефиниция: : $\Theta(g(n))= \{ f(n)$: за които съществуват константи $c_1 > 0, c_2> 0, n_0> 0$, такива че $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ за всяко $n \ge n_0\}$)

От дефиницията следва, че $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$.

Пример: 12n2 - $2n \in \Theta(n2)$

- **o-нотация**

Дефиниция: $o(g(n))= \{ f(n)$: ако съществуват константи $c> 0$, $n_0> 0$, такива че $0 \leq f(n) < cg(n)$ за всяко $n \geq n_0\}$

- **ω-нотация**

Дефиниция: $\omega(g(n))= \{ f(n)$: ако съществуват константи $c> 0$, $n_0> 0$, такива че $0 \leq cg(n) < f(n)$ за всяко $n \geq n_0\}$)

O-нотацията и Ω-нотацията са като ≤ и ≥.
o- нотацията и ω- нотацията са като < и >.

Основни рекурентни формули

Формула 1. Тази рекурентна формула възниква при рекурсивни програми, която изпълнява цикъл по входните данни за да елиминира една от тях:

$C_N = C_{N-1} + N$, за $N >= 2$ и $C_1 = 1$.

Решение: C_N е приблизително $N^2/2$.

$C_N = C_{N-1} + N$

$= C_{N-2} + (N-1) + N$

$= C_{N-3} + (N-2) + (N-1) + N$

…

Продължавайки по този начин, получаваме

$C_N = C_1 + 2 + \ldots + (N-2) + (N-1) + N$

$= 1 + 2 + \cdots + (N - 2) + (N - 1) + N$

$= N . (N+1) / 2.$

Формула 2. Тази формула се получава при рекурсивна програма, която разделя входа си на две половини на една стъпка:

$C_N = C_{N/2} + 1$, за $N >= 2$ и $C_1 = 1$.

Решение: C_N е приблизително lg N. Както е написано, това равенство е безсмислено, освен ако N не е четно и затова приемаме, че N/2 целочислено деление. Нека приемем за момент, че $N = 2^n$, така че рекурентната формула винаги да е добре дефинирана ($n = lgN$.) Тогава:

$$
\begin{aligned}
C_{2^n} &= C_{2^{n-1}} + 1 \\
&= C_{2^{n-2}} + 1 + 1 \\
&= C_{2^{n-3}} + 3 \\
&\vdots \\
&= C_{2^0} + n \\
&= n + 1.
\end{aligned}
$$

Тоест C_N е приблизително lg N.

Формула 3. Тази рекурентна формула възниква при рекурсивни програми, които разделят входа на две половини, но освен това може би трябва да изследват всеки елемент от входа:

$C_N = C_{N/2} + N$, за $N >= 2$ и $C_1 = 0$.

Решение: C_N е приблизително 2N. Рекурентната формула се развива до сумата N + N/2 + N/4 + N/8 + ... Ако редицата е безкрайна, тази геометрична прогресия дава точно 2N. Тъй като използваме целочислено деление и спираме при 1, тази стойност е приближение на точния отговор.

Формула 4. Тази рекурентна формула възниква при рекурсивни програми, които трябва да направят линейно обхождане на входа преди, по време или след разделянето на входа на две половини::

$C_N = 2C_{N/2} + N$, за N >= 2 и $C_1 = 0$.

Решение: C_N е приблизително N lg N. Това решение е най-широко цитирано, защото рекурентната формула е приложима за семейство стандартни алгоритми разделяй-и-владей.

$$
\begin{aligned}
C_{2^n} &= 2C_{2^{n-1}} + 2^n \\
\frac{C_{2^n}}{2^n} &= \frac{C_{2^{n-1}}}{2^{n-1}} + 1 \\
&= \frac{C_{2^{n-2}}}{2^{n-2}} + 1 + 1 \\
&\vdots \\
&= n.
\end{aligned}
$$

От тук следва, че $C_N \approx N \lg N$

Формула 5. Тази рекурентна формула възниква при рекурсивни програми, които разделят входа на две половини и освен това правят константна друга обработка.

$C_N = 2C_{N/2} + 1$, за N >= 2 и $C_1 = 1$.

Решение: C_N е приблизително 2N. Можем да изведем това решение по същия начин, както при формула **4**.

Примери за анализ на алгоритми

Пример 1: Последователно търсене

Тази функция проверява дали числото v е сред числата a[l], a[l+1], ..., a[r], като прави сравнения с всяко число започвайки отначалото. Ако достигнем края и не намерим числото, връщаме -1, иначе връщаме индекса на елемента от масива, съдържащ числото.

```
static int search(int a[], int v, int l, int r)
  { int i;
    for (i = l; i <= r; i++)
      if (v == a[i]) return i;
    return -1;
  }
```

Свойство 1

Последователното търсене изследва N числа при всяко неуспешно търсене и приблизително N/2 числа средно за всяко успешно търсене..

Ако всяко число в таблицата е равно вероятно да бъде търсено, тогава

(1 + 2 + … + N-1 + N)/N = (N + 1)/2

е средната цена на успешно търсене.

От свойство 1 следва, че времето за изпълнение на Пример 1 е пропорционално на N, ако приемем, че средната цена за сравняване на две числа е константа.

Можем да ускорим последователното търсене (особено при неуспешно търсене), ако сортираме числата в таблицата. В подредена таблица можем да прекратим търсенето веднага след достигането на число по-голямо от това, което търсим.

Свойство 2

Последователното търсене в подредена таблица изследва N числа в най-лошия случай и приблизително N/2 числа средно.

Пример 2: Двоично търсене

Тази програма има същата функционалност като Пример 1, но е по-ефективна.

```
static int search(int a[], int v, int l, int r)
  {
    while (r >= l)
      { int m = (l+r)/2;
        if (v == a[m]) return m;
        if (v < a[m]) r = m-1; else l = m+1;
      }
    return -1;
  }
```

Пример 2 е класическо решение на задачата за търсене, което е много по-ефективно от последователното търсене. Тя е основана на идеята, че ако числата в таблицата са подредени, можем да елиминираме половината като сравним търсеното число с числото в средата на таблицата. Ако числата са равни, имаме успешно търсене. Ако търсеното число е по-малко, прилагаме същия метод върху лявата половина на таблицата. Ако е по-голямо – прилагаме същия метод върху дясната половина на таблицата.

Свойство 3

Двоичното търсене никога не изследва повече от $\lfloor \lg N \rfloor + 1$ *числа.*

Доказателство: Нека T_N представя броя на сравненията в най-лошия случай. Тогава от начина, по който алгоритъмът редуцира търсенето в таблица с размер N в търсене в таблица с половината размер следва:

$$T_N \leq T_{\lfloor N/2 \rfloor} + 1, \quad \text{for } N \geq 2 \text{ with } T_1 = 1.$$

От тук като приложим разсъжденията като при Формула 2, ще получим, че

$T_N <= n + 1$ ако $N = 2^n$, което доказва твърдението.

2. Абстрактни типове от данни. Интерфейс и реализация.

Абстрактните типове данни позволяват да се пишат програми, които използват високо ниво на абстракция. Чрез тях може да се разделят концептуалните трансформации върху данните от конкретното представяне на структурите от данни и реализацията на

алгоритмите. Всички компютърни системи са основани на нива на абстракция като например:

- Абстрактен модел на бит с двоична стойност
- Абстрактен модел на компютъра
- Абстрактен модел на език за програмиране
- Абстрактна идея за алгоритъм, реализиран чрез програма на Java
- Абстрактни типове данни – позволяват да се разработят абстрактни механизми на по-високо ниво

За да се разработи ново ниво на абстракция е необходимо:

- *Да дефинираме* абстрактните обекти и операциите върху тях;
- *Да представим* данните чрез определени структури от данни;
- *Да реализираме* операциите.

Необходимо е също *да се отдели* клиента от реализацията по такъв начин, че един клиент да може да ползва множество реализации и една реализация да може да се ползва от много лиенти без да трябва да се променя кода.

Дефиниция. Абстрактен тип данни (АТД) е тип данни (множество от допустими стойности и операции над тях), който е достъпен само чрез **интерфейс**. Програма, която използва АТД се
нарича **клиент**, а програма, която определя типа данни се нарича **реализация**.
Пример: в Java – Java virtual machine е ниво на абстракция, което отделя Java програмите от реализацията на виртуалната машина.
При АТД клиентските програми нямат достъп до данните освен чрез операциите, зададени в интерфейса. Представянето на данните и реализацията на операциите са напълно отделено от клиента чрез интерфейса. Клиентът не може да види реализацията чрез интерфейса.
Пример: Реализация на клас **Point** - Дефинира тип данни, който се състои от множество от двойки реални числа (представящи точки в равнината). Представянето на данните е private. Класът Включва осем метода:

- 2 конструктора
- 2 метода за достъп
- 2 метода за преобразуване в полярни координати
- Метод за изчисляване разстояние между точки
- Метод toString

Реализация:

```
class Point
  {
    private double x, y;
    Point()
      { x = Math.random(); y = Math.random(); }
    Point(double x, double y)
      { this.x = x; this.y = y; }
    double x() { return x; }
    double y() { return y; }
```

```
  double r() { return Math.sqrt(x*x + y*y); }
  double theta() { return Math.atan2(y, x); }
  double distance(Point p)
    { double dx = this.x() - p.x();
      double dy = this.y() - p.y();
      return Math.sqrt(dx*dx + dy*dy);
    }
  public String toString()
    { return "(" + x + ","+y+")"; }
}
```

Клиентските програми не могат да използват полетата x и y, защото са private. Могат да използват само публичните методи.
Не позволявайки клиентските програми да достъпват директно данните, ние сме свободни да променим представянето. Ще разгледаме друга реализация на клас **Point** – вътрешното представяне на точките е чрез полярни координати:

```
class Point
 {
  private double r, theta;
  Point()
    { double x = Math.random(), y = Math.random();
      this = new Point(x, y); }
  Point(double x, double y)
    { r = Math.sqrt(x*x + y*y);
      theta = Math.atan2(y, x); }
  double r() { return r; }
  double theta() { return theta; }
  double x() { return r*Math.cos(theta); }
  double y() { return r*Math.sin(theta); }
  double distance(Point p)
    { double dx = x() - p.x();
      double dy = y() - p.y();
      return Math.sqrt(dx*dx + dy*dy);
    }
  public String toString()
    { return "(" + x() + ", " + y() + ")"; }
}
```

Ще извеждем дефиницията на интерфейса от реализацията на класа като изтрием private членовете, реализацията на public методите и имената на параметрите, като оставим само прототипите на public методите:

```
class Point // ADT interface
 { // implementations and private members hidden
  Point()
  Point(double, double)
  double x()
  double y()
  double r()
  double theta()
```

```
  double distance(Point)
  public String toString()
}
```

Дефинирайки и използвайки интерфейси ние можем да използваме различни реализации, които имат един и същи интерфейс, без да променяме кодът във клиентската програма, която използва АТД. В горният пример ние можем да използваме и двете реализации на класа Point без да променяме клиентската програма. Дефинициите на интерфейсите служат за кратко и пълно описание на договора между клиентите и реализациите, който е основата на ефективното проектиране на АТД.

3. **Свързани списъци. Обработка на списъци.**

Дефиниция: Свързан списък е множество от елементи (items), където всеки елемент е част от **възел** (**node**), който също съдържа **връзка** (**link**) към възел.
Когато основният ни интерес е да обходим последователно съвкупност от елементи, можем да ги организираме в *свързан списък* – основна структура от данни, където всеки елемент съдържа информация как да стигнем до следващия елемент. Това ни дава преимущество за ефективна реорганизация на елементите на свързаният списък, за сметка на липсата на бърз достъп до произволен елемент на списъка - необходимо е последователно обхождане на елементите на списъка докато се достигне желаният елемент. Достъпът до началото на свързания списък е пряк. Има два основни начина за представяне на свързания списък в паметта: с една или с две връзки – съответно едносвързан и двусвързан списък. При едносвързаният списък елементите имат достъп само до съседа си от едната страна, докато при двусвързаният списък елементите имат достъп и до двамата си съседи.
За последния възел имаме следните конвенции:

- Той е нулева връзка (null link), която не сочи никъде
- Реферира се фалшив възел (dummy node), който не съдържа елемент (item).
- Реферира се назад към първия възел, като по този начин списъкът става цикличен

Реализация на Java :

```
class Node
{ Object item; Node next; }
```

Алокиране на памет:

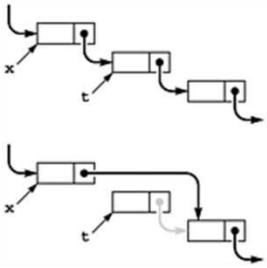
```
Node x = new Node();
```

Инициализация на елементите:

```
Class Node
{ Object item; Node next;
  Node(Object v)
  { item = v; next = null; }
}
...
t = new Node(x);
```

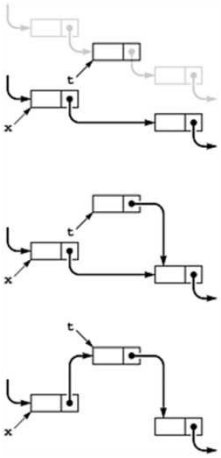
При свързаният списък операциите включване и изключване са допустими на произволно място:

- Изтриване на възел



```
t = x.next; x.next = t.next;    или   x.next = x.next.next;
```

- Вмъкване на възел



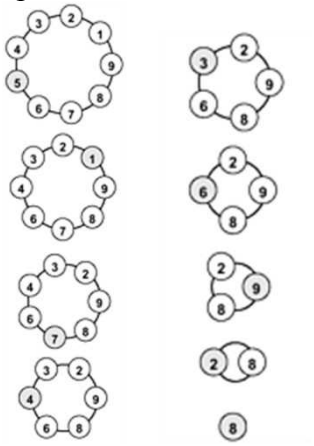
```
t.next = x.next; x.next = t;
```

Пример: Изборът на Josephus ***//този пример може да се изпусне за да бъде по-кратка темата?***
Примерът се изразява в следното: Искаме да изберем един човек от група хора за лидер. За целта хората се нареждат в кръг. Обхождаме кръга с хора и елиминираме всеки M-ти човек (M е дадено число), докато не остане само един човек.
Дадената ситуация може лесно да се моделира със цикличен свързан списък, където всеки елемент представлява човек и има достъп до следващият елемент от списъка.
Реализация на Изборът на Josephus:
Нека N е броя на хората, а M е поредният човек, който ще бъде отстранен:

```
class Josephus
  {
   static class Node
     { int val; Node next;
       Node(int v) { val = v; }
     }
   public static void main(String[] args)
     { int N = Integer.parseInt(args[0]);
       int M = Integer.parseInt(args[1]);
       Node t = new Node(1);
       Node x = t;
       for (int i = 2; i <= N; i++)
         x = (x.next = new Node(i));
       x.next = t;
```

```
    while (x != x.next)
      {
       for (int i = 1; i < M; i++) x = x.next;
       x.next = x.next.next;
      }
    Out.println("Survivor is " + x.val);
  }
}
```

При N = 9 и M = 5 изпълнението на примера изглежда по следният начин:



Ще разгледаме следните операции от обработката на списъци: обхождане на списък, обръщане на списък и сортиране на списък:

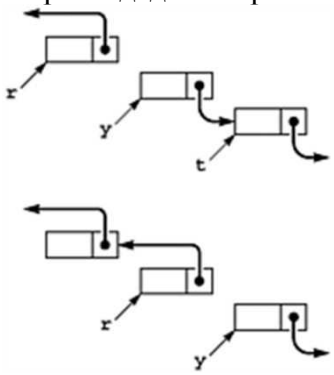
- обхождане на списък

  Ако приемем, че методът visit()извършва желано от нас действие на даден елемент, то можем да обходим елементите на свързан списък по следният начин:

```
for (Node t = x; t != null; t = t.next) visit(t.item);
```

- обръщане на списък

Можем да обърнем даден свързан списък по следният начин:



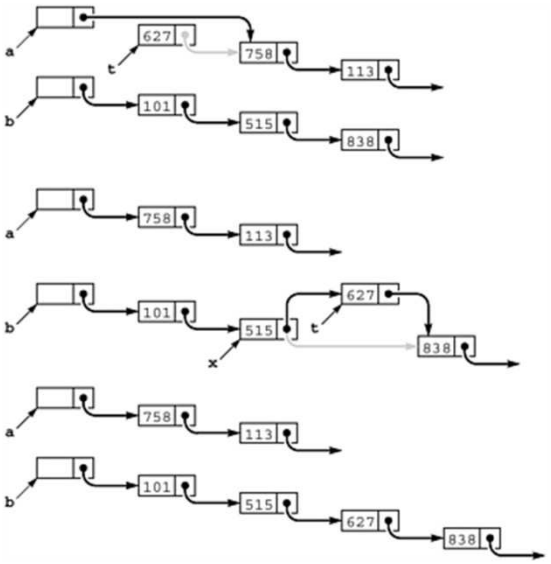
- ☐ r – указател към вече обработения списък
- ☐ y - указател към необработената част
- ☐ Запазваме указател към възела следващ y в t
- ☐ Променяме връзката на y да сочи r
- ☐ Преместваме r да сочи y, а y да сочи t

Реализация:

```
static Node reverse(Node x)
  { Node t, y = x, r = null;
    while (y != null)
      { t = y.next;
        y.next = r;
        r = y;
        y = t;
      }
    return r;
  }
```

- сортиране на списък

Сортирането се изразява в трансформирането на несортиран свързан списък (a) в сортиран (b), използвайки сортиране чрез вмъкване. Това, което правим е да обхождаме елементите на първият списък и да намираме правилното им място във втория, след което да ги вмъкваме, запазвайки вторият списък сортиран.



Реализация:

```
class ListSortExample
  {
    static class Node
      { int val; Node next;
        Node(int v, Node t) { val = v; next = t; }
      }
    static Node create()
      { Node a = new Node(0, null);
        for (In.init(); !In.empty(); )
          a.next = new Node(In.getInt(), a.next);
        return a;
      }
    static Node sort(Node a)
```

```
    { Node t, u, x, b = new Node(0, null);
      while (a.next != null)
        {
          t = a.next; u = t.next; a.next = u;
          for (x = b; x.next != null; x = x.next)
            if (x.next.val > t.val) break;
          t.next = x.next; x.next = t;
        }
      return b;
    }
  static void print(Node h)
    { for (Node t = h.next; t != null; t = t.next)
        Out.println(t.val + ""); }
  public static void main(String[] args)
    { print(sort(create())); }
}
```

4. **Структура от данни стек. Реализация.**

Дефиниция: Стек е АТД (абстрактен тип данни), който има две основни операции: вмъкване (push) на нов елемент и изтриване (pop) на елемента, който последно е вмъкнат.

Интерфейс на АТД стек:

```
class intStack // ADT interface
{ // implementations and private
// members hidden
intStack(int)
boolean isEmpty()
void push(int)
int pop()
}
```

Стекът е крайна редица от елементи от един и същи тип. Операциите включване и изключване на елементи от стека са позволени само в единият му край наречен ‘връх на стека’. Възможен е достъп само до елемента, намиращ се на върха на стека, като достъпът е пряк. При тази организация на логическите оперативи последният включен елемент се изключва пръв. Тоест стекът представлява структура от данни с поведение FILO – First In Last Out. Структурата от данни стек има две основни реализации– динамична и статична:

- **Статичен стек (реализация с масив)**

```
class intStack
{
  private int[] s;
  private int N;
  intStack(int maxN)
    { s = new int[maxN]; N = 0; }
  boolean isEmpty()
    { return (N == 0); }
  void push(int item)
    { s[N++] = item; }
  int pop()
    { return s[--N]; }
}
```

- **Свързан стек (динамична реализация)**

```
class intStack
{
  private Node head;
  private class Node
    {
      int item; Node next;
      Node(int item, Node next)
        { this.item = item; this.next = next; }
    }
  intStack(int maxN)
    { head = null; }
  boolean isEmpty()
    { return (head == null); }
  void push(int item)
    { head = new Node(item, head); }
  int pop()
    { int v = head.item; Node t = head.next;
      head = t; return v; }
}
```

5. **Структура от данни опашка. Реализация**

Дефиниция: FIFO опашка е АТД, който се състои от две основни операции: вмъкване (put) на нов елемент и премахване (get) на елемента, който е вмъкнат най-рано.
Интерфейс на АТД FIFO опашка:

```
class intQueue // ADT interface
{ // implementations and private
// members hidden
intQueue(int)
boolean isEmpty()
void put(int)
int get()
}
```

Опашката е крайна редица от елементи от един и същи тип. Операцията включване е допустима за елементите от единия край на редицата, който се нарича ‘край на опашката’, а операцията изключване на елемент ь от другия край на редицата, който се нарича ‘начало на опашката’. Възможен е достъп само до елемента , намиращ се в началото на опашката, като достъпът е пряк. При тази организация на логическите операции първият включен елемент се изключва пръв. Тоест опашката представлява структура от данни с поведение FIFO – First In First Out. Структурата от данни опашка има две основни реализации– динамична и статична:

- **Статична опашка (реализация с масив)**

```
class intQueue
{
  private int[] q; private int N, head, tail;
  intQueue(int maxN)
    { q = new int[maxN + 1];
      N = maxN + 1; head = N; tail = 0; }
  boolean empty()
    { return (head % N == tail); }
  void put(int item)
    { q[tail++] = item; tail = tail % N; }
  int get()
    { head = head % N; return q[head++]; }
```

}

- **Свързана опашка (динамична реализация)**

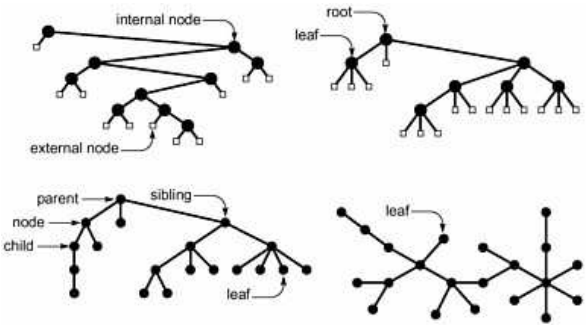
```
class intQueue
{
  private class Node
    { int item; Node next;
      Node(int item)
        { this.item = item; next = null; }
    }
  private Node head, tail;
  intQueue(int max)
    { head = null; tail = null; }
  boolean empty()
    { return (head == null); }
  void put(int item)
    { Node t = tail; tail = new Node(item);
      if (empty()) head = tail; else t.next = tail;
    }
  int get()
    { int v = head.item; Node t = head.next;
      head = t; return v; }
  }
```

6. **Дървета. Типове дървета**.

Дърветата са математическа абстракция,която играе централна роля при проектирането и анализа на алгоритми. Дърветата могат да бъдат използвани за да се да опишат динамичните
свойства на алгоритмите, както за построяването и използването на явни структури от данни, които са конкретни реализации на дървета. Много примери за дървета могат да бъдат намерени във всекидневният живот: семейни дървета, организацията на спортни турнири, структурата на големи организации, файловата система при компютрите и други.
Съществуват много различни типове дървета. Ще дефинираме дърветата като абстрактни обекти и ще въведем основната терминология. Ще разгледаме неформално различни типове дървета по ред на намаляване на тяхната общност:

- Дървета
- Дървета с корен (rooted trees)
- Подредени дървета
- М-арни дървета и двоични дървета

Двоично дърво 3-ично дърво



Дърво с корен Свободно дърво

- *Дърво* е не празна съвкупност от възли и връзки (ребра), които удовлетворяват определени изисквания.
- *Възел* (vertex, node) е прост обект, който може да има име и да носи асоциирана информация.
- *Ребро* (edge) е връзка между два възела.
- *Път* в дърво е списък от различни възли, в който последователните възли са свързани чрез ребра в дървото.
- Дефиниращо свойство на дърво: съществува точно един път свързващ всеки два възела в дървото.
- Ако съществува повече от един път между някои възли или не съществува път между някои възли, тогава имаме граф, а не дърво.
- Непресичащо се множество от дървета се нарича *гора*.
- *Дърво с корен* е такова дърво, в което сме избрали един възел за *корен*.
- **Дефиниция:** *Дърво с корен* (или неподредено дърво) е възел (наречен корен) свързан с множество от дървета с корен.
- **Дефиниция:** *Дърво* (наричано още подредено дърво) е възел (наречен *корен*), който е свързан с редица от несвързани дървета. Такава редица се нарича *гора*.
- Съществува точно един път между корена и всеки друг възел в дървото.
- Казваме, че възел у е *под* възел х (и х е над у), ако х е на пътя от у до корена.
- Всеки възел (без корена) има точно един възел над него, който се нарича негов *родител*. Възлите директно под даден възел се наричат негови *деца*.
- Възли без деца се наричат *листа* или *терминални* възли. Възли, които имат поне едно дете се наричат *нетерминални* възли.
- *Подредено дърво* е дърво с корен, при което е указана подредбата на децата на всеки
  възел
- Ако всеки възел *трябва* да има определен брой деца, подредени по определен начин, тогава имаме *М-арно дърво*.
- При такова дърво, често е подходящо да дефинираме специални външни възли, които нямат деца.
- **Дефиниция:** *М-арно дърво* е или външен възел, или вътрешен възел, свързан с подредена редица от М дървета, които също са М-арни дървета.
- Листо в М-арно дърво е вътрешен възел, всички деца на който са външни възли.

- Разликата между подредените и M-арните дървета, е че възлите в подредените дървета могат да имат произволен брой възли, докато възлите на M-арните дървета имат точно M деца.
- В частност, най-простият тип M-арно дърво е двоичното дърво. *Двоично дърво* е подредено дърво, състоящо се от два типа възли: външни възли без деца и вътрешни възли с точно две деца. Тъй като двете деца на всеки вътрешен възел са подредени, ще ги наричаме *ляво дете* и *дясно дете*.
- **Дефиниция:** Двоично дърво е или външен възел, или вътрешен възел, свързан с двойка двоични дървета, които се наричат ляво поддърво и дясно поддърво на този възел.

7. **Сортиране. Елементарни методи за сортиране.**

Най-общо казано сортирането представлява процесът на пренареждане на елементите на някакво множество от обекти в определен ред. Сортирането е основна дейност с широка сфера на приложение – речници, телефонни указатели, справочни индекси и други – навсякъде, където се налага бързо търсене и намиране на различни обекти.

Ще разгледаме методи за сортиране на множества от обекти съдържащи "ключове". "Ключовете", които са обикновено малка част от обекта, се използват за да контролорат сортирането. Целта на един метод за сортиране е да пренареди обектите, така че техните ключове да са подредени по дадено добре определено правило – обикновено числова или азбучна подредба.

Съществуват различни класификации на на алгоритмите за сортиране. Една от най-популярните е в зависимост от местонахождението на данните: по този критерии различаваме вътрешно сортиране, когато данните се намират в оперативната памет на компютъра, и външно сортиране, когато данните са във външна памет за компютъра. Главната разлика между двете е , че при вътрешното сортиране можем да достъпваме обектите директно, докато при външното сортиране достъпът до данните става последователно, или на големи блокове, като се започва от първия елемент. Друга важна класификация на алгоритмите е според това дали те са "устойчиви" или "неустойчиви".

- Дефиниция: Един метод за сортиране се нарича "устойчив", ако запазва относителната подредба на елементите със равни ключове.

Например ако вземем сортиран по азбучен ред списък със ученици, съдържащ името на ученик и годината на завършване и сортираме този списък по година със устойчив метод, то учениците завършили в една и съща година ще са все още сортирани по азбучен ред в изходният списък. Ако използваме неустойчив метод за сортиране първоначалният относителен ред на учениците най-вероятно няма да се запази.

Ще разгледаме реализацията на три елементарни метода за сортиране:

- Сортиране чрез селекция (selection sort)
- Сортиране чрез вмъкване (insertion sort)
- Сортиране чрез метод на мехурчето (bubble sort)

И трите елементарни метода за сортиране отнемат време пропорционално на N^2 за да сортират N на брой произволно подредени елемента. Ако N е малко, то тези методи за сортиране са напълно достатъчни. Те дори могат да бъдат по-бързи от някои по-сложни методи за сортиране при малко N или в някои други случай. Но когато броят N на елементите е голям и те

са разпределени произволно, то гореспоменатите методи стават неефективни поради голямото време за изпълнение.
За представянето и реализацията на алгоритмите ще използваме методите:

- `less` – сравнява 2 елемента
- `exch` – разменя 2 елемента
- `compExch` – сравнява 2 елемента и ги разменя, ако е необходимо

- Сортиране чрез селекция (selection sort)
Сортирането чрез селекция работи по следният начин: Първо намираме наи-малкият елемент в масива и го разменяме със елемента на първа позиция. След това намираме вторият наи-малък елемент и го разменяме със елемента на втора позиция. Породължаваме по същият начин докато целият масив не е сортиран. Този метод се нарича сортиране чрез селекция, защото работи като постоянно се избира наи-малкият оставащ елемент.

Реализация на метода Сортиране чрез селекция:

```
static void selection(ITEM[] a, int l, int r)
  {
    for (int i = l; i < r; i++)
      { int min = i;
        for (int j = i+1; j <= r; j++)
          if (less(a[j], a[min])) min = j;
        exch(a, i, min);
      }
  }
```

- Сортиране чрез вмъкване (insertion sort)

Сортирането чрез вмъкване работи по следният начин: последователно обхождаме един по един елементите и ги поставяме на правилната им позиция. За вмъкването на всеки елемент на правилното му място се налага последователното му сравнение със вече сортираните елементи до откриване на вярната позиция и след това преместване на по-големите от него елементи с една позиция надясно.

Реализация на метода Сортиране чрез вмъкване:

```
static void insertion(ITEM[] a, int l, int r)
  { int i;
    for (i = r; i > l; i--) compExch(a, i-1, i);
    for (i = l+2; i <= r; i++)
      { int j = i; ITEM v = a[i];
        while (less(v, a[j-1]))
          { a[j] = a[j-1]; j--; }
        a[j] = v;
      }
  }
```

- Сортиране чрез метод на мехурчето:

Сортирането чрез вмъкване работи по следният начин: Нека N е броят на елементите, които ще се сортират. Последователно обхождаме елементите на структурата от дясно наляво N на бой пъти и ако има нужда разменяме съседните елементи. По този начин в процеса на сортиране най-малкият елементи, подобно на мехурчета изплуват към началото на масива, откъдето идва и името на метода.

Реализация на метода чрез метод на мехурчето:

```
static void bubble(ITEM[] a, int l, int r)
  { for (int i = l; i < r; i++)
      for (int j = r; j > i; j--)
        compExch(a, j-1, j);
 }
```

8. **Сортиране - QuickSort.**

Сортирането QuickSort е метод за сортиране от типа „Разделяй и владей". Алгритъмът работи като масива се разделя на две части, на базата на някакъв елемент, и след това се двете части се сортират независимо. Основната част от метода е разделянето, което пренарежда масива, така че следните три условия да са спазени:

- Елементът a[i]е на крайната си позиция в масива за някое i
- Никой от елементите a[l], ..., a[i-1] не е по-голям от a[i]
- Никой от елементите a[i+1], ..., a[r] не е по-малък от a[i]

Използваме следната стратегия за да реализираме разделянето: Първо произволно избираме a[i] да бъде разделящият елемент – този, който ще отиде на крайната си позиция. След това обхождаме, започвайки от левият край на масива докато намерим елемент по-голям от разделящият елемент. Също така обхождаме от десният край на масива докато не намерим елемент по-малък от разделящият елемент. Двата намерени елемента от обхождането очевидно не са на местата си в крайно наредениият масив, и затова ги разменяме. Продължавайки по същият начин сме сигурни, че никой елемент от ляво на левият индекс е по-голям от разделящият елемент и никой елемент от дясно на десният индекс е по-малък от разделящият елемент.

Реализация на QuickSort:

```
static void quicksort(ITEM[] a, int l, int r)
 {
  if (r <= l) return;
  int i = partition(a, l, r);
  quicksort(a, l, i-1);
  quicksort(a, i+1, r);
 }
```

Където реализацията на разделящият метод partition следва:

```
static int partition(ITEM a[], int l, int r)
 { int i = l-1, j = r; ITEM v = a[r];
  for (;;)
    {
     while (less(a[++i], v)) ;
     while (less(v, a[--j])) if (j == l) break;
     if (i >= j) break;
     exch(a, i, j);
    }
  exch(a, i, r);
  return i;
 }
```

В основата си ефективността на сортирането зависи от това колко добре разделяме масива на части, което от своя страна зависи от стойността на разделящият елемент. Най-добре би било за разделящ елемент да изберем елемент, който би разделил масива приблизително по средата. Един подход е да избегнем най-лошият случай (а именно да изберем за елемент

екстремум за масива) е да използваме произволен елемент от масива за разделящ. По този начин най-лошият случай ще се случва с пренебрежимо малка вероятност. Друг подход е за стойност на разделящият елемент е да се вземе средната стойност на три произволни елемента от масива. По този начин вероятността да вземем осреднена стойност за масива е по-голяма и това би дало подобрения във ефективността на метода за сортиране.
QuickSort е най-бързият универсален алгоритъм за сортиране, въпреки, че в дадени ситуации, той може да бъде значително по-бавен от средната си сложност. В най-добрия случай, при размер на масива N, ако всеки път избираме средният по големина елемент, за цялостното сортиране на масива ще са ни достатъчни lgN разделяния. Тъй като при всяко разделяне ще правим N на брой сравнения, за общия брой сравнения в най-добрият случай получаваме N.lgN. От друга страна обаче в най-лошият случай, при обратно сортиран масив с рамер N и разделящ елемент – екстрамума на масива, сложността на алгоритъма може да стане ‘квадратнична’ N.N. Въпреки всичко средната алгоритмична сложност на QuickSort е Θ(N.lgN), правейки го най-бързият алогиръм за общо ползване известен до момента.

13. Софтуерно инженерство и неговото място като дял от знанието. Софтуерен процес и модели на софтуерни процеси. Концепция за многократна употреба.

1. Софтуерното инженерство – какво е софтуер, видове софтуер, същност и обхват на софтуерното инженерство.

Софтуер – представлява:

- Инструкции (компютърни програми), които при изпълнение осигуряват желаните характеристики, функциониране и производителност
- Структури от данни, които дават възможност на програмите адекватно да обработват информация
- Документи, които описват работатата и използването на програмите

Софтуер(2) -Компютърни програми и асоцииранатa с тях документация.

Други специфични характеристики на софтуера са:

- Той е логическа едиица
- Процесът на разработка е специфичен и различен от тази на материални обекти
- Той се разработва, а не произвежда
- Не се амортизира
- Сложен е и има тенденция да става все по-сложен и труден
- Абстрактност
- Надеждност
- Преодбладаващата част от софтуера продължава да е “custom build”

Софтуерен продукт – софтуер, предназначен да се продава на клиент/потребител. Разграничават се два типа софтуерни продукти:

- Общи (generic) – самостоятелни продукти/системи, които се продават на пазара и са предназначени за всеки потребител, който иска да ги купи
- Поръчани (customized) – продукти/системи, които са възложени от отделен клиент и се разработват специално за него.

Разграничават се следните категории софтуер:

- Системен – съвкупност от програми, предназначени да обслужват други програми. Има за цел да улесни работата на хардуера. Примери са ОС, драйвери за устройства, компилатори и пр. Характеризира се с:
  - Интензивна комуикация с хардуера
  - Интензивно използване от много потребители
  - Съвместно използване на ресурси
  - Сложно управление на процеси
  - Сложни структури от данни
  - Многобройни външни интерфейси
- Приложен – състои се от самостоятелни програми, които решават специфична нужда на бизнеса. Приложенията обработват бизнес или технически данни, чрез което улесняват бизнес операциите или вземането на решения.
- Научен – обслужва нуждите на отдените дялове от фундаменталните науки. Приложенията са свързани основно със симулации на системи, автоматизирано проектиране...
- Вграден – намира се в дадена система и се използва да реализира и управлява определени функции вместо крайния потребител или системата.
- Продуктова линия – съвкупност от софтуерни системи, които притежават общи черти, задоволяващи нуждите на определен търговски сегмент или предопределена функция.

Разработени са от общи основни компоненти, което намалява цената на разработката на цялата гама от продукти. Специфично е, че всички продукти от линията се издават едновременно.

- Уеб приложения – в най-простата си форма са множество от свързани файлове хипертекст. Съвременните уеб приложения представляват сложни среди, осигуряващи изчислителни функции и съдържание на крайния потребител и се интегрират с бази данни и други приложения.
- Изкуствен интелект – използват се нечислови алгоритми за решаване на сложни проблеми, които не подлежат на изчисления или директен анализ.
- Open source – софтуер, който позволява свободен достъп и използване на първичния му код, както и условията за разпространението му удовлетворяват определена лицензия, позволяваща потребителите да изучават, променят и подобряват софтуера.
- Legacy software – съществуващ софтуер, който е разработен преди няколко десетилетия и продължава да се използва, като непрекъснато е бил променян, за да задоволи промени в бизнес изискванията на платформите. Ключовите характеристики на наследения софтуер са – дълъг живот и критична важност за бизнеса.

Софтуерно инженерство

Дефиниция 1
Прилагането на систематичен, дисциплиниран, количествено измерим поход към разработването, функционирането и поддръжката на софтуера, както и изледване на подходите в тези дейности. Софтуерното иженерство се касае до теориите, методите и tool-овете за професионална разработка на софтуер.

Дефиниция 2
Софтуерното инженерство е инженерна дисциплина, касаеща се до всички аспекти на продукцията на софтуер. Софтуерните инженери трябва да прилагат систематичен и организиран подход към работата си и да използват подходящи инструменти и техники, зависещи от решавания проблем, ограниченията върху разработката и наличините ресурси.

Дефиниция 3
Софтуерното инженерство е дисциплина, която интегрира процеси, методи и средства за разработване на софтуер.

Обхват на софтуерно инженерство – софт. инж. обхваща процеси, методи, софтуерни модели на процеси, качество.
Обхвата на софтуерното инженерство е изключително обширен. Най-общо включва 5 аспекта:

- Исторически – как се е появила дисциплината
- Икономически – цели се разработката на най-икономически ефективното решение
- Поддръжка на готовите решения
- Изисквания, анализ, дизайн – подходи за намаляване на грешките при разработка
- Екипна разработка – техники за управление и организиране на екипите.

2. Софтуерен процес – фази и основни дейности, видове модели и езици за моделиране, шаблони за описание.

Софтуерен процес:

- Последователност от предсказуеми стъпки
- Рамка за задачите, които да се изпълнят, за да се изгради висококачествен софтуер
- Установяващ базата за управление на софтуерния проект и контекста, в който се прилагат техническите методи, създават се работните продукти, осигурява се качеството
- Описание на разработката на софтуер, която включва някои или всички от изброените дейности, организирани по такъв начин, че накрая се създава работещ качествен софтуер.

- Множество дейности, чиято цел е разработката или еволюцията на софтуер.

Етапи при разработване на софтуер

- Анализиране и дефиниране на изскванията
- Проектиране на системата
- Проектиране на програмата
- Писане/кодиране на програмата
- Unit testing
- Integration testing
- Тестване на системата
- Доставяне на системата
- Поддръжка

Основни дейности в софтуерен процес

- ***Комуникация – цели събиране и разбиране на изискванията към софтуера***
- ***Планиране – създава се план за бъдещата работа по разработката на софтуера като се описват рисковете, необходимите ресурси, работните продукти, които ще се произведат, и времеви график на работата***
- ***Моделиране – включва събиране и анализ на изискванията и проектиране на системата.***
- ***Конструиране – включва генериране на код и тестване.***
- ***Внедряване – софтуерът се предоставя на клиента. Клиентът оценява продукта.***

Модел на софтуерен процес – опростено представяне на софтуерен процес, представен от определена перспектива.

Видове модели

- В зависимост от гледната точка:
    - *Workflow model* **–** описва последователноттта от дейности, които се извършват в процеса на разработка. Отделните дейности се описват входните и изходните им параметри и зависимостите една от друга.
    - *Dataflow model* – описва множеството от дейности, които се извършват от гледна точка на преобразуване на данни.
    - *Role/Action model* – описва отелните роли на хората, които участват в процеса на разработане на софтуер, и дейностите, които всяка от тези роли трябва да извършва.
- Описателни или предписателни
    - *Описателни* – описват историята на това как е протекло разработването на софтуерна система
    - *Предписателни* – предписват как би трябвало да се разработи нова софтуерна система. Използват като насоки и рамки за организиране и структуриране на това как трябва да се извършват дейностите, свързани с разработването на софтуер и в какъв ред.

Езици за моделиране на процеси **–** Езиците и формализмите за моделиране на процес се използват за представяне по прецизен и изчерпателен начин на следните характеристики на софтуерния процес:

- Дейностите, които трябва да се извършат, за да се постигнат целите на процеса;
- Ролите на хората, участващи в процеса;
- Структурата и същността на артефактите, които се създават и поддържат;
- Средствата, които се използват.

Видове езици за моделиранре на процеси:

- по езикови класове
    - *дескриптивни* – използват правила или тригери, базирани са на bottom-up метода като отделят специално внимание на времето заразработка и изключенията, които могат да възникнат.
    - *мрежово базирани* – представят процесите на мрежи Петри. Протичането на процеса е лесно разбираемо, но могат да се появят проблеми с абстракцията.
    - *Императивни* – базирани на езиците за програмиране. Процесът е представен като програма.
    - *Хубридни* –кобинират елементи от няколко парадигми в един език.
- спрямо организацията на езика
    - *Entity-Relation* – организирани са като същности, с връзки между тях
    - *Role-Interaction* – фокусират севъху ролите и техните взаимоотношения. Ролите се изпълняват от организационните елементи, а взаимоотношенията са различните взаимовръзки между тях.
    - *Object-Oriented* – вкл. аспектите на процеса в организационния модел. Основната единица е обект, който комбинира данни и функцоналност в един пакет.
- спрямо формалността
    - *Формални* – представя се чрез формален синтаксис и семантика, има ясно дефинирани правила за употребата им и смисъл на конструкциите им.
    - *Полуформални* – обикновено имат графично представяне с формален синтаксис, но не и формална семантика.
    - *Неформални* **–** няма строго дефинирани правила, а смисълът на конструкциите идва от употребата им.

Шаблони за описание на процес – шаблон, който описва доказан, успешен подход и/или последователност от действия за разработване на софтуер. Представлява структурирано описание на процес, което е е метод за описание на важните характеристики на софтуерния процес. Описват какво трябва да се направи. Според нивото си на абстракция са:

- *Task process patterns* – описва отделни стъпки за извършване на действие или задача, която е част от процеса.
- *Stage process patterns* – дефинира стъпките, които се извършват итеративно в рамките на една базова дайност за процеса.
- *Phase process patterns* – дефинира последователнстта и взаимодействието между базови дейности, които се случват в рамките на процеса.

Всеки шаблон трябва да съдържа:

- *Начален контекст*
- *Проблем*
- *Решение*
- *Краен (резултатен) контекст*
- *Свързани шаблони*
- *Начини на употреба*

3. Сравнителен анализ на описателни модели на софтуерен процес

Сравнение съгласно основните характеристики

Моделът на водопада и на бързата разработка представляват постъпкови модели на разработка. Моделът на бързата разработка е високоскоростна адаптация на този на водопада. При тях се последователно се изпълняват основните дейности при разработката на софтуер (събиране и анализ на изисквания, планиране, проектиране, конструиране и внедряване). При модела на водопада всяка дейност трябва да е завършена и работните продукти да бъдат одобрени преди започването на следващата фаза и всяка фаза завършва със създаването на определени документи. Друга важна

характеристика е ясно разграничения процес. При модела на бързата разработка се определят изискванията, планира се изпълнението на проекта и функционалността се разделя на отделни модули, които всеки от екипите моделира и конструира. Разработката завършва с интеграцията на разработените модули и внедряването на готовия продукт.

Спираловидния модел е итеративен/цикличен поход на разработка, който често се представя като спирала. Всяка итерация по спиралата представлява стъпка от разработката. В началото на всяко завъртане по спиралата се извършва анализ на риска. Възможно е за една и съща дейност да се направят няколко последователни завъртания – например за анализ на изискванията. Може да има множество milestones.

Фазовите модели целят доставянето на системата на части. Разликата между постъпковия (инкрементален) модел и итеративния е, че първият доставя само част от цялата функционалност на всяка стъпка, докато итеративния доставя цялостната софтуерна система в началото, макар и част от функционалността да е в примитивна форма. При постъпковия на всяка следваща стъпка реализира нова функционалност, като използва модела на водопада за тези цели. При итеративния модел всяка итерация доставя все по-завършена версия на софтуера – не се добавя нова функционалност, а само се усъвършенства съществуващата. Много често се прилага комбинация между постъпков и итеративен процес – на всяка стъпка се добавя или подобрява функционалност.

При прототипния модел се цели изграждането на прототип за изясняване на изискванията (thrown-away) или бърза доставка на системата (еволюционен).

Методът на формалната трансформация включва трансформиране на спецификацията на системните изисквания до изпълнима програма. При трансформирането трябва да се запази коректността и ясно да се покаже, че изпълнимата програма съответства на спецификацията

Сравнение съгласно проектите, за които са подходящи

Моделът на водопада и спираловидния процес са подходящи за големи проекти. Първият се прилага при проекти с ясни изисквания, ясно дефинирани роли, повтаряеми проекти и такива, в които времето и бюджетът не са критични. Спираловидният процес също е подходящ при разработка на големи системи, но характерното е, че се фокусира върху оценката на риска при всяка итерация. Той обикновено е комбинация между прототипния модел и модела на водопада.

За малки проекти, при които се цели кратка разработка, е подходящ методът на бързата разработка. Единственото изискване е функционалността да може да се раздели на модули, които да се разработят в рамките на 60-90 дни и да са налични достатъчно човешки ресурси, които да работят по отделните модули.

При проекти с неясни изисквания са подходящи спираловидния и прототипния модел. Обикновено прототипния модел с разработването на thrown-away прототип се използва за постигането на яснота относно изискванията.

При липса на досттъчно човешки ресурси обикновено се прилага постъпковия процес.

При наличие на технологични рискове обикновено се прилага постъпковия процес.

При необходимост за бързо доставяне на системата се използват постъпковия, итеративния модел, еволюционния прототип или модела на бързата разработка.

Прототипния модел може да се използва самостоятелно или в комбинация с други модели.

Методът на формалната трансформация се прилага при критични системи, за които трябва да са гарантирани сигурността и отказоустойчивостта на софтуера, преди да започне използването му.

Сравнение по недостатъци

Основните проблеми при модела на водопада са, че е трудно приложим в реалните проекти, тъй като те рядко следват последователния поток на разработка. Също така обикновено е трудно всички изисквания да бъдат формулирани в началото на проекта и е трудно да се реагира на променящи се изисквания. Основната спънка е трудността на въвеждане на промени след започване на процеса. Друг недостатък е, че клиентът няма достъп до междинни версии или отделни модули на софтуера.

Основният проблем при прототипния модел е, че прототипът може да използва значителни ресурси и да не отговори на очакванията.

Проблемите при модела на бързата разработка са, че изисква значителни човешки ресурси при проекти за големи приложения. Може да се прилага единствено при софтуер, който може да бъде подходящо разделен на отделни модули. Многото интерфейси между модулите влияят негативно върху производителността. Моделът не е подходящ при използването на непознати и нови технологии, т.к. е твърде рисков в тези случаи.

При спираловидния модел е трудно да се убедят клиентите, че процесът на разработка е контролируем, а не безкраен цикъл. Неоткриването на основен риск може да доведе до неуспех. Задължително е участието на разработчици с компетентност за оценка на рисковете.

Недостатъците при постъпковия и итеративния модел са, че клиента трябва да участва активно в проекта. Също така моделът може да доведе до scope creep (постепено разширяване на обхвата).

Методът на формалната трансформация изисква специфични умения от разработчиците, трудно е да се дефинират поведенческите аспекти на системата. Прилагането му е скъп и бавен процес.

4. Концепцията за многократна употреба.

Съществуват два концептуални подхода за многокртна употреба на софтуер:

- Базиран на създаване на програмни единици:
    - Шаблони на проектирането
    - Програмна генерация
- Базиран на съществуващи програмни единици:
    - COTS продукти
    - Компонентно-базирано разработване
    - Аспектно ориентирано разработване
    - Използване на съществуващи системи
    - Application frameworks

Предимства на многократна употреба:

- Увеличена надеждност
- Съвместимост със стандарти
- Намалено време за разработка
- Увеличена продуктивност
- Подобряват комуникацията с външни софтуерни системи (software system interoperability)
- Разработка от по-малко хора
- Намалява цените на разработка и поддръжка
- По-малък риск
- Ефективно използване на специалисти

Проблеми на многкратната употреба:

- Увеличени цени за поддръжка
- Липса на поддръжка на tool-ове
- Трудно е да се намерят, разберат и адаптират компоненти за преизползване

Многократна упротреба, базирана на системи

Реализира се чрез:

- Използване на цяло приложение, което да се конфигурира, така че да е съвместимо с околната среда
- Интегриране на 2 или повече системи в нова

За целите на многократна употреба, базирана на преизползване на готови системи се изпозват 2 подхода:

- интеграция на COTS продукти
- разработка на базата на поточни линии

Многократна употреба, базирана на COTS

COTS системите обикновено са цели приложения, предлагащи API. Те са при приложими в широк кръг области като са най-разпространени в системите за електронна търговия.

При разработката с COTS продукти е възможно да съществуват повече от един потенциални кандидати за интегриране в разрботваната система. Обикновено всеки такъв продукт има собствен формати и структура на данните и е небходимо да бъде разработен протокол за обмен на данни между модулите. Също така най-често COTS продуктите предлагат повече функционалност, отколкто е необходима за

съответната интеграция. В този случай е необходимо да бъде предодвратен достъпа до ненужнта функционалност и разработката на wrapper модул.

Основното предимство на разработката чрез интеграция на COTS продукти е по-бързата разработка и евентуало по-ниската цена. Проблемите при този подход са:

- Липса на контрол върху функционалността и производителността
- Понякога интеграцията на COTS продукти е доста труден процес
- Липса на контрол върху развитието и появата на нови версии на продуктите
- Недостатъчна поддръжка от производителите на COTS продуктите

Поточна линия (product line)

Поточните линии представляват приложения с обобщена функционалност, които могат да се адаптират и след това да се използват в специфична област. Адаптацията предполага някои от следните подходи:

- Конфигурация на компонент или подсистема
- Добавяне на нови компоненти в системата
- Избор на компоненти от предварително подготвени библиотеки
- Промяна на подсистеми

Компонентно-базирана разработка

Дефиниция на компонент

Градивна единица на софтуерните системи, вътрешната структура и процесите, на която са видими само за интегратора на системата, но не и за останалите компоненти. Компонентите могат да осъществяват връзка помежду си единствено чрез входно-изходни точки, наречени интерфейси.

Компонетите предоставят услуги, без значение от платформата на която се изпълняват или програмния език, на който са реализирани. Те са независими самостоятелно изпълними единици. Основната цел при компонентите е те да не се компилират преди интеграция с други компоненти.

Компонентите са по-общо онятие от обектите. Всеки компонент може да е изграден от 1 или повече обекти. Всеки компонент може да се изпълнява самостоятелно и чрез тях се постига по-високо ниво на абстракция.

Компонентите предоставят следните интерфейси:

- Входен (requires) интерфейс - Дефинира какво трябва да се предостави на компонента, така че изпълнението му да стане според спецификацията
- Изходен (provides) интерфейс - Дефинира услугата (или резултата от нея), която компонентът предоставя, в следствие на изпълнението му и при условие, че са удовлетворени изискванията на входния интерфейс

Компонентни модели

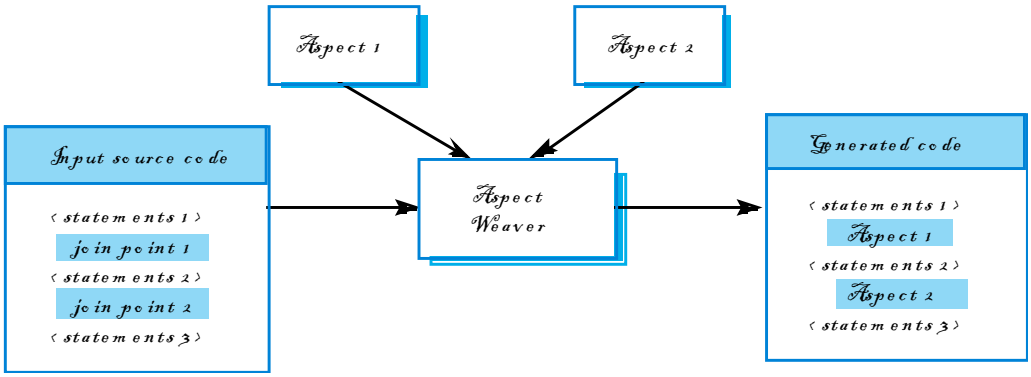
Представляват дефиниция на стандарт за реализация, документиране и метод за разпространение на компоненти. Те определят и правилата за дефиниция на интерфейсите. Примери за компонентни модели са EJB, COM+, .NET.

Аспектно-ориентирано разработване

Аспектно-ориентираното разработване засяга т.нар. разделяне на функциите (*separation of concerns*). Често е много трудно те да се отделят чрез стандартните методи на проектиране. Например:

- Всички компоненти трябва да наблюдават състоянието си
- Всички компоненти трябва да изпълняват политики по сигурността

Припокриващата се функционалност се разработва като аспекти, които динамично се вмъкват в програмата.



Application frameworks

Frameworks са дизайни на подсистеми, съставени от колекция от абстрактни и конкретни класове и интерфейсите между тях. Подсистемите се имплементират чрез добавянето на компоненти и инстанцирането на абстрактните класове във framework-а.

Шаблони на проектирането

Шаблонът на проектирането (*design pattern*) е техника за многократна употреба на някакво абстрактно знание за дадена задача и решението ѝ. Те включват описание на задачата и най-съществените моменти от решението. Абстрактността предполага възможност за прилагане в различни ситуации. Най-често използваните свойства на класовете са наследяване и полиморфизъм.

Класификация на шаблоните на проектирането:

- Според предназначението
    - Съзидателни (*creational*) – за създаване на обектите
    - Структурни *(structural)* – за улесняване на интеграцията на класове и обекти
    - Поведенчески (*behavioral*) – представят методи за комуникация между обектите
- Според обхвата
    - Шаблони на класовете - засягат отношения между класовете, които не се променят след компилацията

- Шаблони на обектите - засягат отношения между обекти. Тези отношения може да се променят по време на изпълнение

Програмна генерация

Програмните генератори използват вече дефинирани шаблони и алгоритми. Те са вградени в генератора и може да се параметризират. Програмата се генерира автоматично.

Програмната генерация е възможна, когато може да се дефинира съответствие между понятията от приложната област и програмния код. Използва се специфичен (за проблемната област) език

Програмната генерация е ефективен метод за разработка, приложим в ограничен набор от приложни области.

13. Софтуерно инженерство и неговото място като дял от знанието. Софтуерен процес и модели на софтуерни процеси. Концепция за многократна употреба.

1. Софтуерното инженерство – какво е софтуер, видове софтуер, същност и обхват на софтуерното инженерство.

Софтуерът се използва от всеки един човек в обществото. Софтуерът се проектира и изгражда от софтуерни инженери. Софтуерните инженери разглеждат софтуера като съставен от програми, документи и данни, необходими за проектирането и изграждането на системата

Софтуерът представлява (IEEE)

- **Инструкции (компютърни програми)** които при изпълнение осигуряват желаните характеристики, функциони–ране и производителност;
- **Структури от данни,** които дават възможност на програмите адекватно да манипулират информация.
- **Документи,** които описват работата и използването на програмите.

Софтуерът, който е предназначен да се продава на клиент/ потребител, се нарича продукт. Могат да се разграничат два типа софтуерни продукти общи (generic) и поръчани (customized/ бутикови) продукти. Общите таргетират голяма група от клиенти и потребители, докато бутиковите се правят за конкретен клиент и са съобразени изключително с неговите нужди и изисквания.

Независимо дали софтуерът е бутиков или с общо предназначение, могат да се идентифицират няколко категории софтуер:

- **Системният софтуер** е съвкупност от програми, които обслужват други програми. Системният софтуер се характеризира с интензивна комуникация с хардуера. Пример за системен софтуер са операционните системи, драйверите, компилаторите, асемблери.
- **Приложният софтуер** се състои от самостоятелни програми, които решават специфична нужда на бизнеса или потребителите. Може да се раздели на няколко големи групи комуникационни приложения (мейл клиенти, уеб браузъри, скайп), мултимедийни приложения (плейъри, графични програми, игри), аналитичен софтуер (статистически програми), collaboration софтуер (уики, блогове, трак), enterprise софтуер (ERP, CRM, BPM) и др.
- **Научен софтуер** обслужват нуждите на конкретна наука . Ползват се в научно–изследователски проекти, за симулации, проектиране, анализ и др.

- **Вграден (εμβεδδεδ) софтуер** нарича се вграден защото се вгражда в различни от класическия компютър хардуерни системи телефони , фотоапарати , автомобили , перални , микровълнови и т.н. Може да е насочен както към потребителски, така и към бизнес нужди.
- **Продуктова линия** съвкупност от приложения, които притежават общи черти или се допълват в задоволяването на конкретна категория потребителски нужди. Продуктова линия е ΜΣ Οφφιχε пакета. Ключово при продуктовите линии е, че има преизползване на софтуерен компонент, на документ или процес при разработката на всяко приложение от пакета. Това преизползване е предварително дефинирано и планирано.
- **Уеб приложения** – множество от свързани файлове хипертекст, които представят информация чрез текст и прости графики. Съвременните уеб приложения обикновено са многослойни и комуникират интензивно помежду си посредством мрежовата среда. Уеб приложенията могат да бъдат силно натоварени в пиков момент. За уеб приложенията е важна естетиката и сигурността на данните.
- **Изкуствен интелект** този софтуер ползва нечислови алгоритми за решаване на сложни проблеми, които не подлежат на директен анализ. Примери за ИИ са някои приложения в роботиката, приложения за разпознаване на образи и звук, умни търсачки.
- **Отворен код (οπεν σουρχε)** всяка от изброените категории може да е с отворен код. Отвореният код и свободният софтуер са нова философия в света на ИТ. Свободният софтуер дава право на потребителите да го ползват, да преглеждат и редактират кода, да разпространяват програмата. Приложенията с отворен код дават същите права, но със следните ограничения:
    - Първичният код трябва да бъде включен, когато се разпространява софтуерът и неговият автор да е видим в явен вид;
    - Повторното разпространение на приложението трябва да става по силата на същата лицензия и правила за разпространение на отворен код.
- **Наследен софтуер** – Съществуващ софтуер, който е разработен преди няколко десетилетия и продължава да се използва, като непрекъснато е бил променян, за да задоволи промени в бизнес изискванията и платформите (хардуерни и софтуерни). Наследеният софтуер се характеризира с
    - Дълъг живот
    - Критична важност за бизнеса
    - *Лошо качество*

Софтуерното инженерство е инженерна дисциплина, която се занимава с всички аспекти на проектирането, разработването и

преизползването на висококачествен софтуер. Като инженерна дисциплина то поставя в центъра си един проблем, анализира го, декомпозира го на подпроблеми, систематизира подпроблемите и търси тяхните решения, които да интегрира в едно крайно решение.

Софтуерното инженерство е прилагането на систематичен, дисциплиниран, количествено измерим подход към разработването, функционирането и поддръжката на софтуера. Софтуерното инженерство трябва да се базира на общо съгласие в рамките на организацията по отношение качеството.

2. Софтуерен процес – фази и основни дейности, видове модели и езици за моделиране, шаблони за описание.

Софтуерният процес е *последователност от предвидими стъпки* – план, който следвате за да създадете продукт или система в срок и с високо качество.
Софтуерният процес може да се разглежда и като *рамка за задачите*, които даден екип от софтуерни инженери трябва да изпълни, за да реализира висококачествен софтуер.
Обикновено дейностите, които трябва да се извършват в рамките на софтуерния процес, са:

- Анализ и дизайн на изискванията;
- Проектиране на системата;
- Проектиране на програмата;
- Писане на код;
- Тестване на единиците;
- Интегрирано тестване;
- Системно тестване;
- Доставяне и внедряване на системата;
- Поддръжка;

Така споменатите дейности дефинират и съответните роли – аналитик, проектант, програмист, тестер, инструктор и др.
Основните цели на софтуерния процес са:

- Ефикасност;
- Възможност за поддръжка (maintainability);
- Предвидимост/ предсказуемост (predictability);
- Повторяемост (repeatability);
- Качество;
- Постоянно усъвършенстване;
- Проследяване на прогреса;

Моделът на софтуерен процес е опростено описание на софтуерен процес, което е представено от определена перспектива. В зависимост от перспективата могат да се разграничат следните модели:

- Workflow model – последователността от действия, които се извършват, описани с техните входно-изходни параметри;
- Dataflow model – проследява преобразуването на данни и при какви дейности става това;
- Role/action model – описва ролите в екипа и съответните дейности, за които всяка роля отговаря;

Целта на моделите на софтуерния процес е да го направят разбираем за всички участници в него и да служи на ръководителите на проекта да проследяват прогреса и евентуално да намират несъответствия и пропуски в процеса.

Езиците за моделиране на процес се използват за детайлното и изчерпателно представяне на *дейностите, ролите на хората, структурата и същността на артефактите и средствата*.
Има три основни парадигми езици за моделиране на процеси:

- **Дескриптивни** – ползват bottom-up метода за описание, като обръщат внимание предимно на времето за разработка и изключенията, които може да настъпят.
- **Мрежово базирани** – представят процесите като мрежи (най-често мрежи на Петри).
- **Императивни** – базирани на езиците за програмиране, представят процеса като програма.

Съвременните езици за моделиране представлятат комбинация от тези три парадигми.

Тъй като има много езици за моделиране на процеси при избора на такъв за даден проект трябва да се водим от следните цели:

- Разбираемост на модела на процеса;
- Лесен дизайн и поддръжка на модела на процеса;
- Симулация и оптимизация на процеса;

Шаблонът е описание на общ проблем, но в допълнение дава идея за неговото решение. Тази идея се ползва милиони пъти, но решението винаги е уникално, защото шаблонът описва **какво** трябва да се направи, а не **как** трябва да се направи.

Шаблонът на процес е шаблон, който описва доказан, успешен подход и/или последователност от действия за разработване на софтуер.
Антишаблон на процес описва последователност от действия за разработване на софтуер, които са доказано неефективни.

3. Сравнителен анализ на описателни модели на софтуерен процес – модел на водопада, прототипен модел, модел на бързата разработка, спираловиден модел и др.

Моделите на софтуерен процес биват два типа – описателни и предписателните. Първи описват как е протекъл или протича софтуерния процес, докато другите описват как би трябвало да протича софтуерния процес. Предписателните модели на процеси дефинират специално множество от дейности, задачи, milestones, и работни продукти, които са необходими за създаването на софтуер с високо качество.

Ще разгледаме най-разпостранените предписателни модели на софтуерни процеси.

3.1. **Waterfall**

- най-старият структуриран модел за разработка на софтуер

- систематизиран и последователен подход към разработването на софтуер, който включва *събиране на изисквания, планиране, анализ и проектиране, имплементация, тестване, внедряване и поддръжка*;
- лесен за разбиране;
- тромав и твърде много бюрократщина, трудно се реагира на промени;
- приложим само в проекти, където изискванията са абсолютно ясни;

3.2. **Модел на бързата разработка (RAD model)**

- Основната цел е кратък цикъл на разработката;
- "Високоскоростна" адаптация на модела на водопада чрез паралелна разработка на системата, което се реализира с помощта на обособяване на модули;
- Скъп модел, поради увеличената консумация на човешки ресурси;
- Компонентно базирани архитектури с цел преизползване на стари компоненти;
- Ако се ползват нови технологии, много вероятно да няма готови компоненти;

3.3. **Еволюционни модели**

3.3.1. **Инкрементален модел**

- Системата не се доставя като едно цяло, а вместо това процесът на разработка и доставянето са разделени на стъпки, като всяка стъпка доставя само част от цялата функционалност.
- На идентифицираните потребителски изисквания се присвояват приоритети и тези с по-висок приоритет се реализират в първите стъпки.
- След като започне разработката на една стъпка, изискванията не се променят.
- Комбинира елементи на модела на водопада, но приложени на отделни стъпки

3.3.2. **Итеративен модел**

- В самото начало доставя цялостната софтуерна система, макар и част от функционалността да е в примитивна форма
- При всяка следваща итерация не се добавя нова функционалност, а само се усъвършенства съществуващата

3.3.3. **Приемущества и недостатъци**

- Необходимостта от активно участие на клиентите по време на изпълнение на проекта може да доведе до закъснения.
- Уменията за комуникация и координация са от особено голямо значение при разработката и ако не са на достатъчно добро ниво, водят до проблеми.
- Неформалните заявки за подобрения след завършването на всяка стъпка могат да доведат до объркване.
- Този модел може да доведе до т. нар. "*scope creep*" – бавно и постепенно разширяване на обхвата на приложението, без процесът да е сходящ
- Прилага се, когато организацията няма достатъчно човешки ресурс за цялостната реализация в определен срок - в разработването на по-ранните версии участват по-малко хора и в зависимост от обратната връзка, получена от клиентите, могат да се присъединят още разработчици на следващите итерации
- Когато с итерациите може да се управляват технологичните рискове - итерация, която изисква използването на нова технология или продукт

може да се планира по-късно с цел да има достатъчно време да се усвои или да се достави новият продукт

3.4. **Прототипен модел**

- Създаване на основните потребителски интерфейси, без да има някакво значително кодиране.
- Разработване на съкратена версия на системата, която изпълнява ограничено подмножество от функции.
- Използване на съществуваща система или компоненти от система, за да се демонстрират някои функции, които ще бъдат включени в разработваната система

3.4.1. **Еволюционен прототип**

- Цел - да достави работеща система на крайния потребител

3.4.2. **Изхвърлен (throw-away) прототип**

- Цел - да подпомогне специфицирането на изискванията към софтуера

3.4.3. **Приемущества и недостатъци**

- Прототипният модел може да използва значителни ресурси, а като резултат прототипът да не успее да удовлетвори очакванията.
- Прототипът може да доведе до лошо проектирана система, ако самият той стане част от крайния продукт.
- Прототипният модел не е подходящ за използване при разработване на софтуерни системи, където проблемът е добре разбран и интерфейсът е ясен и прост
- Прилага се в проекти, където не са достатъчно ясни потребителските изисквания и дизайнът на софтуерната система
- Прилага се както самостоятелно, така и в комбинация с други модели на процеси - модел на водопада, спираловиден модел, постъпков модел и т.н.

3.5. **Спираловиден модел**

- Спираловидният модел е еволюционен модел на софтуерен процес, който съчетава прототипния модел и модела на водопада
- Движещият фактор е анализ на риска
- Основни характеристики: итеративен/цикличен подход и наличие на множество от точки на прогреса (anchor point milestones)
- Може да се окаже трудно да се убедят клиентите, че процесът на разработка е контролируем, а не е безкраен цикъл.
- Изисква се участието на разработчици с компетентност за оценка на рисковете.
- Ако не се идентифицира и открие някой основен риск, това може да доведе до неуспех.
- Прилага се при разработване на големи (large-scale) софтуерни системи.

3.6. Избор на подходящ модел на процес

3.6.1. Изборът зависи основно от два фактора:

- Организационната среда (променяща се, неопределена, стабилна)
- Същността на приложението
- Компетентност и опит на разработчиците
- Използвани технологии
- Бюджет

4. Концепцията за многократна употреба.

В софтуерното инженерство често се търсят решения на сходни проблеми. За целта няма нужда „топлата вода да се открива" отново. Добрите софтуерни инженери постоянно мислят да създават продукт, част от който или целия да могат да преизползват и в друг проект. Това води до повишаване качеството на продукта и намаляване цената му.

4.1. Многократна употреба, базирана на съществуващи програмни единици

4.1.1. Интеграция на COTS (Commercial-Off-The-Shelf) продукти

- COTS системите обикновено са цели приложения, които предлагат API (Application Programming Interface)
- Приложимо в широк кръг области
- Широко разпространение намира в системите за електронна търговия (e-shop, e-mail systems и т.н.)

4.1.1.1. Основни въпроси

- Кои COTS продукти предлагат най-подходяща функционалност?

Възможно е да съществуват повече от един потенциални кандидати за интегриране в системата

- Как ще се обменят данни между модулите?

Всеки модул има собствен формат и структура на данните

- Кои част от продукта ще се използва

Най-често се предоставя повече функционалност, от това което е нужно и трябва да се предотврати достъпа до ненужната функционалност.

4.1.1.2. Проблеми

- Липса на контрол върху функционалността и производителността
- Понякога интеграцията на COTS продукти е доста труден процес
- Липса на контрол върху развитието и появата на нови версии на продуктите
- Недостатъчна поддръжка от производителите на COTS продуктите

4.1.2. Компонентно-базирано софтуерно инженерство

Модел на процес, който стъпва изключително на принципа на преизползването е моделът на компонентно базираното разработване на софтуер. Този модел е комбинация от еволюционния и спираловидния модел.

4.1.2.1. Компонент

- Компонентните са по-общо понятие от обектите
- Всеки компонент може да е изграден от един или няколко обекта
- Може да се изпълняват самостоятелно
- Чрез компонентите се постига по-високо ниво на абстракция
- Предоставят услуги, без значение от платформата на която се изпълняват или програмния език, на който са реализирани
- Независими самостоятелно изпълними единици
- Целта е компонентите да не се компилират преди интеграция с други компоненти

Градивна единица на софтуерните системи, вътрешната структура и процесите, на която са видими само за интегратора на системата, но

не и за останалите компоненти. Компонентите могат да осъществяват връзка помежду си единствено чрез входно-изходни точки, наречени интерфейси

- *Примери за компонентни модели*
  - *EJB (Enterprise Java Beans)*
  - *COM+, .NET*
  - *CCM (Corba Component Model)*

4.1.3. Аспектно-ориентирано разработване

4.1.3.1. **Същност**

Аспектно-ориентираното разработване засяга т.нар. разделяне на функциите (*separation of concerns*).

Често е много трудно те да се отделят чрез стандартните методи на проектиране. Например:

- Всички компоненти трябва да наблюдават състоянието си
- Всички компоненти трябва да изпълняват политики по сигурността

Припокриващата се функционалност се разработва като аспекти, които динамично се вмъкват в програмата

4.2. Многократна употреба, базирана на създаване на програмни единици

4.2.1. Шаблони на проектирането

4.2.1.1. **Същност**

- Шаблонът на проектирането (*design pattern*) е техника за многократна употреба на някакво абстрактно знание за дадена задача и решението й
- Описание на задачата и най-съществените моменти от решението
- Абстрактността предполага възможност за прилагане в различни ситуации
- Най-често използваните свойства на класовете са наследяване и полиморфизъм

4.2.1.2. **Защо се използват шаблони на проектирането**

- Дефинирани са на общодостъпен език
- Независими са от езика за програмиране

4.2.1.3. **Класификация**

- Създидателни - използват се за създаване на обектите (Singleton, Factory, Builder)
- Структурни - използват се за улесняване на интеграцията на класове или обекти
- Поведенчески - представят методи за комуникация между обектите (Observer)

4.2.2. Програмна генерация

4.2.2.1. **Същност**

- Програмните генератори използват вече дефинирани шаблони и алгоритми

- Те са вградени в генератора и може да се параметризират.
- Програмата се генерира автоматично
- Програмната генерация е възможна, когато може да се дефинира съответствие между понятията от приложната област и програмния код
- Използва се специфичен (за проблемната област) език
- Ефективен метод за разработка
- Приложимостта му е в ограничен набор от приложни области
- От гледна точка на крайния потребител е по-лесно в сравнение със стандартните компонентни подходи
-

16. Управление на качеството на софтуерни приложения. Тестване на софтуер. План за тестване и тестови сценарии.

Една от основните характеристики на софтуерното инженерство като дял от бизнеса е разработката на КАЧЕСТВЕН софтуер. Качеството на софтуера най-общо казано е съответствието между създадения продукт или услуга и нуждите и очакванията на потребителя. Оценяването на качеството на даден софтуер се осъществява посредством моделите на качество.

1. Модели на качеството (класически и съвременни).

Качеството представлява композиция от много характеристики. Разработените модели представляват описание на определен набор от тези характеристики и взаимовръзките между тях.

1.1. **Класически модели**

- Тези модели се фокусират върху качеството на крайния **продукт** и във връзка с това задават ключовите **фактори** за качеството от гледна точка на потребителите.
- Факторите на качеството са абстрактни и не могат да бъдат директно измервани. За целта се прави декомпозиция на факторите до **атрибути**, които могат да бъдат директно измервани.
- По своята същност тези модели са **йерархични** модели.

1.1.1. **Модел на McCall**

Обхваща три обособени области в употребата на софтуерния продукт:

1.1.1.1. **Работа с продукта**.

Извършва се декомпозиция до следните фактори:

- Лекота за използване /Usability/
- Интегритет /Integrity/
- Ефективност /Efficiency/
- Коректност /Correctness/
- Надеждност /Reliability/

1.1.1.2. **Ревизия на продукта**. Факторите са:

- Възможност за поддръжка /Maintainability/
- Възможност за тестове/Testability
- Гъвкавост /Flexibility/

1.1.1.3. **Промяна на продукта** – фактори:

- Възможност за повторна употреба (Reusability);
- Възможност за преносимост /Portability/;
- /Interoperability/

Всички тези фактори се разбиват до съответното множество от измерими атрибути, наричани също така **критерии за качество**.

1.1.2. **Модел на Boehm**

1.2. **Съвременни модели**

1.2.1. **ISO 9126**

Модел за качество на продукт. Изхожда от модела на Маккол и най-приложимият към настоящият момент за оценка качество на продукт.

Функциониране /Functionality/

- Приложимост /Suitability/

- Прецизност /Accuracy/
- Сигурност /Security/
- /Interoperability/

Надеждност /Reliability/

- Степен на развитие /Maturity/
- Допустимост по отношение на откази /Fault-tolerance/
- Възможност за възстановяване /Recoverability/

Лекота за използване /Usability/

- Интуитивен /Understandability/
- Лесен за усвояване /Learnability/
- Лекота на осъществяване на операциите /Operability/

Ефективност /Efficiency/

- Бързодействие /Time behavior/
- Употреба на ресурси /Resource behavior/
- Възможност за анализи/Analyzability/
- Възможност за промени /Changeability/
- Стабилност /Stability/
- Възможност за тестване /Testability/

Възможност за преносимост /Portability/

- Възможност за адаптация /Adaptability/
- Възможност за инсталация /Installability/
- Съответствие /Conformance/
- Заменяемост /Replacebility/

1.2.2. **NASA - SATC Software Quality Model**

1.2.2.1. **Същност**

Този модел е насочен както към качеството на крайния продукт, така и към качеството на процеса по разработка на софтуера.

- дефинира множество от цели, които са свързани както с факторите за качество на софтуерния продукт, така и с динамиката на тяхното развитие по време на жизнения цикъл на проекта.
- съдържа процесно-ориентирани индикатори за качество както и класическите индикатори за продукт
- обвързан е силно с управлението на риска в проекта.

1.2.2.2. **Цели на модела**

- Качество на изискванията /Requirements Quality/;
- Качество на продукта{Кода} /Product(Code) Quality/;
- Ефективност на имплементацията (Implementation Efficiency);
- Ефективност на тестването (Testing Efficiency);

Целите се оценяват с дефинирано множество от атрибути, които са измерими **в рамките на жизнения цикъл** на проекта и дават нужната информация.

16. Управление на качеството на софтуерни приложения. Тестване на софтуер. План за тестване и тестови сценарии.

2. Класификация и видове тестове в зависимост от готовността на софтуерното решение - тестване на софтуерна единица, тестване на модул, интеграционно тестване на модули, тестване на система, потребителски тестове за приемане на системата).

Тестването е процес на изследване на софтуерните системи, чиято цел е да провери дали те удовлетворят потребителските изисквания и нужди. В хода на изследването могат да се открият грешки и несъответствия.
В зависимост от етапа на разработка на софтуерния продукт се определят следните типове тестове.
Тестов сценрий – последователност от потребителски стъпки, които реализират даден бизнес сценарий. Тестовият сценарий се състои от предпоставки, входни данни за изпълнение, дефинираната последователност от елементарни потребителски стъпки и очаквани резултати. При създаването на пакет от тестови сценарий се цели пълно покриване на всички възможни ситуации, с колкото се може по-малък брой и по-прости сценарии.

2.1.Тестване софтуерна единица (Unit testing)

Софтуерна единица е термин, който носи значението на най-малката компилируема единица на софтуерната система.
Тестването на софтуерни единици обикновено се извършва от самите програмисти. Изисква детайлно познаване на дизайна и кода на модулът, в който попада единицата. При тестването на софтуерни единица предварително се дефинират статично входни параметри, с които ще се тества единицата и съответните изходи, които се очакват като резултат при изпълнението на тестовете. Тези тестове се пускат многократно в процеса на разработка на софтуер и гарантират, че съответната единица работи коректно.
Тестовете се базират на техническата спецификация и структурата на кода. Прилага се подхода на бялата кутия.

2.1.1. Помощни средства

Тестването на единици обикновено се осъществява с помощта на специални среди и инструменти за тестване.

- Такова помощно средство е **stub**-ът, който се ползва като заместител на програмни единици, с които комуникира единицата, която е обект на тестването. Най-често с оглед на това да тестваме единицата в изолация от другата част от системата се създават въпросните **stub**-ове.
- Друго помощно средство е **test driver**-а, който изпълнява функционалностти на дадена единици с цел да я изтества.

2.1.2. Подходи за unit testing

- **Top-down unit testing**
  Конструира се йерархия/дърво от единици, като най-отгоре е единицата, която ползва други единици. В началото ползваните единици се заместват със stub-ове и постепенно вместо stub-ове в тестовете участват реалните единици. Този подход е удачен за тестване при интеграция на изтествани вече единици.
- **Bottom-up unit testing**

Тестването започва от най-ниските нива на йерархията от зависимости между програмни единици. След като се изтества дадена единица се отива една стъпка нагоре в йерархията. При този подход се ползват test driver-и, но не и stub-ове.

- **Isolated unit testing**
  Предимствата на изолираното тестване на единици е, че ако възникне грешка, веднага може да се каже в коя програмна единица е проблемът. Този подход в тестването е нужен, за да се използват другите подходи – bottom-up и top-down.

2.2.Модулно тестване (Module testing)

При модулното тестване се изследват функционални единици на софтуерната система. Тестващият трябва да може да се ориентира в самия код. Ако липсва пълна яснота по отношение на програмните единици в модула и тяхната йерархия, този тест се нарича **grey box** – свързва се с лимитираната видимост на кода. По-често метод за тестване на модули е методът на **бялата кутия,** при който тестващият има на разположение кода и го разбира.

Тестовете се извършват от програмисти и тестери. Необходимо е да се състави тестов план, включващ тестови сценарии, входни данни и очаквани резултати. Грешки, открити на този етап от разработката на софтуерното решение, са от висока важност. Обикновено се документират в Bug-Tracking системата.

2.3.Интеграционно тестване на модули (Integration testing)

Съвместното тестване на модули се нарича интеграционно тестване. Интеграционите тестове изследват как два или повече модула работят заедно. Този тип тестове могат да се разделят на две основни групи:

- С ниско ниво на интеграция (два или три модула)
- С високо ниво на интеграция (4+)

Интеграционното тестване най-често ползва white-box testing подхода с оглед на това да има видимост върху евентуални конфликтни зони между отделните модули. Подобни конфликтни зони, където най-често се откриват грешки са **интерфейсите, използваните работни области в паметта и базата данни**.

2.4.Системно тестване (System testing)

Системното тестване стартира едва след като всички функционални и технически изисквания на клиента са имплементирани. Включва различен набор от тестове в зависимост от спецификата на проекта. Задължително се тестват **всички възможни бизнес сценарии**, **сигурността** на системата, **надеждността** й и **възможността за възстановяване** след авария.

- Алфа тестове – след реализация на спецификациите
- Бета тестове – след алфа тестовете и при доказано стабилно поведение се стартира бета тестинг, чийто извършител са реални потребители
- Финални /release/ тестове – последни тестове, не се предвиждат никакви повече промени в приложението

2.5.Потребителски тестове за приемане на системата (User-acceptance testing)

- Обучение на потребителите
- Съвместни прегледи
- Тестване
- Анализ на резултатите
- Протокол за приемане на системата
- Описание на процедурата за управление на грешки

3. Подходи за тестване - бяла кутия, черна кутия, сива кутия.

3.1.Бяла кутия

При този подход за тестване, известен още като **white-box testing** и **glass-box testing**, тестващият има пълен достъп до кода на програмата и го разбира. По този начин тестващият има представа каква част от кода биват покрити от неговите тестове (**coverage of code statements, branches**). Този подход е приложим от хора с поне базови програмни и алгоритмични умения.

3.2.Черна кутия

При този подход тестващият няма никакъв достъп до кода на програмата. Той ползва единствено техническа спецификация на системата, за да създаде пакет от тестове с валидни и невалидни входни данни, с който пакет да верифицира и валидира коректността на системата.

3.3.Сива кутия

При този подход тестващият има лимитирана видимост върху софтуера. Най-често тази видимост е върху дизайнът на модула/ системата и/или върху интерфейсите на дадени модули или други програмни единици.

4. Класификация и видове тестове, свързани с функционални и технически изисквания към системата - функционално, регресионно, тестване на производителност, тестване на сигурност, други.

4.1.Функционални тестове

Тези тестове целят да покрият колкото се може по-голям процент от всички бизнес процеси и сценарии. За целта се ползва black-box тестване, при което тестерът не трябва да познава кода, а да симулира реалното поведение на потребителите. Работи се с набор от валидни и набор от невалидни данни. При едните програмата трябва да работи коректно, а при другите да уведомява адекватно потребителя за възникнали грешки.

4.2.Регресионни тестове

Това тестване се изпълнява винаги след извършването на значителни промени по кода с цел да се провери дали промените в една област не са довели до грешки в друга. При компаниите, които държат на качеството на

своите продукти регресионни тестове се правят при всеки build, което най-често става автоматично със съответните помощни средства.

4.3.Smoke/Sanity тестове

Тези тестове целят да покрият само основната функционалност на даден софтуер. Ако те минат, тогава се пускат целият пакет от тестове.

4.4.Тестове за производителност /Performance/

Служи за оценка поведението на системата при реални натоварване. Критерии за качество са response time, transaction rates. Симулира се паралелната работа на множество потребители.

- Load testing – тества системата при очакваното натоварване
- Stress testing – тества системата при свръхнатоварване

4.5.Тестове за сигурност /Security/

- Приложно ниво на сигурност (потребителски достъп)
- Системно ниво на сигурност

4.6.Тестове за удобство за работа /Usability/

Тези тестове са субективни, зависят от характера на съответните потребители. Целят да проверят дали интерфейсът на програмата е ясен и разбираем за потребителите. Тестовете обикновено включват наблюдения върху работата на група потребители със системата.

4.7.Тестове за възстановяване след срив /Backup and recovery tests/

Този тип тестове има за цел да изследва поведението на системата при евентуален неин срив. Проверяват се вътрешните й механизми за съхранение на текущото състояние и механизмите за възстановяване на предишно състояние. Целта е да се удостовери, че при изключителни ситуации няма да има загуба на данни.
Тестването протича като система се подлага на извънредни ситуации – хардуерна повреда, входно-изходни грешки, след което се проверява **реакцията на системата, времето за възстановяване** и евентуално количеството **загубени данни**.

5. Техники за оптимизация на функционални тестови сценарии - клас на еквивалентност, анализ на граничните стойности, таблица за вземане на решение).

5.1.Класове на еквивалентност

5.1.1. Същност

Работи се с определени класове на еквивалентност. Входните елементи се обединяват в групи, категоризирани според резултатите или изходното поведение на програмата. Ако вход А и вход Б водят до едни и същи резултати, то тези два елемента попадат в един и същ клас на еквивалентност.

5.1.2. Приемущества

Разделянето на класове на еквивалентност повишава ефективността на процеса по тестване, намалявайки броя на реализираните тестови сценарии без това да се отрази върху коректността на тестването.
На практика избягваме изпълнението на излишни тестове, но запазваме пълнотата и коректността на тестовия процес.

5.1.3. Приложение

Подобно разделяне на едно множество на различни класове на еквивалентност е приложимо не само спрямо входа, а и спрямо изхода или други характеристики на работата със системата.
Единственото нужно е да се определи **Релация на еквивалентност** спрямо дадените множества (вход, изход)

- Weak equivalence class testing – по една променлива за всеки клас на еквивалентност;
- Strong equivalence class testing – пълно комбиниране на класовете на еквивалентност;

5.2.Анализ на граничните стойности

Това е най-разпространената техника за функционално тестване. Тя се фокусира върху изследване на областта на входните параметри. Идеята на тази техника е за всяка променлива да се разгледат 3 стойности – минимална, близка до минималната (но във вътрешната част на интервала от допустими стойности), номинална, близка до максималната (но във вътрешната част на интервала от допустими стойности) и максимална.

Основният принцип, който се използва, е **Принципът на единичната грешка** – > причината за дефект в някакъв продукт много рядко се дължи на едновременното допускане на две различни грешки. В резултат на този принцип, тестовите случаи се построяват по следния начин: *във всеки тестов случай точно една от променливите се взима с някоя от граничните стойности (тъй като възможността за грешки при тях е най-голяма) и се комбинира с номиналните стойности на останалите променливи*.

Техниката на граничните стойности е подходяща за числови променливи, чиито допустими стойности са зададени с интервал, когато входните променливи са независими една от друга и когато променливите имат характер на физически величини – температура, скорост, ъгъл.

Тестването за устойчивост е разширение на тестването с гранични стойности – добавят се още две възможни стойности – близо до минималната, но извън интервала от допустими стойности, и близо до максималната, но извън интервала от допустими стойности. Целта на тези тестови сценарии е да се провери дали системата е стабилна, когато с нея се работи неправилно.

Разновидност на тестването с гранични стойности е **Тестването в най-лошия случай** (worst case testing). При него не се спазва Принцип за

единичната грешка и се изследва поведението на системата, когато повече от една променлива има стойност, различна от номиналната.

5.3.Таблица за вземане на решение

При тази техника се обвързват изискванията към системата, които съдържат различни логически условия, с вътрешния й дизайн. На базата на анализ на спецификацията се идентифицират различните входни състояния /комбинация от възможни входове/ и обработките, през които трябва да премине системата в зависимост от тях. Възможните условия се представят като комбинация от False/True входни стойности.
Описват се и всички допустими действия, които могат да се асоциират към условията. Таблицата дава зависимостта за изпълнението или неизпълнетието на дефинираните действия, в зависимост от изследваната входна комбинация.

6. План за тестване и тестови сценарии – предназначение и примерна структура

- Изготвя се от Ръководителя по качеството в проекта, съгласува се Ръководителя на проекта. По-скоро има характер на вътрешен документ, но в някои случаи може да се предостави на клиента по негово искане.
- Базира се на функционални и дизайн спецификации, както и на изисквания, договорени с процедурата за управление на промените в проекта;
- Съгласуван е с Плана на проекта;
- Създава се паралелно с техническата документация към проекта – т.е в началото на фазата за кодиране и тестване /development/
- Документ, който се развива и отразява промените в проекта;

6.1.Структура на плана

- Цел и предназначение на плана
- Обхват на тестовите активности
- Какво ще бъде тествано
- Какво няма да бъде тествано
- Референции към документи, на които се базира плана
- Критерии за достигане на планираното и договореното качество и за приемане на софтуера
- Описание на тестовия хардуер – тестов сървър(и), клиент, комуникации ..
- Описание на необходимия тестов софтуер
- Описание на необходими предпоставки и зависимости за провеждане на тестовете – наличие на набор с тестови данни,предварително обучение, /рискове?/ …;
- Инструменти за тестване, които ще се използват – за управление на грешките /Bug Tracking/, за автоматизирано тестване /Code Coverage, Memory Leaks, Functional, Stress, Performance …/ , Installation-monitoring tools, Virtual Machines
- Тестов екип – роли и отговорности, необходима подготовка, управление на тима (планиране на срещите на тима, срещи с клиенти …)

16. Управление на качеството на софтуерни приложения. Тестване на софтуер. План за тестване и тестови сценарии.

6.2. Примерен списък на групите тестове, които ще се извършат

- Тестове за качество на кода – code coverage, memory leaks …;
- Тестове на бизнес процеси / функционални тестове / - Списък на процесите, които ще бъдат покрити с тестове, с референция към съответстващите им тестови сценарии. Описание за реализация на regression testing;
- Тестове за производителност – списък и описание на скиптовете и референции към тях;
- Тестове за сигурност и права на достъп– списък и описание на тестовите сценарии с референции към тях;
- Тестове за архивиране и възстановяване след авария - списък и описание на тестовите сценарии с референции към тях;
- Тестове за инсталация - описание;
- Тестове на съпровождащата документация – описание;
- Тестове за приемане на системата – опционални, често се представят като отделна процедура за приемане (**User Acceptance Procedure**);

6.3.Тестови сценарии

Приоритет на тестовия сценарий – определя се от :

- Дали представя основен процес;
- Брой на клиентите, които работят с него;
- Дали е критичен за работата на останалата част на системата.

Тест сценариите идентифицират “дълбочината” на тестовия процес –респективно на степента на качеството.
Усилията за тестване на даден продукт могат да се оценят и на базата на множеството от тест сценарии – идентификация, разработка и изпълнение и отчетите от тях.
Тест сценариите се категоризират в зависимост от задачите, които изследват – функционалност, бързодействие, сигурност ...

17. Софтуерна архитектура. Проектиране и документиране на софтуерни архитектури.

С навлизането на софтуерните системи във все повече области през последните десетилетия – медицина, образование, държавна администрация, бизнес, софтуерните системи стават все по-сложни и разнообразни. Повишената комплексност на системите неминуемо води до нуждата от по-абстрактен поглед върху структурата на една система с цел тя да бъде **разбираема** от всички участници в процеса на изграждане на софтуерния продукт. Описанието на въпросния абстрактен поглед върху софтуерната система се нарича още **софтуерна архитектура**.

1. Дефиниция на софтуерна архитектура. Структури и изгледи (*structures and views*) на архитектурата.

1.1. Какво е софтуерна архитектура?

- **Дефиниция на SEI:**

"Архитектура на дадена софтуерна система е някаква съвкупност от структури, показващи различните софтуерни елементи на системата, външно видимите им свойства и връзките между тях".
"Външно видими" свойства са онези предположения, които другите елементи могат да направят относно елемента – какви услуги предлага, с какво бързодействие, как се оправя с грешките, със споделените ресурси и т.н.

- Следователно, първо и най-важно, СА е абстракция, която **скрива детайлите**, от които взаимодействието между елементите не зависи;
- Съгласно дефиницията става ясно, че системите могат да имат (и имат) **повече от една структура**. Нито една от тях самостоятелно не представлява Архитектурата на системата (структура на модулите, на процесите; елементите може да са обекти, модули, процеси, БД, библиотеки, продукти, екипи и т.н.);
- Обикновено СА се създава като **първа стъпка по време на проектирането**, като целта е да се гарантира наличието на дадени качества в системата;
- Детайли като алгоритми, представяне на данни, реализация, и т.н. не са предмет на СА;
- Предмет на СА е поведението и връзките между различни елементи, разглеждани като "черни кутии" (**systems theory**);

1.2. Какво определя софтуерната архитектура?

Изборът на софтуерна архитектура зависи от различни характеристики на средата. От друга страна избраната вече софтуерна архитектура налага известни ограничения върху средата. Тоест процесът по определяне на архитектурата и средата е цикличен (Architecture Business Cycle).

1.2.1.1. Функционалните и технически изисквания
1.2.1.2. Изменяемост и устойчивост на бизнес средата
1.2.1.3. Изменяемост, устойчивост и структура на организационната среда

1.2.1.4. Опит (с предишни системи и архитектури), знания и умения на архитекта и програмистите
1.2.1.5. Влияние на заинтересованите лица (шефове, клиенти, потребители)
1.2.1.6. Ограничения и възможности на съвременните технологии

1.3. От какви качества се нуждае софтуерният архитект?

В крайна сметка софтуерният архитект трябва **да балансира** избора си между очакванията на всички заинтересовани лица.
Отговорност на софтуерния архитект е **да притисне** ЗЛ да сближат позициите и очакванията си по отношение на софтуерната система.

1.3.1.1. Отлично познаване на технологиите;
1.3.1.2. Отлично аналитично мислене;
1.3.1.3. Комуникативност, дипломатичност и умения за убеждаване и водене на преговори;

1.4. Отговорности на софтуерния архитект

1.4.1.1. Вземане на бизнес решенията за създаване на системите;
1.4.1.2. Разбиране на изискванията (участие в изготвянето им);
1.4.1.3. Създаване или избор на архитектура;
1.4.1.4. Документиране и преразказване на СА (по различен начин за различните групи ЗЛ);
1.4.1.5. Анализ и оценка (архитектурна и финансова) на СА;
1.4.1.6. Създаване на системата;
1.4.1.7. Следене за наличие на съответствие между системата и СА.

1.5. Първични критерии за качество на софтуерна архитектура

1.5.1.1. Малък екип от архитекти с утвърден лидер, който взима решенията;
1.5.1.2. Добре дефинирани и приоритизирани изисквания;
1.5.1.3. СА трябва да е добре документирана и разбираема от всички ЗЛ;
1.5.1.4. ЗЛ трябва да са активно въвлечени в рецензията и одобрението на СА;
1.5.1.5. Количествени измервания за съответствие с качествените изисквания;
1.5.1.6. СА трябва да позволява постепенна реализация (обособен скелет);
1.5.1.7. Присъствие на софтуерни техники за разумно разходване на ресурси;
1.5.1.8. Модулност на СА;
1.5.1.9. Модулите, които произвеждат данни, трябва да са отделени от модулите, които консумират данни;

1.6. Структури и изгледи на архитектура

1.6.1.1. Както в медицината например, различните специалисти имат различен поглед върху човешкото тяло и всички те заедно дефинират архитектурата на Човека, така и СА предлага *няколко различни погледа (структури)* на софтуерната система;
1.6.1.2. **Структура** – съвкупност от софтуерни елементи, техните външно видими свойства и връзките между тях;

1.6.1.3. **Изглед** – конкретно документирано представяне на дадена структура;

1.6.1.4. **Структура/ Изглед** - взаимозаменяеми

1.7. Типове структури

1.7.1. **Модулни структури** (*кой модул каква функционалност реализира, кой модул с кой друг модул взаимодейства и т.н.)*

- **Декомпозиция на модулите** – разбираемост, разпределение на задачите, лесна променяемост и разширяемост на архитектурата;
- **Структура на слоевете** - Частен случай на структура на модулите е структура на слоевете, при които слой N може да ползва единствено възможностите на слой N-1;
- **Структура на употребата на модули**
  Разработчиците знаят с кои интерфейси трябва да се запознаят, за да си свършат работата по създаването на даден модул;
  Тестерите знаят къде трябва да гледат, за да тестват даден модул;

1.7.2. **Структура на компонентите и конекторите** (*процеси,изпълнявани в компонентите* и *комуникационните канали между процесите*)

- Кои са основните изчислителни процеси в системата? Как тези процеси въздействат върху данните? Кои са споделените ресурси?
- **Структура на процесите** - има отношение към *бързодействието* и *надеждността*. Избягване на deadlock.
- **Структура на конкурентното изпълнение**
- **Структура на споделените данни** – има отношение към бързодействието

1.7.3. **Структура на разположението**
(физическо, ресурсно(CPU), организационно)

- Структура на внедряването
- Разпределение на работата (модул <-> екип/способности)

2. Изисквания към качеството (нефункционални изисквания) на системата.

Качеството е субективно възприятие – различните ЗЛ могат да не одобрят даден дизайн, тъй като тяхната идея за качество се различава от идеята за качество на архитекта;

- Бизнес целите определят **Качествата**, които трябва да бъдат вградени в архитектурата на системата. Тези Качества поставят изисквания отвъд функционалните (описание на основните възможности на системата и услугите които тя предоставя);
- Въпреки, че функционалността и Качествата са тясно свързани, функционалността често е **единственото, което се взема под внимание** по време на проектирането.

- Като следствие много **системи се преправят** не защото им липсва функционалност, а защото е трудно да се поддържат, трудно е да се смени платформата, не са скалируеми, прекалено са бавни, или пък са несигурни.
- **СА е тази стъпка** в процеса на създаването на системата, в която за пръв път се разглеждат качествените изисквания и в зависимост от тях се създават съответните структури, на които се вменява функционалност.
- За да притежава дадена система изискваните качествени характеристики, те трябва да се имат предвид както по време на проектирането, така и по време на разработката и внедряването.

2.1. Сценарий за качество

- Изискванията за качество трябва да се формализират от архитекта посредством т.н. "*сценарии за качество*", за да бъдат те поставени на обективна основа;
- *Сценарият за качество е специфично изискване към поведението на системата в дадена ситуация, в светлината на дадено качество;*
- Сценариите демонстрират какво е качество в рамките на създаваната система, като дават на архитекта и на ЗЛ еднозначна основа за оценка на дизайна;

2.1.1. Характеристики

- **Въздействие** – състояние/събитие, което подлежи на обработка;
- **Източник** – обект (човек, система или нещо друго) който генерира въздействието;
- **Обект** – системата, или конкретна нейна част, върху която се случва въздействието;
- **Контекст** – Условията, при които се намира обекта по време на обработка на въздействието;
- **Резултат** – действията, предприети от обекта при случването на въздействието;
- **Количествени параметри** – резултатът трябва да подлежи на някакви количествени измервания, така че да позволи проверката дали сценарият се изпълнява съгласно изискванията;

- **Сценарий за Изменяемост (Пример)**
  - *"Преди пускането на системата в експлоатация… (Контекст)*
  - *…клиентът… (Източник)*
  - *…желае промяна на фоновия цвят на екрана….(Въздействие)*
  - *За целта се променя изходния код…. (Обект)*
  - *Това не трябва да предизвиква никакви странични ефекти в поведението на системата. (Резултат)*
  - *Времето за извършване на промяната, вкл. тестването й трябва да отнеме по малко от 3 часа*
  - *(Количествени параметри)*

2.2. Взаимовръзка между качествата

Трябва да се има предвид, че качествата са взаимносвързани и завишаването на едно качество, води до понижаване на друго.

- **Сигурност vs. Отказоустойчивост (Fault Tolerance)** – въпреки, че обикновено ги поставят в една и съща графа, те си противоречат –

сигурността изисква наличието на single point of failure, т.е. системата да може да се компроментира от едно единствено място; Обратно, отказоустойчивостта предполага репликация

- **Преносимост vs. Производителност** – основната техника за постигането на преносимост е изолирането на зависимостта от ОС и хардуер в специализирани модули, което внася допълнителна тежест по време на изпълнението и намалява производителността;

2.3. Технологични качества

2.3.1. Надеждност (Reliability)

2.3.2. Изменяемост (Modifiability)

Определя се най-вече от декомпозицията. Свързва с цената на промените.

2.3.3. Производителност (Performance)

Времето, за което системата реагира на дадени събития. Зависи от комуникацията между компонентите, консумацията на споделени ресурси, разпределението на функционалността между компонентите

2.3.4. Сигурност (Security)

Способността на системата да устоява на опити за неразрешена употреба.

2.3.5. Изпитаемост (Testability)

Лекотата, с която системата може да бъде изтествана.

2.3.6. Готовност (Availability)

Честота на сривове и време за оправяне на повредата.

2.3.7. Използваемост (Usability)

2.4. Бизнес качества

2.4.1. Време за пускане на пазара (Time to market TTM)

Намалява се с употребата на COTS продукти.

2.4.2. Себестойност и печалба

Зависи от архитектурата => технология (непозната, изменяема)

2.4.3. Предвидено време за живот (Time to live TTL)

2.4.4. Възможност за постепенно пускане на пазара (extensibility)

2.5. Архитектурни качества

2.5.1. Идейна цялост

Всички да са наясно с общата идея/визия на СА. Подобни задачи да се решават с подобни решения.

2.5.2. Коректност и пълнота спрямо изискванията

2.5.3. Градивност (с каква лекота и в какъв срок наличният екип ще построи системата)

3. Проектиране на софтуерната архитектура. Процес за проектиране. Избор на подходящи структури. Последователност на създаване на архитектурата. Тактики (архитектурни решения) за постигане на желаните качествени показатели.

При наличието на основните функционални и технически изисквания може да се стартира процеса по проектиране на СА. В началото се избират до 10-тина основополагащи функционални, технически и бизнес изисквания, наричани **архитектурни драйвери**. Въпросните изисквания са с най-високи приоритет при проектирането на СА. Анализ на въпросните драйвери и решение на евентуални конфликти между тях. В следствие на анализа някои изисквания може да се променят.

3.1. Attribute-Driven Development (ADD)

- Подход за проектиране, в който главна роля играят качествените свойства (атрибути);
- Рекурсивен процес на дефиниране на архитектурата, като на всяка стъпка се избират конкретни техники и архитектурни модели за постигане на желаните качествени свойства;

3.1.1. Стъпка 1 – Избира се модул за декомпозиция

3.1.2. Стъпка 2 – Избира се архитектурни драйвери

3.1.3. Стъпка 3 – Избира се архитектурен модел (стратегия)

3.1.4. Стъпка 4 – Обособяват се подмодули

3.1.5. Стъпка 5 – Приписване на функционалност към подмодулите

3.1.6. Стъпка 6 – Създават се и други структури

3.1.7. Стъпка 7 – Верификация на декомпозицията

3.1.8. Стъпка 8 – Отиваме на стъпка 1 за някои подмодули

3.1.9. Стъпка 9 – Формират се екипи и се раздават задачи

3.1.10. Стъпка 10 – Създава се СКЕЛЕТ на системата

3.1.11. Стъпка 11 – Документиране на СА

Таблица – кой от какво се интересува (какви View-та)

3.2.Тактики (архитектурни решения)

Тактиката е архитектурно решение, чрез което се контролира резултата от даден **сценарий за качество**. Наборът от тактики се нарича **архитектурна стратегия** или **архитектурен модел**.

A. Тактики за готовност

a. Откриване на дефекти

- **Ping/echo** – един компонент, проверява друг компонент;
- **Heartbeat** – периодично излъчване на сигнал. В сигналът може да има някакви данни за състоянието на компонента или настъпили събития;
- **Exceptions (Изключения)** – обработка на изключения, за да не се чупи цялата система;

b. Отстраняване на дефекти

- **Voting** – на различни процесори работят различни процеси, които получават един и същ вход и би трябвало да генерират един и същ резултат. Ако има разлика в някой резултат централният процесор (voter) предприема някакви действия. Тази тактика обикнове се ползва за тестване на процесори;
- **Active Redundancy –** важните компоненти в системата са дублирани. Дублираните компоненти се подържат в едно и също състояние и резервният компонент е винаги готов да премине в състояние на активност. Downtime-ът е няколко милисекунди;
- **Passive Redundancy –** един активен и няколко резерви, които обаче се актуализират само ако основният компонент се дефектира. Downtime-ът е от няколко секунди, до няколко часа;
- **Резерва (Spare) –** поддръжка на резервни изчислителни мощности, които трябва да се инициализират и пуснат в действие при дефектиране на някой от компонентите;

c. Повторно въвеждане в употреба

- **Shadow Mode (Паралелна работа) –** преди да се въведе в употреба компонент, който е бил повреден, се пуска в shadow mode, за да се провери дали работи коректно.
- **Синхронизация на състоянието**
- **Checkpoints/Rollbacks**

d. Предотвратяване на дефекти

- **Transactions**
- **Removal from service (Извеждане от употреба) –** автоматично или ръчно
- **Process Monitoring –** посредством специален процес се следят основните процеси в системата. Ако даден процес дефектира, мониторинга може да го премахне, рестартира, дупликира и т.н.

B. Тактики за изменяемост

Директно (*променя се поведението*) и индиректно(*не се промня поведението, но се променя реализацията*) засегнати модули от дадена промяна.

a. Локализиране на промените (намалява броя на *директно* засегнатите модули)

- **Поддръжка на семантична свързаност (Cohesion)**
- **Очакване на промените (списък с евентуални промени)**
  Списъкът е основополагащ при декомпозицията на модули.
- **Ограничаване на възможните опции**
  При продуктовите линии се налага ограничение върху дадени очаквани промени с оглед на това да не се изменя тотално цялата продуктова линия.

b. Предотвратяване ефекта на вълната (да няма *индиректно* засегнати модули)

- **Скриване на информацията**
  - Декомпозиция на отговорността на даден елемент (система или конкретен модул) и възлагането ѝ **на по-малки елементи**, като при това част от информацията остава публична и част от нея се скрива.
  - **Публичната функционалност** и данни са достъпни посредством специално дефинирани за целта **интерфейси**.
- **Поддръжка на съществуващите интерфейси**
  - Добавяне на нов интерфейс в модула А
  - Wrapper на модула А
  - Добавя се нов модул A_REAL, а самият модул А става stub
- **Ограничаване на комуникацията (Loose Coupling)**
- **Използване на посредник**
  - **Синтаксис на данните (Repository)**
    *MVC и многослойните архитектури – преобразуване на данните от един слой в подходящ вид за друг слой***)**
  - **Синтаксис на услугите**
    Ползват се посредници – чрез шаблоните *façade, bridge, mediator, strategy, proxy, factory.*
  - **Идентификация на интерфейсите (Broker)**
    А предоставя няколко интерфейса. Б се свързва с брокер, който знае чий интерфейс да му предостави.
  - **Местоположение на А (Name Server)**

c. Отлагане на свързването

- **Plug and Play**
  Възможност за добавяне и изключване на модули, докато системата работи.
- **Component Replacement**
  Възможност за добавяне и изключване на модули, при стартиране на системата.
- **Config files**
  При зареждане на системата или отделен модул от нея се взимат конкретни параметри.

C. Тактики за производителност

a. Намаляване на изискванията

- **Увеличаване на производителността на изчисленията (алгоритми/ хеширания)**
- **Намаляване на режийните (overhed) –** намаляване на действията, които не са пряко обвързани с конкретното събитие
- **Увеличаване на периода при повтарящи се събития**

b. Управление на ресурсите

- **Паралелна обработка**
- **Излишък на данни/ процеси (cache/ load-balancing)**
- **Включване на допълнителни ресурси**

c. Арбитраж на ресурсите

- **Scheduling –** приоритизиране на събитията

D. Тактики за сигурност

a. Устояване на атаки (ключалка на вратата)

- **Автентикация на потребители**
- **Нива на потребителски достъп**
- **Конфиденциалност на данните (криптиране)**
- **Ограничаване на достъпа (firewall, access control)**

b. Откриване на атаки (аларма)

- **Intrusion Detection Systems (IDS)**
  Интелигентно открива в реално време чрез анализ на трафика и БД реализирани атаки.

c. Възстановяване след атака (застраховка)

Тактиките за възстановяване се припокриват с тези за готовност, тъй като една атака може да се приеме като срив на системата.

E. Тактики за проверяемост

- **Разделяне на интерфейса от реализацията**
- **Вградени модули за мониторинг и журнал**

4. Архитектурни стилове и шаблони.

Архитектурният стил дефинира семейство от системи чрез използването на шаблон за структурна организация. Дефинирани са ограничения и начини за използването им.

Архитектурният шаблон (АШ) е описание на типове софтуерни елементи и връзки, заедно с ограничения относно тяхната употреба;

- В шаблоните няма описание на никаква функционалност освен тази, която служи за реализация на комуникационните протоколи;
- Шаблоните са полезни, тъй като те притежават добре известни качества. Архитектите най-често започват проектирането с избор на шаблон, от който да започнат;

4.1. Хранилище (Repository)

Подсистемите си комуникират единствено чрез хранилището. Сравнително независими са. Контролът върху поведението на системата може да се осъществява както от хранилището (при настъпването на някакво събитие в него), така и от подсистемите, които да освобождават ключове в хранилището и по този начин да дават зелена светлина на друга подсистема да ползва съответните ресурси.

4.2. MVC

Разделение на бизнес логиката от ГУИ. Разновидности на MVC

- Maintainability, flexibility

4.3. Клиент – Сървър

Сървърът не знае за клиента. Клиентът обработва и валидира действията на потребителя, след което праща заявки до сървърът. Сървърът се грижи за интегритет на данните.

4.4. P2P

- Прилича на Client-Server с разликата, че сървърът може да инициира контрол върху клиентите и освен това между клиентите може да се осъществява директна връзка.
- Структурирани, неструктурирани и централизирани p2p системи
- Distributed Hast Tables – fast resource discovery
- High Scalability
- Complex due to prevention algorithms of deadlocks

4.5. n-tier architecture

- Calls n-1 tier functionality
- Independent physical machines
- Maintainability, scalability, flexibility, availability

4.6. Pipes and Filters

Use the Pipes and Filters architectural style to divide a larger processing task into a sequence of smaller, independent processing steps (Filters) that are connected by channels (Pipes).

Each filter exposes a very simple interface: it receives messages on the inbound pipe, processes the message, and publishes the results to the outbound pipe. The pipe connects one filter to the next, sending output messages from one filter to the next. Because all component use the same external interface they can be composed into different solutions by connecting the components to different pipes

5. Документиране на софтуерната архитектура. Предназначение на документацията. Основен принцип на документиране. Съдържание на документацията. Структура на документацията.

5.1. Предназначение на документацията

Документацията е от важност с оглед на това да може в бъдеще системата да бъде лесно поддържана, безпроблемно изменяна и ясно разказвана на клиенти, потребители.
Зле написаната документация често е по-лошият вариант от никаква документация, защото може да заблуди четящия.

5.2. Основен принцип на документацията

Документацията се състои от описание на различните структури и когато тя се пише се мисли най-вече **кой ще я чете**. Техописецът трябва да се поставя на мястото на четящия. В документацията различните структури може да се групират по различни начини в зависимост от това, за кого конкретно е предназначена съответната документация.

5.3. Съдържание на документацията

- Първично представяне
- Описание на елементите и връзките
- Описание на обкръжението
- Описание на възможните вариации
- Архитектурна обосновка
- Терминологичен речник
- Допълнителна информация

5.4. Структура на документацията

- Каталог на структурите
- Шаблон за описание на структурите
- Кратко абстрактно описание на системата
- Защо така е проектирана СА

18. Инженеринг на изискванията като част от системния и софтуерен инженеринг. Представяне на техники, прилагани на отделните етапи от инженеринга на софтуерните изисквания. Основни методи за моделиране.

1) Цел и задачи на инженеринга на софтуерните изисквания.

Цели и задачи на инжинеринга на изискванията:

- Да създаде и поддържа документ със системните изисквания.
- Да идентифицира целта на софутерната система и контекстите, в които ще бъде използвана.
- Да играе ролята на мост между реалните нужди на потребителите, клиентите и други, засенати от системата, и възможностите, които се предлагат от софтуерните технологии.
- Да дефинира обхвата на продуктите, които се разработват.
- Да установи и специфицира прецизно какво трябва да прави софтуера без да описва как го прави.
- Да разбере и определи проблема, който трябва да се реши чрез разработката на софтуера.
- Да разбере какво се изисква от софтуера.
- Да комуникира разбирането за това какво се изисква с всички заинтересовани лица.
- Да осигури начини за контрол, които да осигурят, че финалната система задоволява изискванията (включително управлението на ефекта от промените).
- Да определи целите, функциите и ограниченията към софтуерните системи, както и връзката между тези фактори.
- Създаването на прецизна спецификация на поведението на софтуера и еволюцията му.
- Да определи какво трябва да прави софтуера за да добави стойност за заинтересованите лица – функционални изисквания.
- Да определи какво трябва а представлява софтуера – нефункционални изисквания.
- Да определи какви са ограниченията при имплементацията.

2) Параметри на процеса

Класификация на изискванията:

- функционални и нефукнкционални
- бизнес и потребителски
- спрямо устойчивост (stability)
    - стабилни - не се променят по време на разработката, свързани са със същитната на системата или приложната й област.
    - променливи - изисквания, които се изменят с времето или употребата на системата. Характерни са за инстанция на системата в определена среда и за определен клиент.
- съгласно приотиризация
    - задължителни
    - препорачителни
    - възможни - могат да бъдат елиминирани
- класификация на Pfleeger

    - функционални
    - качествени - нефункционални
    - ограничения към дизайна - решение на дизайна
    - ограничения към процеса - ограничение върху техниките или ресурсите, които могат да се използват за построяването на системата.
- Класификация:
    - Променливи/Мутиращи (Mutable) – изиквания, променящи се поради средата на системата.
    - Появяващи се (Emergent) – изисквания, които се появят в процеса на по-добре разбиране на системата.
    - Последващи (Consequential) – изисквания, които прозтичат от представянето на системата.
    - Съвместими (Compatibility) – изисквания, зависещи от други системи и организационни процеси.

Видове изисквания:

- функционални - описват желаното поведение под формата на сървиси, задачи или функции, които системата трябва да изпълнява. Дефинират границите на решението на проблема.
- нефункционални - описват ограничения върху системата или разработката. Описват изискваните свойства или качества на системата. Има 2 типа:
    - Observable at runtime
    - non observable at runtime
- Изисквания на приложната област - произлизащи от приложната област на системата, които отразяват спецификите ѝ.
- бизнес изисквания - представят целите от високо ниво на организацията или клиента, който поръчва системата. Описват защо организацията имплементира системата и целите, които желае да бъдат постигнати. Представляват бизнес deliverable, които осигуряват стойност. Формулират се от перспективата и на езика на бизнеса или потребителя.
- потребителски изисквания - описват целите или задачите, които потребителите трябва да могат да изпълняват с продукта - какво потребителят трябва да може да прави със системата.
- продуктови изисквания - изискванията от гледна точка на дефинирания продукт, който е един от начините за задоволяването на бизнес изискванията. Осигуряват стойност единствено, ако отговарят на истинските бизнес изисквания.
- системни изисквания - описват изискванията от високо ниво за продукт, съдържащ няколко подсистеми. Структуриран документ с детайлно описание на системните функции, услуги и оперативни ограничения.
- процесни изисквания - ограничения върху процеса на разработка. Обикновено се включват при големи организации с установени стандарти и практики.
- външни изисквания - изисквания, които могат да са наложени върху продукта и процеса, и са получени от средата, в която се разработва системата.

Видове нефункционални изисквания:

- класификация според IEEE
    - performance
    - interface
    - operational
    - resource
    - verification
    - acceptance

  - documentation
  - security
  - portability
  - quality
  - reliability
  - maintainability
  - safety
- класификация според Sommerville
  - подуктови - Изисквания, които определят поведението на продукта, като време за изпълнение, надежност и др. (delivery, implementational, standards).
  - процесни - Изисквания, които са следствие на практиката и процедурите в дадена организация, като прилаганите стандарти при процеса на разработване, изисквания към имплементцията и др. (usability, reliability, safety, efficiency(performance, space), portability, capacity).
  - външни - Изисквания породени от външни за системата и процеса на разработване фактори, като изисквания за оперативна съвместимост и нормативни (законодателни) изисквания и др. (legal, economic, interoperability(privacy, safety), ethical)

Заинтересовани лица (stakeholders):

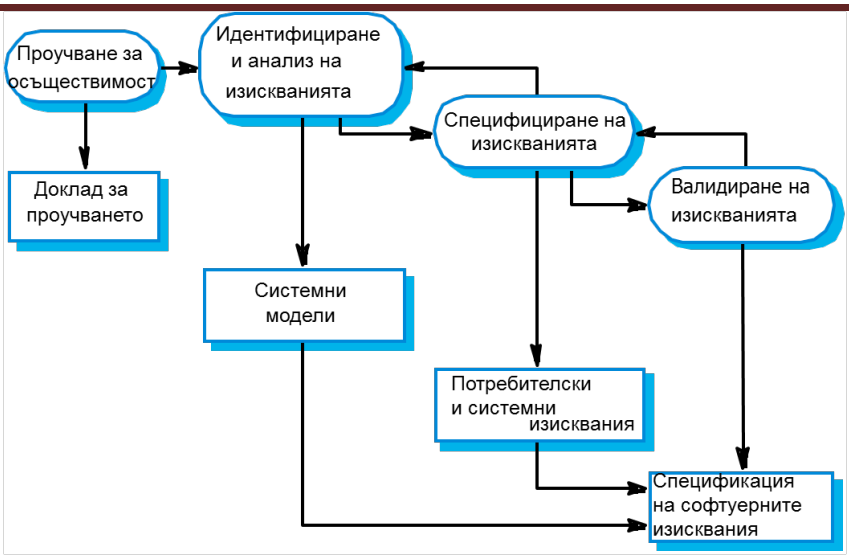
- клиенти, които плащат за разработката
- клиенти, които купуват софтуера след като е разработен
- потребители
- експерти в приложната област, запознати с проблема, който софтуерът трябва да автоматизира
- адвокати и одитори
- софтуерни инженери и др. технологични експерти
- хора, проучили пазара, за да опраделят бъдещите тенденции и потенциалните нужди на клиентите.

Потребители

Те са различни и трябва да се обединят в няколко категории. Всяка категория има свой изисквания, които трябва да бъдат взети впредвид.

3) Етапи (дейности) на инженеринга на изискванията.

18. Инженеринг на изискванията като част от системния и софтуерен инженеринг.



Етапите при инженеринга на изискванията са:

- проучване за осъществимост (feasibility study) – решават дали предложената система си заслужава. Събират информация относно дали е възможно да се постигнат целите на проекта, дали системата би допринесла за целите на организацията, дали системата може да бъде интергирана с останалите системи в организацията и дали може да се постигне съгласие относно контекста на работата. Проучването за осъществимост завършва с изготвянето на доклад за осъществимост (feasibility report), който трябва да дава препоръка дали разработката да продължи.
- извличане на изисквания - състои се от разучаване на проблемната област и извличането на изискванията. Основните задачи са да се идентифицират заинтересованите лица, откриването и разрешаването на конфликти между твърденията на различните лица, идентифициране и постигане на съгласие относно границите на системата. Необходимо е да се постигне разбиране на приложната област, проблема, бизнеса и нуждите и ограниченията на системата от заинтересованите лица. Основната цел на извличането на изискванията е да се определят проблемът, който се решава, и границите на системата, която ще се разработва. В процесът по извличане на изискванията се създава чернова на документа със системните изисквания.
- Анализ на изискванията – при него се включват действия по разбирането на приложната област, събиране на изисквания, класификацията им, решаване на конфликти между тях, приотиризация и проверка на изискванията. По време на анализа могат да бъдат изготвени различни модели на системата. Проверява се дали изискванията са необходими, консистентни, цялостни и реализируеми. Целта на анализа е да откире пробеми, непълноти и неконсистентности в извлечените изисквания.
- дефиниране на изискванията - изготвяне на дефиниции и диаграми на услугите, които системата предоставя и нейните огранчиения.
- спецификация на изискванията - изготвяне на детайлно софтуерно описание, което да осигури база за дизайна и имплементацията. Формализира реултатите от извличането и анализа в документ.
- валидиране на изискванията - осигурява, че изискванията отговаят на желанията на клиента. В този етап се работи с крайния вариант на формулираните изисквания и те се проверяват за пълнота, последователност, съответствие със стандартите, конфликти между изискванията, неясни такива, технически грешки и греши в моделите на системата.
- управление на изисквания - занимава се с осигуряването на ефективен контрол върху системните изисквания и на интегритет на информацията през целия жизнен цикъл на

системата като има в предвид и промените в системата и средата й. Занимава се с управлението на промените в изискванията за системата.

4) Техники за извличане, анализ и валидиране на изискванията.

Техники за извличане и анализ на изисквания:

- Viewpoint-oriented elicitation – Заинтересованите лица представят различни гледни точни на проблема. При този подход се анализират перспективите на различните stakeholder-и. Примери за viewpoint-oriented методи са SADT, CORE.
- Сценарии – представляват описания как системата би била използвана на практика. Включват описания на състоянието на системата преди и след изпълнението на сценария, нормалния поток на събитията, какво би могло да се обърка и как се управлява, други конкурентни дейности.
- Use cases – техника в UML, базирана на сценарии, за идентиициране на участниците в интеракция със системата и самата интеракция. Множество use case-и трябва да описват всички възможни интеракции със системата
- Интервюта – инженерът по изискванията или аналистът дискутира системата с различни заинтересовани лица и изгражда представа за тяхните изисквания. Предимуществата на тази техника са, че се провеждат с различни заинтересовани лица, събира се много информация, а недостатък е, че информацията идва от множество източници.
- FAST (Facilitated Application Specification Technique) – представляват структурирани срещи между разработчиците и клиентите, целящи да определят основните изисквания. Основната идея на тази техника е да запълни дупката между представите на разработчиците за това, което ще разработват, и на клиентите за това, което ще получат.
- Soft System methods – създават неформални модели на системата. Взимат впредвид системата, хората и организацията. Не са подходящи за детайлно извличане на изискванията. Най-популярният от тях е Software Systems Methodology.
- Наблюдение и социален анализ (Observation & social analysis) – аналистът наблюдава процеса на работа на хората, които са бъдещите потребители на системата, и си изгражда представа на това как се върши работата.
- Преизползване на изисквания – включва взимането на изискванията, разработени за една система и използването им в различна система. Спестява врме и усилия, тъй като преизползваните изисквания вече са били анализирани и валидирани в други системи.
- Прототипиране – разработва се прототип на системата и се предоставяна заинтересованите лица. Има 2 типа:
    - Throw-away прототипиране – идеята на прототипът е да помогне при извличането на системните изисквания. Трябва да се разработят единствено най-неясните изсквания.
    - Еволюционно прототипиране – целят да предоставят работеща система на потребителя възможно най-бързо. Първо се разработват най-добре разбраните изиквания.
- Въпросници – информация се събира под формата на проучване. Позволява да се достигнат много хора, но е трудно да се събере контекстуална информация. Трудна е и разработката на въпросниците.
- Брейнсторминг – подходяща при неясни изисквания.
- Storyboards – идентифицират се участниците, какво се случва с тях и как при използването на системата.
- Role playing – позволява на заинтересованите лица да изпитат света на потребителя от неговата перспектива.

Техники за валидиране:

- Преглед/рецензия(review) на исканията – най-често използваната техника за валидиране. Формира се специална група, която да се запознае с изискванията, да ги анализира, да формулира забелязани проблем и да дискутира възможностите за справяне с откритите проблеми. Необходимо е екипът, извършващ прегледа да включва хора от различни специалности – като са задължителни поне един експерт в приложната област на системата и 1 краен ползвател.
- Прототипиране – създава се прототип, който да демонстрира изискванията и да помогне на ползвателите да открият проблемите. Тази техника си заслужава само в случай, че при извличането на изискванията вече е разработен прототип. Прототипът за валидиране трябва да е пълен, ефикасен, надежден и дуракоустойчив.
- Валидиране на модел на системата – етап от процеса на валидиране. Цели да провери дали всеки модел на системата е логичен и цялостен, дали отделните модели са съвместими и дали те представят реалните изисквания на ползвателите на системата. Една от техниките за валидиране на модел е перефразиране на модел- изискванията да бъдат систематично преведени на описания на естествен език. Има 3 вида перефразиране:
    - Шаблони – от моделите се генерира перефразирано описание, позволяващо да се открие липсващата информация.
    - „Автоматизирани“ шаблони – обикновено се използват Case tool-ове, които докладват за проблеми в модела.
    - Математическо доказателство – използва се при използването на формални спецификации и модели.
- Тестване на изискванията – всяко изискване трябва да бъде тестируемо и невъзможността да се създаде тест говори за липсваща или неясна информация в описанието на изискванията. Всяко функционално изискване трябва да бъде анализирано с подходящ тест.

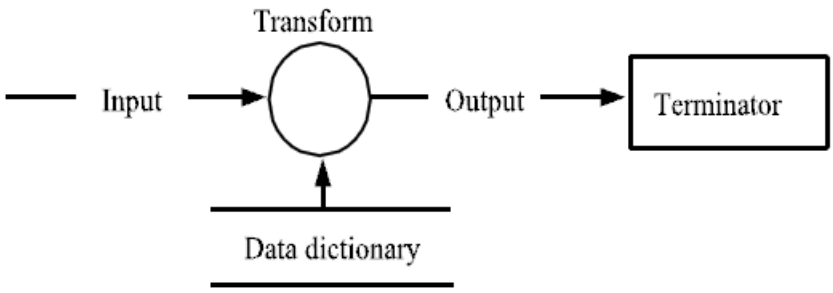
5) Основни методи за моделиране

Процесът на инжинеринг на изискванията обикновено е съпроводен от метод за моделиране. Тези методи представляват систематични начини за изготвяне на системни модели.

Data-flow Модели

Data-Flow моделите се базират на идеята, че системите могат да бъдат моделирани като множество от взаимодействащи си функции. Използват data-flow диаграми (DFD) за графично представяне на външните същности (entities), процеси, data-flow и data stores. DFD имат важна роля при определянето на данните, достъпни в даден компонент, чрез което помагат за разбирането на това какво трябва да прави компонентът и как ще завършва задачите си. DFD показват потока на данни между множество компоненти, реализиращи функционалността на системата. Актьорите не се включват в DFD. Последователността на действията може да бъде заключена от последователността от activity boxes.

Нотацията при DFD е следната:



- стрелките прдставят данните
- кутиите са процеси

Data-flow подходът се представя от метода за структурен анализ. При него има 2 стратегии:

- стрaгетия на ДеМарко - възприема top-down подход, при който аналистът се опива да съпостави текущата физическа система на текущия логически data-flow модел.
- модерен структурен анализ - прави разграничение между нуждите на потребителя и изискванията за външното поведение на системата, задоволяващи тези нужди.

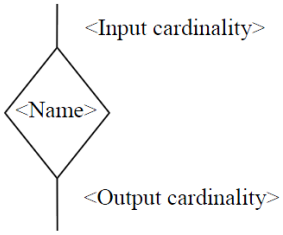
Семантичен модел

Използва се за описване на логическата стуктура на данните в системата. Идентифицира същоностите (entities) в базата данни, атрибутите им и връзките между тях. Използва графични нотации. Подходите към семантичното моделиране на дните включват Entity-relationship модел, RM/T, SDM.

Нотацията за семантичните модели надани е следната:

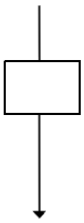
- същност (entity)

- релация между същности



A relation between entities

- релция за наследяване



An inheritance relation

Обектно-ориентираният модел

Обектно-ориентираният модел е базиран на идеята, че системите могат да бъдат моделирани като множество от взаимодействащи си обекти. Основния подход при обекно-ориентираната методология е да се разработи обектен модел, описващ приложната област. Основната цел е моделирането на приложната област, а не проектирането на имплементацията. Следните основни правила важат за обектните модели на изискванията:

- всички същности от реалния свят, важни за разбирането на приложната област, трябва да бъдат включени
- всички методи и атрибути, важни за разбирането на приложната област, трябва да бъдат включени

- обектите, атрибутите и методите, които са важни за имплементацията, НЕ трябва да бъдат включени.

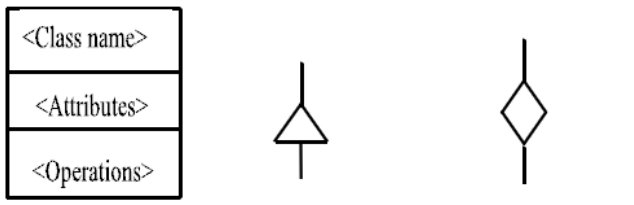
Обектните модели представят същности и връзките между тях.
При този подход се моделират обектите, класовете и операциите, които те предоставят. Основните понятия са:

- обект – идентифицират се при анализа на приложната област. Обектите са нещо реално или абстрактно, за което се съхраняват данни и операции, които манипулират тези данни. Могат да се състоят от други обекти.
- клас – имплементация на тип обекти, които съдържат общи атрибути и операции. Обектите са инстанции на класовете.
- Операции и методи – използват се за четене и манипулация на данните на обект. Реферират единствено структурите от данни на обектен тип. За достъп до данните на друг обект, обектите трябва да изпратят съобщние до него. Методите описват начин, по който операциите са реализирани в софтуера.
- Енкапсулация – групирането на данни и операциите, манипулиращи тези данни. Предодвратява неоторизирания достъп до данните на обекта и скрива детайлите за изпълнението на операцията.
- Наследяване
- Съобщения – обектите комуникират помежду си чрез изпращането на съобщения. Съобщенията се състоят от името на получаващия обет, операцията, който да бъде изпълнене, параметри. Получаването на съобщение предизвиква извикването на операция и изпълнението на подходящия метод върху получаващия обект.

При използването на обектно-ориентирания модел се изпълняват следните стъпки:

- Идентифициране на основните (core) обекти.
- Конструиране на обекните структури, дефиниращи асоциациите между класовете.
- Дефиниране на атрибутите на всеки обект.
- Определяне на операциите за всеки обект. – могат да се идентифицират чрез сценарии и use cases.
- Дефиниране на съобщенията, които се предават между обектите.

Нотацията, която се използва при обектно-ориентирания модел е:



Всяка кутия представя вид обект. Името, атрибутите и методите на обекта са изброени в кутията. Стрелка между 2 обекта представя връзка между тях. Стрелките могат да са именувани с името на асоциацията и във всеки край може да се задава число, указващо колко различни асоциации от този тип са позволени. Има 3 типа връзки:

- Наследяване – обектът от долу е специален случай на този отгоре. Представя “is-a” връзка.
- Агрегация – задава, че единият обект е компонент на другия. Представя “part-a” връзка.
- Асоциация – един от обектите асоцииран с другия.

Формални методи

Формалните методи се базират на математически принципи и целят да постигнат голяма убеденост, че системата ще отговаря на спецификацията си. Състоят се от:

- синтаксис – дефинира нотацията, чрез която се представя спецификацията.
- семантика - помага за дафинирането на обектите, които ще бъдат използвани за описанието на системата.
- релации – дефинират правилата, по които се посочва кои обекти отговарят на спецификацията.

Формалната спецификация използва формално дефиниран модел, за да описва поведението на софтуера. Описанията обикновено попадат в 3 категории:

- precondition – твърдение, което трябва да е врно преди изпълнението на функция.
- post-condition - твърдение, което трябва да е врно след изпълнението на функция.
- invariant – условие, което винаги е вярно преди и след изпълнението на функция.

Формалните методи се делят на 2 категории:

- Нотации, базирани на модели (model-based notations) – Z, VDM.
- Нотации, базирани на process algebras – LOTOS, CSP, CCS.

Обикновено се прилагат в проекти, в които сигурността и безопасността са критични.
Предимствата на формалните методи са, че те премахват двусмислеността, насърчават по-голяма точност в началните етапи на разработка на проекта и позволяват верифицирането на коректността, неточността и противоречивостта на спецификацията. Недостатъците са, че чрез тях е трудно да се представят поведенческите аспекти на проблема, не адресират проблема за дефинирането на изисквания, не е поддържат от достатъчно tool-ове.

18. Инженеринг на изискванията като част от системния и софтуерен инженеринг. Представяне на техники, прилагани на отделните етапи от инженеринга на софтуерните изисквания. Основни методи за моделиране.
Изложението по въпроса трябва да включва следните по-съществени елементи:
1) Цел и задачи на инженеринга на софтуерните изисквания.
2) Параметри на процеса – видове изисквания, класификация, потребители.
3) Етапи (дейности) на инженеринга на изискванията.
4) Техники за извличане, анализ и валидиране на изискванията.
5) Основни методи за моделиране – Data-flow модели, семантични модели, обектно-ориентирани модели, формални модели.

1. Инженеринга на софтуерните изисквания е свързан с идентифицирането на целите на софтуерната система и контекста, в който тя ще бъде използвана

Инженеринга на изискванията е процеса на разбирането и дефинирането на услугите, които ще предоставя системата и идентифицирането на ограниченията към системата. Това е ключова фаза от софтуерния процес, тъй като грешки на този етап водят до грешки в дизайна и имплементирането на системата, което води до загуба на време и пари. Изходния продукт от този етап е документ със спецификацията на системата. Целта на инженеринга на изискванията е да се създаде документ с изискванията към системата. Този етап от разработката на една софтуерна сиситема трябва да идентифицира проблема – кои проблеми трябва да бъдат решени, къде е проблема (разбирането на контекста), на кой е проблема (идентифицирането на заинтересованите лица), защо е нужно разрешаването му (идентифицирането на целите на заинтересованите лица), как софтуерната система би помогнала (събиране на различни сценарии), кога трябва да бъде разрешен (идентифицирането на ограничения за разработването), какво би попречило разрешаването му (идентифицирането на рисковете). Целия процес по инженеринга на изискванията се състои от четири подпроцеса- анализ/проучване за осъществимост на системата (feasibility study), откриване на изискванията – извличане и анализ(elicitation and analysis), преобразуване на изискванията в някаква стандартна форма (specification) и проверка за това дали изискванията наистина отговарят на това, което потребителя очаква от системата (validation). Влияние върху процеса на инженеринга на изискванията имат всички заинтересовани лица: клиентите – тези, които плащат за да бъде разработена системата и тези, които закупуват системата след като тя бъде разработена, потребителите – ще използват системата, експертите от приложната област, които са запознати с проблемите, които трябва да бъдат решени от системата, маркетинг експерти, които са проучили нуждите на клиентите, адвокати и одитори и софтуерни разработчици.
Какви са възможните източници на изисквания: хората – всички заинтересовани лица, системи – с какви други системи ще си взаимодейства нашата система и от документи – маркетингови проучвания, стандарти, анализи за приложната среда и други.

2. Изискванията трябва да бъдат коректни (correct), консистентни (consistent), ясни (unambiguous), пълни (complete), изпълними (feasible), приложими (relevant), тестериуеми (testable) и с възможност за проследяване (traceable).

В зависимост от нивото на описание на изискванията те биват, потребителски изисквания и системни изисквания.

Потребителските изисквания на софтуерната сиситема трябва да описват както функционалните така и нефункционалните изисквания и да бъдат разбираеми за потребителите на сиситемата. Те трябва да описват само външното поведение на сиситемата без да се включва информация за дизайна. Не трябва да се използва софтуерен жаргон, нотации или да се включва информация за имппламентацията на системата. Потребителските изисквания трябва да са на достъперн език, представени чрез таблици или диаграми. Могат да възникнат следните проблеми при описанието на изискванията: няколко различни изисквания могат да бъдат описани като едно изискване, липса на достатъчно яснота по време на описването на изискванията, различните видове изисквания не са описани достатъчно ясно и не могат да се разграничат.

Потребителските изисквания са с по-високо ниво на абстракция, докато системните са детайлно описание на това какво ще прави системата.Потребителските изисквания са твърдения на достъпен език и диаграми за това какви услуги ще предоставя системата и ограниченията, при които трябва да работи.Системните изисквания са детайлни и описват точната функционалност и ограничения на системата.

Системните изисквания са разширена версия на потребителските, използват се от софтуерните инженери като отправна точка за правенето на дизайна на системата. Те са по-детайлни и изясняват как потребителските изисквания ще бъдат внедрени в системата. Системните изисквания също трябва да бъдат написани на достъпен език, въпреки че това може да е трудно.

Друг начина за класифицирането на изискванията е на функционални и нефункционални и такива произлизащи от приложната област (бизнеса).Функционалните описват услугите и функциите, които системата трябва да предоставя на потребителите. Какво системата ще прави при определени условия или входни и изходни данни. Функционалните изисквания трябва да бъдат пълни и консистентни. Пълни означава, че всички услуги, които потребителя очаква от системата трябва да са описани. Консистентни означава, че изискванията не трябва да бъдат дефинирани противоречиво Нефункционалните включват ограничения към системата или процеса на разработването и.

Нефункционалните изисквания са свързани с качеството на системата. Могат да се разделят в следните категории. Изисквания произтичащи от продукта - описват

поведението на продукта. Колко бързо системата ще изпълнява определено действие, колко памет е необходима, колко е надеждна системата, portability, usability
Организационни - произлизат от политиките или процедурите в организацията на потребителите или разработчиците. Стандарти, които трябва да бъдат спазени, език за програмиране, който трябва да се използва, дизайн, кога продукта и документацията трябва да бъдат завършени. Външни - произтичат от всички фактори, които са външни за системата или за процеса по разработването и. Например как системата ще си взаимодейства с вече съществуващите системи на клиента, етични изисквания...

Нефункционалните изисквания трябва да се описват с метрики, така че да могат да се тестват Нефункционалните изисквания често са в конфликт или зависят от функционалните.
Бизнес изискванията могат да бъдат както функционални така и нефункционални и се извлича от приложната област към, която ще принадлежи системата. Описва защо има нужда от тази система или какво иска да постигне чрез системата. Този тип изисквания идват от спонсорите на проекта или от мениджърите и са изисквания на високо ниво.

Устойчивост на изискванията (stability)
- постоянни изисквания – такива, които не се променят по време на разработката на системата
- непостоянни (изменчиви) изисквания – изисквания, които се променят по врене на разработката или, когато системата влезе в употреба

Приоритизиране на изискванията:
- изисквания, които задължително трябва да бъдат изпълнени
- изисквания, които са желателни
- изисквания, на които е възможно да отговаря системата, но могат да бъдат пренебрегнати

3.
Основните етапи при инженеринг на изисквания са: извличане на изискванията, анализ, валидиране и управление. **Извличането** на изискванията става с помощта на клиентите, открива се приложната област, услугите, които ще предоставя системата и какви са ограниченията. Дейностите свързани с извличането са: откриването на приложаната среда – къде системата ще бъде използвана, разбиране на проблема (нуждите), за които системата ще бъде разработена, разбиране на бизнес средата, в която системата ще бъде използвана и разбиране на нуждите и ограниченията на заинтересованите лица. По врема на тази фаза възникват проблеми породени от факта, че заинтересованите лица не винаги знаят какво искат, изразяват се със собствени термини, по време на разработването се появяват нови заинтересовани лица, организационни и политически различия. Изходния продукт от тази дейност е чернова с изискванията.
Фазата на **анализа** на изискванията проверява дали всички изисквания са нужни и необходими за разработването на сиситемата. Прави се проверка дали изискванията са

пълни и консистентни – изискванията трябва да не бъдат противоречиви и да бъде описано всичко за сиситемата. Проверява се дали са изпълними спрямо бюджета и графика за разработването на системата. Обсъждат се със заинтересованите лица и се постига съгласие. Изискванията се приоритизират, като се взима предвид кои от тях са с критична важност за системата. По време на анализа е неободимо да се отговори на следните въпроси: дали има изисквания за предварителен дизайн иии информацията за внедряването, дали дадено изискване може да бъде разделено на няколко подизисквания, дали всички описани изисквания наистина са необходими, дали изискванията не са двусмислени – могат да се разбират по различен начин от различните заинтересовни лица, дали изискванията са реалиситчни, дали изискванията са тестеруеми

Валидацията на изискванията удостоверява, че документът, описващ изискванията е, приемливо описание на системата, която ще бъде реализирана. Документът на изискванията се проверява за пълнота и последователност, съответствие със стандартите, конфликт в изискванията, неяснота на изискванията, технически грешки. Валидирането работи с вече подготвени и съгласувани изисквания. Входна инфрмация за процеса на валидирането е документ на изискванията,знание за организацията и организационни стандарти – трябва да бъде проверено спазването на локални стандарти. Документът с изискванията трябва да е цялостна завършена версия на документа. Изготвянето и форматът на документа е съобразен със стандартите на организацията.Знанието за организацията включва – използвана терминология, умения на работещите, култура и т.н. Изходната информация от процеса на валидирането е списък на откритите проблеми в описанието на изискванията – откритите проблеми се класифицират според типа им и списък на действията, които трябва да се извършат за коригирането на изискванията.

4) Техники за извличане, анализ и валидиране на изискванията
Техники за извличане и анализ на изискванията: интервюта, сценарии, FAST, Soft System Methods, наблюдение, преизползване на изискванията
Формални или неформални *интервюта* със заинтересованите лица е една от техниките за извличане на изискванията. Тя се използва също и при анализа. Екипа по инженеринга на изискванията задава въпроси на заинтересованите лица за системата, която трябва да се разработи. Изискванията се извличат от отговорите на поставените въпроси. Интервютата биват затворени, отворени или смесен тип. При затворените интервюта заинтересованите лица отговарят на набор от предварително подготвени въпроси, докато при отворените интервюта се задавт въпроси, които не са предварително подготвени, а възникват по време на интервюто в зависиост от дадените отговори. В повечето случаи се използва смесения тип интервюта. Отговора на един въпрос води до друг въпрос и така до пълното изясняване на изискванията към системата. Интервютата са подходяща техника за придобиване на цялостна информация за системата. За да бъдат полезни заинтересованите лица трябва да бъдат открити и да им се даде отправна точка за дискусия.
Сценарии:

Сценариите предтавляват релален пример за това как системата ще бъде използвана. Те трябва да съдържат описание на това което системата и потребителите очакват, когато системата се стартира; описание на нормалния поток от събития; описание на това какво може да се объркa (повреди); информация за други конкурентни дейности; описание на състоянието на системата, когато сценария свършва

Use Case:

Това е техника базирана на сценариите но се използва обектен подход за описването и.

- Use-cases are a scenario based technique in the UML which identify the actors in an interaction and which describe the interaction itself.
- A set of use cases should describe all possible interactions with the system.
- Sequence diagrams may be used to add detail to use-cases by showing the sequence of event processing in the system.

Facilitated Application Specification Technique

FAST

Срещите се състоят на неутрална среда, има правила за провеждане на срещтата, могат да бъдат формални и неформални…

- Meeting at neutral site
- Rules for preparation & participation
- Agenda – formal + informal
- Facilitator controls meeting
- Definition mechanism
- Identify problem, propose solution, negotiate approaches, specify set of solution requirements

Soft System Methods

- These produce informal models of a socio-technical system. They consider the system, the people and the organisation
- Not techniques for detailed requirements elicitation. Rather, they are ways of understanding a problem and its organisational context
- Software Systems Methodology (SSM) is probably the best known of these methods
- The essence of SSM is its recognition that systems are embedded in a wider human and organisational context

Stages of SSM

- Problem situation assessment
- Problem situation description
- Abstract system definition from selected viewpoints
- Conceptual system modelling
- Model/real-world comparison
- Change identification
- Recommendations for action

Наблюдение и социален анализ: Хората често изпитват затруднения да опишат какво правят в своята работа, защото това се е превърнало за тях в нещо естествено. Затова добър начин да разбереш с какво е свързана работата им е наблюдението по време на работата. Това помага да се разбере работния процес и действията, които извършват хората и да се изгради представа за техните нужди.

Преизползване на изискванията:
Взимат се изисквания, които са били открити за една система и се използват за друга. Преизползването на изискванията спестява време и усилия, тъй като изискванията, които се използват вече са били валидирани и анализирани за предишната система.
Прототипиране: изгражда се прототип, който да извлече или валидира изискванията
Има два вида прототипа:
- throw away помага за извличането на изискванията, изискванията които се включват в прототипа са тези, които са най-трудни за разбиране. Отразява най-неясните изисквания към системата.
- evolutionary prototyping – доставя на потребителите работеща система, в него се включват изискванията, които са лесни за разбиране и могат да предоставят на крайния потребител желаната от него функционалност.
Прототипирането може да доведе до загуба на време, ако системата не е сложна и изискванията са достатъчно ясни. Могат да бъдат използвани много ресурси (време, пари), а прототипът да не задоволява нуждите на клиентите.

Техники за валидиране: преразглеждане/преглед/рецензиране на изискванията. Формира се специална група, която да се запознае с изискванията и да ги анализира; формулира забелязани проблеми; дискутира възможните действия за справяне с проблемите.
Дейностите сврзани с прегледа на изискванията са: **Планиране -** Определя се екипа, ръководител и протоколчик; Избира се време и място за срещите; **Разпространение** на документите за преглед до участниците в екипа; **Подготовка** за прегледа - Всеки рецензент да прочете документа за изискванията и да набележи открити конфликти, пропуски, непоследователности, отклонения от стандартите, др. проблеми; **Среща по прегледа** - Дискутират се коментарии и проблеми на отделни участници от екипа и се дефинира набора от възможни действия за справяне с проблемите; **Проследяване на действията** - Ръководителят на екипа проверява дали формулираните действия са изпълнени; **Ревизиране на документа -** Документът на изискванията се редактира, за да отрази съгласуваните действия. На този етап той може да бъде предаден като окончарелен вариант или върнат за нов преглед.
//Действия за решаване на всеки отделен проблем:
Изясняване на изискването, защото То може да е лошо представено или да има случайно изтървана информация. Презаписване на изискването.
Липсваща информация Отговорност на инженерите по изискванията е да открият тази информация от съответен източник – ползвател или др.
Конфликт в изискванията Ако съществува значим конфликт в изискванията, то ползвателите трябва да преговарят, за да достигнат до съгласие.

Нерелистични изисквания Изискването не би могло да бъде реализирано, защото има технологични и/или други ограничение върху системата. Необходимо е ползвателите да се консултират, как да направят изискването по- реалистично. //

Участници в екипа – формирането на екипа е от съществено значение за качествения преглед

• Част от рецензентите трябва да включват ползватели с различни специалности, за да се съберат различни умения и знания;
• Участвайки в процеса на прегледа ползвателите от различни специалности могат да разберат по-добре необходимостта от промяна на изискванията;
• В екипа трябва задължително да се включи:
поне един *експерт от специфичната област* на създаваната система, както и понеедин *краен ползвател*.
Брой на експертите, формиращи екипа;
Организиране на работата на екипа.

Прототипиране
Прототипите, предназначени за валидиране на изискванията, имат за цел да **демонстрират изискванията** и **да помогнат на ползвателите да открият проблемите**. Прототипите за валидиране трябва да бъдат **пълни, ефекасни** и **робастни**. Те трябва да могат да се ползват по същия начин, по който ще се използват в исканата система. Документация и ръководство за ползвателя трябва да бъдат предоставено.Валидирането може да започне с непълен прототип, но задължително системата трябва да бъде допълнена и довъроена по време на валидирането.Надеждността на прототипа е задължителна.Техниката на прототипиране заслужава да се прилага ***само*** вслучай, че вече е разработен прототип за целите на извличане на изискванията!

Действия при прототипирането: успоредни процеси
Избор на подходящ екип за тестване на прототипа Най-подходящи са тези ползватели, които имат умерен опит и които са “отворени” към използването на нови системи. Крайни ползватели с различен характер на извършваната дейност трябва да участват, за да се изпробват/покрият различни функционалности.
Разработване на тестови сценарии Внимателното планиране предполага формулиране на пълен набор от тестови сценарии, които осигуряват широко покритие на изискванията. (End-users shouldn’t just play around with the system as this may never exercise critical system features).
Изпълнение на сценариите При изпълнение на сценариите ползвателите е добре да работят самостоятелно, като по този начин симулират и реалната ситуация на използването на системата.
Документиране на проблемите Електронен или хартиен документ с откритите проблеми е препоръчителен

5. Методите за инжинеринга на изискванията могат да бъдат категоризирани като формални, полуформални и неформални.
Полуформалните и неформалните използват обикновен (достъпен) език, таблици и диаграми. Включват структурни и обектно ориентирани анализи. Формалните методи са базирани на математически синтаксис и семантика. Такива методи са Z, B, VDM, LOTOS.
Формалните методи:
Provide a means for achieving a high degree of confidence that a system will conform to its specification; Do not absolute guarantee of correctness;
Have little directly to offer to the problems of managing software projects. However, benefits can be gained from gaining a clear understanding of the task at an early stage.
Формалните методи се състоят от синтаксис, семантика и връзка между тях. Синтаксиса показва специфичната нотация, която се използва при метода(that defines the specific notation with which the specification is represented – often based on a syntax that is derived from standard set theory notation and predicate calculus). Семантиката определя как се представят изискванията (that help to define a "universe of objects" that will be used to describe the system – how the language represents system requirements). Релацията (*Relations* which define the rules that indicate which objects properly satisfy the specification).
Формалните методи не са широко разпространени, тъй като нотацията им е трудна за разбиране, трудно с тяхна помощ се описват определени аспекти от изискванията, липсват инструменти, които да правят употребата им по-лесна. Предимства на формалните методи са, че тяхната употреба намалява двусмислеността, тъй като се използват математически формализъм, което прави възможно проверката за пълнота, коректност и консистентност. При използването на тези методи се изисква по-голяма точност и взискателност още в ранната фаза от инженеринга на изискванията. Формалните методи могат да се разделят в две категории - Model-based notations – Z and Vienna Development Method (VDM) и Process algebras -based notations - *Communicating Sequential Processes (CSP), CCS and LOTOS.*

Обектно - ориентирания подход е базиран на идеята, че системата може да бъде моделирана като набор от взаимодействащи си обекти.

Data – flow model е базиран на идеята, че системата може да бъде представена като набор от взаимодействащи си функции.

19. Интеграция на разпределени информационни системи

Нужда и дефиниция за разпределена система

За нарастващата нужда от разпределени системи се обуславя от няколко различни фактора:

- Нарастващи сливания между фирми
- Намалява се наличното време за осигуряване на необходима услуга
- Интернет осигурява нови възможности за предлагане на софтуер и услуги на много клиенти

Дефиниции

Таненбаум

Разпределената система представлява съвкупност от компютри, които изглеждат на потребителите си като една кохерентна система.

Емерих

Разпределена система е съвкупност от автономни хостове, които са свързани чрез компютърна мрежа. Всеки хост изпълнява компоненти и върху него работи разпределен middleware, който позволява компонентите да координират своите активности по такъв начин, че потребителите да възприемат системата като единично, интегрирано средство за работа.

Middleware

С усложняването на софтуера се налага и по- голяма абстракция и все по-сложни примитиви за разработка.Така се преминава от операционни системи за разпределени компютри, разпределени операционни системи към мрежови операционни системи докато се стига от нуждата от нов абстрактен слой, които да предоставя основни услуги и който има за цел прозрачност на разпределеността на системата.Този слой е middleware.

Дефиниции

Дефиниция 1:

Лепило, което държи всички компоненти на разпределените приложения заедно.

Дефиниция 2:

Компонент, който държи решението на проблема с разширяемостта чрез създаване на ниво в разпределената инфраструктура, върху което се разполагат приложенията.

Дефиниция 3:

Middleware е термин, който се отнася до множество от софтуерни услуги, което се намира между приложението и операционната система и има за цел да улесни разработването на разпределени приложения чрез абстрахиране от сложността и хетерогенността на намиращата се отдолу среда операционни системи, хардуерни платформи и комуникационни протоколи).

Функции

Middleware трябва осигурява примитиви на високо ниво, които опростяват конструирането на разпределена система Трябва да предоставя Консистентен опростен приложен програмен интерфейс (API).Трябва да спомага за оперативната съвместимост на хардуер, мрежи и програмни езици.Позволява на разработчиците да се фокусират върху изискванията , а не върху технологичната среда. Middleware-ът трябва да поддържа мрежова комуникация, да позволява синхронизация по различни начини, да се справя с хетерогенността, да бъде надежден и разширим.

Роля

За да се разбере ролята на middleware-а, трябва да бъде осъзната двойнствената му същност на **програмна абстракция** и на **инфраструктура.**

Програмна абстракция

- Има за цел да скрие детайлите от ниско ниво за мрежи, хардуер и разпределеност
- Има тенденция към все по мощни примитиви, които без промяната на концепцията на RPC, да предоставят повече свойства и по-голяма гъвкавост.
- Начинът за предоставяне на примитивите на разработчика зависят от различните програмни езици;

Инфраструктура

- Има за цел да достави изчерпателна платформа за разработка и изпълнение на сложни разпределени системи
- Има течения към SOA
- Използва се продуктовата линия на един доставчик на middleware за да се улесни разработката
- Тенденцията е към интеграцията на различните платформи и към гъвкавост чрез конфигурации

Видове Middleware

Remote procedure calls (RPC)

Осигурява инфраструктура за отдалечени извиквания на процедури. Служи за основа на почти всички останали форми на middleware.

Transaction processing monitors (TP monitors)

Могат да се разглеждат като RPC с възможност за транзакции. Класифицират се в TP-Lite (RPC интерфейс за бази от данни) и TP-Heavy (средства и функционалност подобни и дори надминаващи тези на операционните системи).

Обектни брокери
Повечето от тях използват RPC при имплементацията на отдалечено повикване на обекти.

Обектни монитори
Обединяване на TP мониторите и обектните брокери в хибридни системи.

Message-oriented middleware (MOM)
Осигурява транзакционен достъп до опашки,примитиви за четене и запис в локални и отдалечени опашки.

Брокери на съобщения
Осигуряват възможност за трансформиране и филтриране на съобщения при прехвърлянето между опашките. Динамично определят получателя на съобщението въз основа на съдържанието му.

Проблеми на интеграцията на софтуерни приложения с използването на обмен на файлове, по сокети, RPC и RMI;

Интеграция чрез обмен на файлове

Общо описание
Всяко приложение трябва да създава файлове , които другите приложения трябва да консумират.Интеграторите се грижат за това как файловете се трансформитат към различни формати.Файловете трябва да се създават на равни интервали според нуждите на бизнеса.

Положителни страни
- Не изисква промени в системите , които биват интегрирани.Подходът не е инвазивен.
- Ниско ниво на свързаност между интегрираните системи
- Сравнително прост метод

Отрицателни страни
- Неефективни при чест обмен на файлове
- Неконсистентност на данни, трудно се синхронизират файловете
- Трябва да се имплементират механизми за логическо заключване, да се използват сложни именуващи конвенции, проверки за актуалност на файла
- Проблем за местоположението на файла , когато се интегрират два бизнеса.

Интеграция чрез сокети

Общо описание

Сокет е една от крайните точки на двупосочна комуникация между две програми, които работят в мрежова обстановка.Всеки сокет е свързан с номер на порт , така че TCP слоят да може да идентифицира приложението, където информацията трябва да се предате

При интеграция софтуерни системи чрез сокети , трябва да се отбележи, чесе използват възможностите на TCP(Transmission control protocol) и UDP (User DataGram Protocol) протоколите.При този вид едната или повече от системите играят ролята на сървър и една или повече от системите играят ролята на клиент.Сървърът притежава сокет , които е свързан с определен порт на който слуша за съобщения. Клиентът трябва да знае името на хоста на машината , на която се намира сървъра и порта , на който сървърът слуша. За да осъществи връзка клиентът опитва да се свърже със сървъра през хоста и порта на сървъра.Клиентът трябва да се индентифицира като му подава локален за клиента порт , които ще бъде използван по време на комуникацията.

Положителни страни

- Удобни за имплементация на клиент-сървър комуникация
- Едновременна поддръжка на много клиенти
- Ефикасна UDP комуникация

Отрицателни страни

- Високо ниво на свързаност
- Повечето системи използват собствени формати на данните
- Трябва да се обръща твърде много внимание на жизнения цикъл на комуникационните обекти

Интеграция с RPC

Общо описание

Remote procedure calls е технология за комуникация, която позволява на една програма да извика функционалност на друга, която действа в друго адресно пространство като най-често тя физически е разположена на друг компютър. RPC скрива детайлите при комуникацията зад процедурни извиквания и подпомага взаимодействието между хетерогенни платформи.Архитектура на RPC включва:

- Клиентски стъб: парче от код, което се компилира и свързва при клиента.
- Сървърен стъб: имплементира сървърната страна на повикването.
- Кодови шаблони и референции: IDL компилатора създава всички необходими помощни файлове; може да генерира и шаблони с първичен код за сървъра.

Данните се предават като битови потоци от данни.Проблем е предаването на сложни и абстрактни типове данни.RPC предлага решение чрез:

- Marshalling – данните се трансформират до удачен за предаване вариант
- Unmarshalling – обратен процес на marshalling

Свързването е процес, чрез който клиентът създава локална асоциация (handle) за даден сървър при извикване на отдалечена процедура.Има възможност за статично и динамично свързване.

Проблемът с хетерогенността се решава от стъбовете. Възможни решения на проблема с хетерогенността:

- Използване на различни клиентски и сървърни стъбове за всяка възможна комбинация на платформи и програмни езици (2 x n x m).
- IDL – използва се за дефиниране на интерфейси и дефиниране на междинното представяне на данните обменяни между клиента и сървъра.

Асинхронно RPC

Позволява на клиента да изпраща съобщения за заявка на услуга без изчакване на отговор.Стъбът притежава две входни точки за **извикване на процедурата** и за **получаване на резултата.** При липса на резултат клиентът получава съобщение, указващо повторение на запитването. При грешка клиентът получава стойност, показваща характера на грешката.

По същество обработката в **стъбовете е синхронна**, а от гледна точка на **клиента и сървъра** изпълнението е **асинхронно**. Истинска асинхронна обработка е имплементирана в системите управляващи опашки и брокерите на съобщения.

Положителни страни

- Мощна абстракция , скрива детайлите от ниско ниво
- Справя се с хетерогенността

Отрицателни страни

- Изисква доста ресурси (naming , communication, processing, memory (stub and skeleton))
- Проблеми с времето за отговор
- Не е стандарт
- В днешно време е примитив от доста ниско ниво
- Висока свързаност (tight coupling)

Интеграция с използването на Уеб услуги

Дефиниции за уеб услуга

Дефиниция 1
Приложение, достъпно за други приложения през уеб.

Дефиниция 2 (на UDDI консорциум)
Самосъдържащи се, модулни бизнес приложения, които имат отворен, Интернет ориентиран, базиран на стандарти интерфейс.

Дефиниция 3(на W3C)
Софтуерно приложение, идентифициращо се чрез URI, чиито интерфейси е възможно да бъдат дефинирани, описани и открити чрез XML артифакти. Уеб услугата поддържа директно взаимодействие с други софтуерни агенти, използвайки XML базирани съобщения, обменяни посредством Интернет базирани протоколи.

Дефиниция 4
Стандартизиран начин за интеграция на уеб базирани приложения при използване на XML, SOAP, WSDL и UDDI отворените стандарти, основавайки се на Интернет протокол. XML се използва за форматиране на данни, SOAP за тяхното трансфериране, WSDL – за описание на наличните услуги, а UDDI – за откриването им.

Описание на уеб услуги
Основните аспекти в описанието на уеб услуги са:

- Общ мета-език
    - XML се използва като базов език за спецификация на всички езици,необходими за описание на уеб услугите.
- Интерфейси
    - Описанията на уеб услугите са по-обширни (в сравнение, например с CORBA): спецификация на адреси (URI), транспортни протоколи (HTTP).
    - Web Service Definition Language (WSDL) е базиран на XML език , който предоставя модел за описание на уеб услуги.
- Бизнес протоколи
    - Доставчиците на услуги правят опити да наложат правила за комуникация между уеб услугите и клиентите, специфицирани като част от т.нар. бизнес протокол.

- Съществуват предложения за стандартизиране на езици за дефиниране на бизнес протоколи: Web Service Conversational Language (WSCL),Business Process Execution Language (BPEL).

- Свойства и семантика
  - При избора на уеб услуга освен данни за интерфейса й клиентът има нужда и от допълнителна информация като например нефункционални свойства.
  - Спецификацията на UDDI описва как да бъде организирана информацията за уеб услугите и как да бъдат изградени хранилища за нейното регистриране и заявяване.

- Вертикални стандарти - дефинират специфични интерфейси, протоколи, свойства и семантики,които услугите, предлагани в определен домейн, трябва да притежават.

Откриване на уеб услуги

Описанията на уеб услугите се съхраняват в “директории”,които позволяват:

- регистриране на нови услуги от доставчиците
- търсене на и локализиране на определени услуги от потребителите.

Откриването на уеб услуги може да стане в дизайн режим и режим на изпълнение.(design time and runtime)

Директориите могат да бъдат хоствани и управлявани централизирано и P2P базирано. UDDI спецификацията предоставя стандартни APIs за публикуване и откриване на информация в директориите за услуги.

Протоколи за взаимодействие:

Транспортен протокол

мрежовата комуникация на уеб услугите е скрита зад транспортни протоколи като HTTP.

Протокол за обмяна на съобщения

Simple Object Access Protocol (SOAP) стандартизира начина, по който обменяната информация се форматира и пакетира, т.е. дефинира шаблон за съобщения.

Мета протоколи

- Улесняват и координират прилагането на бизнес протоколите.
- WS-Coordination е спецификация, която прави опит да стандартизира мета-протоколите и начина, по който WSDL и SOAP трябва да бъдат използвани за предаване на информация, релевантна на избрания протокол.

Хоризонтални (middleware) протоколи

- Уеб услугите и поддържащата ги инфраструктура са разпределени, поради което основната middleware комуникация се постига посредством P2P протоколи, наречени хоризонтални протоколи.
- Скрити са от разработчиците и потребителите и се управляват напълно от инфраструктурата.
- WS-Transaction протокола дефинира как да бъдат имплементирани транзакционни свойства (базиран е на WS-Coordination).

Композиция на уеб услугите

- Съставна услуга: уеб услуга имплементирана посредством извикването на други уеб услуги.
- Базова услуга: уеб услуга, достъпваща директно локалната система.

Съставните и базовите услуги трябва да се описват, откриват иизвикват по един и същ начин.

Уеб приложения и технологии за поддръжка на отдалечени клиенти

Аплети

Един от първите решения за пренасяне на динамично съдържание по мрежата са аплетите.Аплетите представляват Java програма, която е вградена в HTML страница. Когата страницата бъде свалена от браузъра, Java виртуалната машина започва да изпълнява кода.

Предимства

- Предимства предоставя част от предимствата на десктоп приложенията
- Предоставя по-голяма динамичност
- Можем да ги използваме за по-сложни клиенти
- Изпълними за всички операционни системи, за които има java виртуална машина.

Недостатъци

- Сигурност – от съображения за сигурност , аплетите се изпълняват в sandbox, с ограничени възможности за взаимодействие с локалната операционна система
- Двоичният java код трябва да се сваля всеки път , когато приложението се достъпи.

Common Gateway Interface

След като аплетите не са подходящи за по-сложни клиенти , следващият подход е да се гледа на уеб сървъра като на интерфейс.Т.е уеб сървърът може да отговори на заявка след като изпълни дадена програма, която ще генерира съответният документ , които да бъде върнат от сървъра.

CGI е стандартен механизъм, които дава възможност на HTTP да си сътрудничи с външни приложения които са като шлюз за локалната информационна система. CGI съпоставя на всяка програма URL , а програмата получава своите параметри чрез параметрите на HTTP заявката.

Предимства:

- Поддържат се различни езици за програмиране
- Поддържат се интерфейси към бази данни

Недостатъци:

- Ако се използва за интеграция на ниво http- няма стандарт , които да преписва синтаксис и семантика на данните (HTML не е задължителен , да не говорим за XML или JSON)
- Поддръжката на много платформи може да е проблем.
- Проблеми с производителността - всяка заявка се изпълнява в совствен процес.Няма спомагателен жизнен цикъл на приложенията.

Сървлети

За да се избегнат проблемите с производителността на CGI, могат да се използват сервлети.Зад тях стои същата идея като за CGI , но има разлика в имплементацията.Първата разлика е , че сървлетите не вървят в собствени процеси , а в собствени нишки. Нещо повече те са част от уеб сървъра, т.е се спестява междупроцесната комуникация в рамките на една и съща операционна система. Тази архитектура позволява допълнителни оптимизации , като имплементирането на кеш, които трудно би риализируем – всяка програми трябва да имплементира свой кеш.

Подобни технологии –ASP, ASP.NET

Предимства

- Имат свой собствен жизнен цикъл, позволяват на приложния програмист да се съсредоточи върхи бизнес логиката.
- Презиползването на инстанциите на сървлетите
- Надграждането с различни технологии
- Всички REST –full услуги в java се базират на Java Servlet Specification
- Могат да се използват за диспачери , филтри

Недостатъци

- Все още имаме примитиви от ниско ниво
- Неудобно е да се използват за писане на Mark-up (HTML).Решение - JSP, JSF

Проблеми:
- файлове
- различни формати
- трудна синхронизация
- конвенции за именуване, правила за заключване, проверки за актуалност
- проблеми за местоположението
- често обмен - неефективно

Но е прост, не се налагат промени в приложенията

- сокети
- трябва да има синхронизация м/у клиента и сървъра (двустранна връзка, поддържа се през цялия процес на обмен)
- трябва
- еднакъв формат на данните

- RPC
- изисква доста ресурси
- проблем с времето за отговор (през мрежа)
- не е стандарт (≠ платформи, ≠ езици)

- RMI
- платформено и езиково ~~X~~ зависим

20. Планиране на проект – обхват на проекта, структурата на работа по проекта, време за изпълнение на задачите и създаване на план график, планиране на ресурси, разходи и бюджет на проекта

1. Планиране на проекта – обхват на проекта и определяне на структурата на работа по проекта (WBS)
2. Планиране на проекта – време за изпълнение на задачите, мрежови диаграми
3. Методи и средства за създаване на график. Критичен път. Метод на критичния път. Метод PERT. GANTT диаграми. Създаване на план - график на проекта
4. Планиране на проекта – ресурси, разходи и бюджет на проекта

1. Планирането на проекта продължава и детайлизира направеното по време на инициалиазицацията на проекта и отразява определени факти и действия, които се извършват в процеса на изпълнение на проекта.

Съдържанието на план за управление на проекта обикновено включва следните компоненти:

- Въведение
    - Цел на плана – описва целта на плана на проекта (напр. Дефиниция на проекта, създаване на план-график и др.)
    - История на проекта до този етап (изминало време от стартирането на работата по инициализация на проекта, очаквано влияние на закъснението (ако има такова), предишни дейности, промени в средата за изпълнение на проекта и др.)
    - Подход за управление на проекта (основен метод за управление на проекта, основни изисквания към хората, работещи в екипа на проекта, мотивационни техники (ако са предвидени такива) и др.)
- Бизнес цели на проекта
    - Намаляване на разходите
    - Намаляване на броя на грешките при работа
    - Намаляване на времето за обработка на информацията
    - Увеличаване капацитета на информацията и производителността на работа
    - Намаляване риска при работа с информацията
    - Отговор на законови изисквания (напр. Въвеждането на електронния подпис и др.)
    - Осигуряване на по-добър достъп до информацията при взимане на решения
- Основни резултати (изброяват се основните резултати, очаквани от проекта, връзката им с бизнес целите на проекта, както и интервалът за реализирането им)
- Обхват на проекта. Описва функциите на продукта или услугата, която ще се реализира от проекта, и идентифицира работата, необходима за изпълнението му. Целта на това описание е да се създадат колкото може по реалистични задачи за работа, план-график, бюджет и очаквания по проекта и се определи работата, която е извън обхвата на проекта. Това описание се развива динамично по време на проекта и може да се промени при одобрена формална промяна на обхвата на проекта.

- Дефиниция на обхвата. Определя работата, която трябва да се извърши по проекта, и доколкото е възможно на този етап – какво не се включва в проекта. Дефинират се архитектурата и технологията за изграждане на продукта, както и методът на неговото внедряване. Добра практика е по време на дефиницията на обхвата да се определят и критерии за работата, която ще се свърши по време на поддръжката на системата или приложението (напр. Поправяне на грешките, които не са критични или работа по част от функционалността)
- Разходи и ползи. Описва разходите по проекта, вкл. разходите за управление и администрация
- Подход за управление на риска. Включва описание на идентифицираните рискове, тяхната приоритизация и начините за тяхното решение (съответните коригиращи действия в зависимост от влиянието на риска). Описва и периода, когато се прави нова оценка на риска (напр. преди започването на всяка фаза), както и периода за отчитане на риска (напр. двуседмично или месечно)
- Списък на доставките на проекта (deliverables list). Изброява основните формални доставки по проекта (формални резултати, очаквани да се доставят), вкл. доставките на управлението на проекта и неговата администрация
- Контролни точни (milestones). Това са времеви точки, в които завършват една или повече задачи и се доставят един или повече важни резултати от проекта. Може да съвпаднат с края на фазите от жизнения цикъл за изграждане на системата и се използват за мониторинг и контрол на прогреса по проекта. Ако прогресът на проекта, измерен в тези точки не е планираният, тогава има немалка вероятност проектът като цяло да не завърши, когато е планирано
- Предположения. Засягат главно предположения за ресурсите, обхвата, очакванията, графика на доставките, които трябва да се осъществят, за да се изпълни успешно проекта.
- Ограничения. Обикновено основните ограничения обхващат:
  - Ограничения на средата за реализация на проекта, вкл. наличието на обучени ресурси
  - Свързани проекти, които влияят на реализацията на този проект
  - Критични зависимости между отделните задачи, необходимите ресурси и може да се опише и критичният път на проекта
- Подход за управлението на качеството. Включва:
  - Прегледи на изпълнението на задачите
  - Техники и средства
  - Подходи за тестване
  - Стандарти, които ще се използват (напр. ИСО 9001:2000, CMMI)
  - Критерии за приемане
  - Роли в управлението на качеството (мениджър по качеството, инженер по качеството, тестер)

- Обучение на клиента/потребителите. Описва подхода за обучение, както и мястото/местата за неговото въвеждане
- Подход за управление на проекта. Състои се от:

- План-график на проекта, изразен чрез структура за работа по проекта (Work breakdown structure WBS) и/или диаграма на Гант (Gantt Chart), в която са отразени ресурсите по проекта
- Оценка на работата по проекта. Описва подхода използван при оценката на работата и включва следните категории:
    - Работа на хората от екипа на проекта;
    - Административни ресурси, свързани с управление на клиента, участие в срещи на управляващия комитет или в други дейности по управление и администрация на проекта
    - Материални ресурси (софтуер и хардуер, помещения, комуникационни средства и др.)
    - Стандарти и методи (напр. PRINCE, EU PHARE, IPMA)
- Изисквания към ресурсите, ролите и отговорностите. Включва също така знания в областта на съответната технология, знания в съответната бизнес област, както и опит в предишни проекти.
- Указател на контактите
- Подход за управление на промените и спорните въпроси (Change and Issue Management Approach). Описва процедурите за обработка на искания за промяна (change requests) и тяхното одобряване/отхвърляне
- Управление на комуникациите. Описва процедурите за комуникация между заинтересованите лица и вътре в екипа на проекта, какви отчети, кога и на кого ще се изпращат.
- Управление на конфигурацията. Описва процедурите за управление на различните версии на продукта и неговата документация
- Приложение към плана на проекта. Към проекта може да се приложат различни документи, които са били създадени по време на планирането на проекта, а също така и различни шаблонни форми на документи, които се изискват при изпълнението на проекта
- Одобрение и упълномощаване (authorization). Подписите на ръководството на организацията – изпълнител, спонсор и други упълномощени лица

2. Обхват и структура на работа по проекта

Терминът обхват има различни значения, когато се отнася за продукта или за проекта:

- Обхват на продукта (product scope) – описание на характеристиките и функциите на продукта или услугата;
- Обхват на проекта (project scope) – работата, която трябва да се извърши за създаването на продукта или услугата;

Планирането на обхвата на проекта има за цел създаването на писмен план за обхвата на проекта и на неговия продукт, служещ за бъдещата му реализация. Дефиницията на обхвата става чрез разделянето (декомпозицията) на основните

резултати от проекта на по-малки и по-управляеми компоненти, така се създава специфична дървовидна структура. След това обхватът се приема и одобрява.

Структура на работата по проекта (WBS)

За планиране на обхвата на проекта работата на проекта се разделя на сбор от постижими резултати, реализирани чрез задачи/дейности, и това разделяне се представя в графичен или табличен вид. Задачата е усилие или част от усилие, което е необходимо, за да се изпълни специфична работа, и което се извършва от определена единица в организацията на проекта. Структурата на работата по проекта (WBS) е графично средство, което организира работата по проекта в йерархична логическа структура. То може да изброява:

- Задачи
- Продукти
- Функционалност

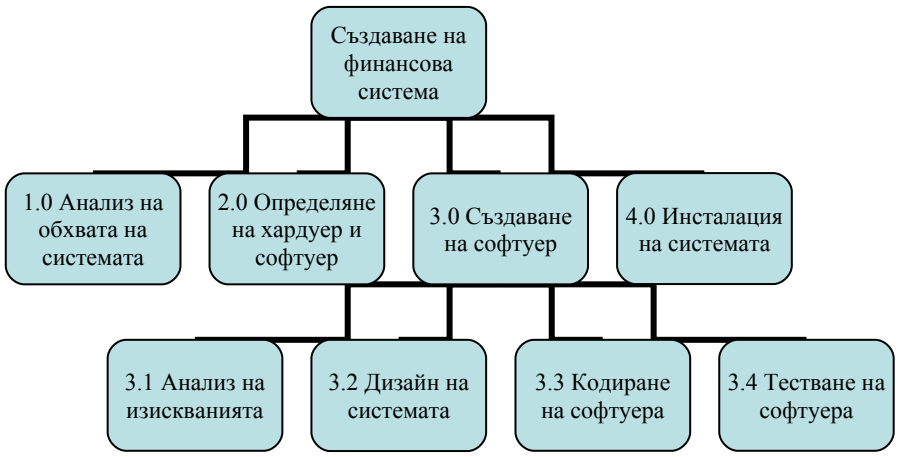
Целта на WBS е да идентифицира сравнително малки специфични части от работата на проекта, като стартира от основните задачи на първо ниво и постепенно ги детайлизира на следващите нива, обхващайки по този начин всички планирани задачи и резултати от проекта.

Пример на WBS

Ниво 1

Ниво 2

Ниво 3



Стъпки за реализация на WBS

- Идентификация на основните задачи и резултати от проекта, вкл. задачите, свързани с управлението на проекта
- Взимане на решение, дали съответните оценки за разходи и времетраене на задачите могат да се направят на това ниво на детайл или трябва да се продължи декомпозицията на работата
- Продължаване на декомпозицията – идентифицират се компоненти за всяка от определените на по-горното ниво задачи и резултати и отново се взима решение дали на това ниво те могат да се оценят и измерят

- Верификация на резултатите от декомпозицията – извършва се, за да се провери дали тя е пълна, т.е. дали всички планирани задачи са включени и дали е изцяло покрит обхватът на продукта или проекта.

Докъде може да стигне декомпозицията и кое е нейното дъно, е въпрос, с който се сблъскват много проекти. Отговорът се определя от логическата структура на WBS. Тя зависи от големината и сложността на проекта и може да се състои както от няколко хиляди задачи, така и само от няколко десетки задачи. Като референция за определяне на дъното може да се използват следните две правила:

- При по-малки или средни проекти (до 1 година)като дъно на декомпозицията може да се приемат задачи в размер от 8-10 часа до 40 часа. В много софтуерни проекти, чиято цел е реализацията на определени програмни средства или елементи от по-големи системи, задачите могат да бъдат дефинирани и в много по-малък размер, напр. в рамките на 1-2 часа, но това може да намали гъвкавостта на управлението и контрола на резултати за сметка на контрола на отделните технически действия, свързани с изпълнението на задачите, и може да доведе до микромениджмънт.
- При по-големи проекти (1 година или повече) размерът на задачите определящи дъното на декомпозицията, е пропорционален на големината на проекта, около 2-4% от времетраенето на проекта. Напр. при проект от 18 месеца –среден размер на задачите е около 1-2 седмици.

Редица задачи, напр. някои задачи по организацията, комуникацията и управлението на проекта (срещи, презентации, дискусии и др.), обикновено са с продължителност до 1-2 часа. Затова при определяне на големината на WBS е добре да се знаят положителните и отрицателните страни на по-голямата детайлизация и да се балансира в зависимост от големината, сложността и техническата оценка на конкретния проект:

Положителна страна: позволява по-голям и по-детайлен мониторинг, технически контрол и отчетност

Отрицателна страна: необходими са повече време, ресурси и финансови разходи за създаване на голяма структура на работата по проекта и за нейния последващ мониторинг, контрол и отчитане. Тези ограничителни елементи трябва предварително да са планирани и да се очаква, че колкото е по-голяма детайлизацията, толкова повече в процеса на изпълнение ще бъдат промените в плана и съответно по-труден техният контрол и управление.

Правила за създаване на WBS

- Всеки елемент на WBS трябва да има свой уникален идентификационен номер**.** Този номер се използва като референция при оценката на задачите по проекта, за контрол на разходите, за докладване и промяна на обхвата.
- При създаване на WBS не се определя последователността на изпълнение на задачите и не се включват ресурсите към задачите. Това се извършва в следващите стъпки – планиране на времето и планиране на ресурсите.
- За всяка задача се подготвя описание на работата (statement of work), което служи при дефиницията на обхвата на проекта. В много случаи самото име на задачата може да съдържа описание на работата по нея

20. Планиране на проект

Планиране на времето –мрежови диаграми, време за изпълнението на задачите, график на проекта

При планиране на времето се изпълняват следните дейности:

- Създаване на мрежова диаграма на задачите
- Определяне на времето за изпълнение на задачите
- Създаване на план-график на проекта

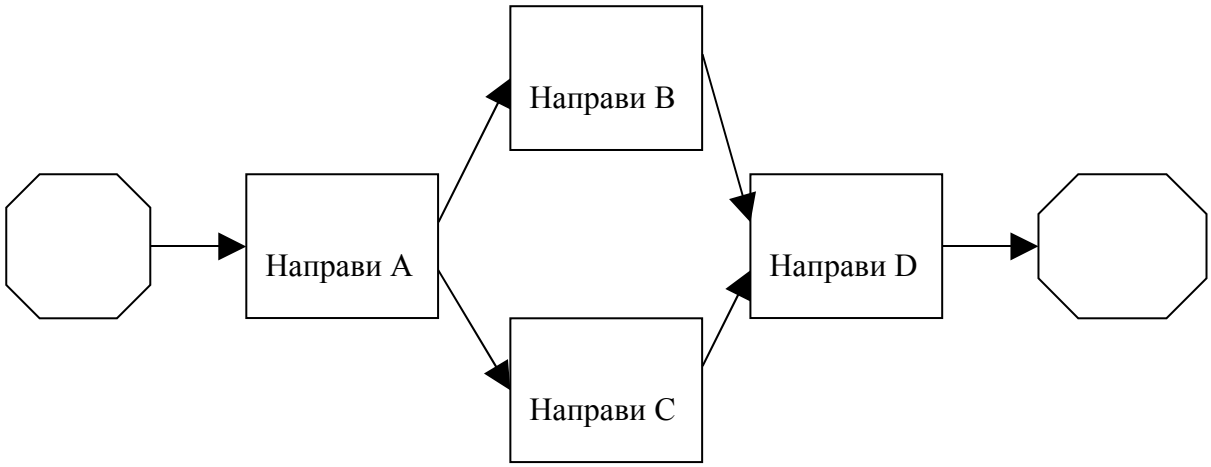
Създаване на мрежова диаграма

Мрежовата диаграма е логическа диаграма и включва задачите по проекта. Задачите са подредени според тяхното протичане: последователни или паралелни. Процесът на реализация на мрежова диаграма се състои от следните задачи:

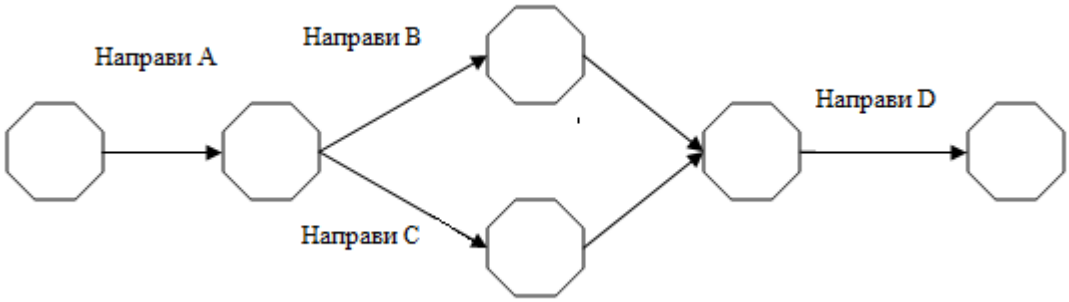
- Задачите се дефинират като списък от задачи от описанието на работата или се взимат от WBS
- Определя се дали задача е зависима от завършването на друга задача или може да се извърши паралелно с нея
- Добавят се зависимостите в списъка от задачи. Тези логически зависимости са:
    - Да завърши едната задача, за да започне зависимата от нея задача (finish to start – FS)
    - Да завърши една задача, за да завърши зависимата от нея задача (finish to finish – FF)
    - Да започне една задача, за да започне зависимата от нея (start to start – SS)
    - Да започне една задача, за да завърши зависимата от нея (start to finish – SF)

- Използват се различни методи за изобразяване на последователността на задачите:
    - Предшестващ метод (Precedence diagramming method – PDM). Използва правоъгълници за изобразяване на задачите и ги свързва със стрелки за определяне на зависимостите. Задачите са наредени в зависимост от предшестващите ги задачи и по този начин се показва редът, по който те ще се изпълняват. Нарича се още Activity-On-Node (AON) метод и се използва най-често при създаване на мрежови диаграми и план-график.
    - Метод чрез стрелки (Arrow diagramming method – ADM). Използва стрелки за изобразяване на задачите и ги свързва с възлите, за да покаже зависимостите. Възлите изобразяват събития (events), които стартират задачите. Нарича се още Activity-On-Arrow (AOA) метод и използва само FS зависимости.

  PDM



ADM



Определяне на времето за изпълнение на задачите

Определя се време за изпълнението на всяка задача. Характеристики на времето са работа и времетраене:

- Времетраене (duration) –времето, за което ще се изпълни задачата
- Работа за изпълнение на задачата (effort) – измерва се чрез човекочасове/дни/месеци и зависи както от нейното времетраене. Така и от това, колко хора работят по нея

3. Създаване на план-график

План-графикът включва пет основни стъпки, които се изпълняват в итерация до постигане на оптимален резултат:

- Определяне на предварително времетраене на задачите - при идеални условия с неограничени ресурси и пълно натоварване. Могат да се използват следните правила за определяне и оценка на времетраенето на задачата:
    - Използване на стандартна метрика или на собствена фирмена метрика, създадена от други проекти;
    - Използване на исторически данни – подобни задачи от други проекти;

    - Използване на експертна оценка от опитни колеги;
    - Използване на песимистични, оптимистични и най-вероятни оценки;
    - Проучване на опита в други фирми, реализиращи подобни проекти;
    - Обръщане на специално внимание и следене на задачите, разположени на критичния път;
- Определяне на предварителен график (идеален) според последователността на задачите в мрежова диаграма
- Идентификация на ресурсите и времето, през което те ще работят по задачите – определяне на работата по изпълнение на всяка задача
- Определяне на реалното време (duration) на задачите, което включва:
    - Календарното време за изпълнение на задачата, съобразено с почивни дни, празници, определено работно време
    - Количество ресурси за свършване на работата и тяхната ефективност
    - Колко време човешките ресурси ще работят по задачата –пълен ден, половин ден, само определени часове
- Създаване/промяна на основния план-график (control schedule)

Методи и средства за създаване на план график

- Метод на критичния път (Critical Path Method – CPM) – калкулира еднозначно следните дати за всяка задача:
    - Ранен старт (Early start – ES)
    - Късен старт (Late start – LS)
    - Ранен край (Early finish – EF)
    - Късен край (Late finish – LF)

на базата на последователната мрежова логика и определеното време (duration). CPM се фокусира върху калкулацията на плаващото време (float), за да определи кои задачи имат най-малка гъвкавост по отношение на графика.

- Критичен път (Critical Path). Последователността от задачи, които цялостно определят цялостното времетраене на проекта (project duration). Това са обикновено задачите с минимално плаващо време, което най-често е равно на 0. В мрежовата диаграма това е най-дългият път (последователност) от задачи в проекта.
- Program Evaluation and Review Technique (PERT) – метод за оценка, който използва средни претеглени стойности за времетраенето на задачите, за да изчисли техните времена.
  Формула за изчисление:
  (Оптимистично време + 4 * Най – вероятно време + Песимистично време) / 6
  Различава се от CPM по това че, CPM основно използва най-вероятното време
- Диаграми на Гант (Gantt Charts). Времетраенето на задачите се изобразява с правоъгълници във времето за изпълнение на проекта. Това е най-използваният и лесно разбираем метод, особено за определяне статуса на проекта и неговия прогрес. Лесно представя план-графика и зависимостите между задачите.

Определяне на датите по метода на критичния път

- Прав пас (forward pass). Правият пас (пас напред) представлява сумиране на всички времена на задачите, като се започва от първата задача на мрежовата диаграма (старт на проекта) и се завършва с последната задача (край на проекта). Целта му е да се намери възможно най-краткото време за изпълнение на проекта. Тази процедура изчислява колко е времето за изпълнение, ако всяка задача започва колкото е възможно най-рано, т.е. без закъснение на предишната задача. С други думи, правият пас показва най-ранната дата, когато една задача може да започне (ES), и най-ранната дата, когато една задача може да завърши (EF).
- Обратен пас (backward pass). Обратният пас (пас назад) изработва най-късното време, когато една задача може да започне, без да причини закъснение на целия проект. Изчисляването започва от края на проекта чрез изваждането на времето за изпълнение на всяка задача, докато се стигне до началото на проекта. Чрез обратния пас се определят най-късната дата за завършване на една задача (LS). В практиката обратният пас се използва, за да се определи колко време една задача може да закъснее, без да закъснее целия проект. Това се нарича плаващо време
- Плаващо време (float). Плаващото време е разликата между реалното време за изпълнение на една задача и планираното време за нейното изпълнение. Задачите на критичния път обикновено имат минимално или въобще нямат плаващо време. Изчислява се за всяка задача по следния начин:

  Float = LF – EF или Float = LS – ES

- Общо плаващо време (Total float). Общото време, за което една планирана задача може да закъснее спрямо най-ранната дата на нейното започване (ES), без да причини закъснение на целия проект. Изчислява се със следната формула:

  Общо плаващо време = късен старт (LS) – ранен старт (ES) – времетраене на задачата (duration) + 1

  Прибавя се единица във формулата, защото включва и самите дати на LF и ES.

- Свободно плаващо време (Free float). Периодът от време, когато една задача може да закъснее, без да причини закъснение на следващите задачи. Това време определя доколко времето за изпълнение на една задача е гъвкаво и позволява определено закъснение
- Определяне не ES, EF, LF и LS:
  - ES за текущата задача = EF на предишната. При повече от една предишни задачи се взима най-късната EF.
  - EF = ES – времетраене на задача (duration).
  - LF е най-късната дата, когато дадена задача може да завърши без да причини закъснение на даден резултат (milestone or deliverable), или на крайната дата за завършване на проекта
  - LS = LF – времетраене на задачата (duration)

4. Планиране на ресурси, разходи и бюджет на проекта

20. Планиране на проект

За да се планират разходите и се направи бюджетът на един проект, е необходимо да се определят какви ресурси (човешки и материални) са на разположение за работа по неговите задачи, разходите на всяка задача и общите разходи на целия проект

Определяне на ресурсите, необходими за изпълнението на всяка задача

- за всяка задача се определят необходимите ресурси, включват:
  - човешки ресурси – хората от екипа на проекта с техните специализирани умения
  - материални ресурси – софтуер и хардуер за изграждането на технологична среда, комуникационни средства, материали, помещения и др.
- Определя се количеството на ресурсите, необходими за изпълнение на задачата в определеното време
- Определя се кога и колко време ресурсите са на разположение на проекта

Дефиниция и оценка на разходите

Разходите се състоят от:

- Разходи за ресурсите: цената на човешките ресурси, цената на машините и материалите, софтуера, лицензи, ноу-хау, консултантските услуги и др.
- Разходите за извършване на дадена задача в определеното време и с определените ресурси
- Разходите за изпълнение на целия проект

Планиране на разходите

Видовете разходи, които се планират, са:

- Труд (човекодни)
- Материали
- Компютри и оборудване, машини
- Софтуер
- Консумативи, комуникации, ток, отопление, поддръжка
- Сгради и помещения
- Обучение
- Пътуване и командировки
- Управленски и административни разходи
- Други непредвидени разходи
- Резерв

Мерни единици за оценка на разходите

За постигане на оптималната оценка на разходите и бюджета на проекта се прилагат различни метрики в зависимост от неговата големина, сложност и съответните функционални изисквания към продукта:

- Ориентирана към оценка на разходите за труд:
  - Разход на регламентиран труд, измерва се в лв./час;
  - Извънреден труд, измерва се в лв./час;
  - Премии, измерва се в лв./час;
- Ориентирана към размера на продукта:
  - Линии, код, измерва се в брой линии

  - Време за реализация – в часове, дни, седмици или месеци
  - Човешки ресурси – в брой на хората, участващи в екипа на проекта
  - Труд – в човекочасове/дни/седмици/месеци
  - Документация, в брой страници
- Функционално ориентирана:
  - Количество на задачите, чрез използване на коефициент или мащаб на сложността
  - Количество интерфейси – в необходимия брой интерфейси на продукта
- Ориентирана към производителността:
  - При софтуерни приложения - чрез линии код/човек/ден
  - Документация – чрез страници/човек/ден

Видове подходи и методи за оценка на разходите

- Преувеличена оценка (overestimate). Тази оценка се базира на закона на Паркинсон: „Работата по проекта се самоувеличава, за да запълни определеното време" (Work expands to fill the time available). Преувеличената оценка най-вероятно ще доведе до удължаване на времето за изпълнение на проекта (в сравнение с реалното време).
- Подход за оценка „от горе на долу" (top down). Оценките са базирани на необходимата работа на базови модули, резултати, продукти. Разпределят се пропорционално на работата по съответните компоненти, като се стартира от най високото ниво WBS
- Подход за оценка „от долу на горе" (bottom up). Изпълняват се следните действия:
  - Разделя се проектът на по-малки и по-малки компоненти;
  - Спира се до задача/резултат, който може да се постигне от 1-2 души за 1-2 седмици (3-4 седмици при голям проект)
  - Оценяват се разходите за най-малките задачи
  - Сумират се оценките на по-горните нива
  - Сумира се цялата оценка на разходите по проекта
- Оценка на базата на експертно мнение. Използва се при приблизително 26% от оценките на разходите
- Оценка по аналогия (прилага подхода top down), използва се при приблизително 60% от оценките на разходите. Това е форма на експертна оценка, която реализира следните стъпки:
  - Идентификация на основните и важни за разходите компоненти (функции, процеси, софтуерни програми, резултати, документи и др.)
  - Избор на предишни аналогични проекти
  - Избор на най-близките по параметри компоненти от тези проекти
  - Сравнение с бъдещия проект
  - Оценка на разликите между различните проекти и бъдещия проект посредством анализ на различията (gap analysis)
  - Прибавяне на установените разлики към оценката
- Оценка на базата на „цена на проекта, за да се спечели" (price to win)
- Параметрични модели, използва се при приблизително 14% от оценките на разходите. Тези модели използват различни характеристики и функционални елементи на проекта, които се включват в математически модели за оценка на разходите по проекта.

- Анализ на функционалните елементи (Function Point Analysis):

Използват се функционални елементи и характеристики:

- Процеси, Use Cases;
- Брой транзакции в процес или Use Case;
- Работа за извършване на транзакциите/процеса;
- Компоненти (обекти) (брой и сложност):
  - Брой екрани * човекодни
  - Брой справки * човекодни
  - Брой компоненти * човекодни
  - Брой класове * човекодни
- Функционални елементи (файлове)
  - Вътрешни файлове
  - Изходни файлове
  - Файлове с данни
- Интерфейси
- Външни заявки към системата

- COCOMO (Constructive Cost Model). Предложен е от B.Boehm и като вход на модела е големината на системата, измерена като: линии код (Lines of code – LOC). Производителността се измерва на база линии код/човек/ден. Реализира се чрез три модела:
  - Основен модел. Оценката при него изцяло се базира на големината на системата в линии код.
  - Междинен модел. Оценката се базира на големината на системата в линии код, като се взима предвид опитът и производителността на човешките ресурси, ограниченията на софтуера и хардуера на проекта и използването на съвременни средства и техники за програмиране
  - Детайлен модел. Системата се разделя на нива и се оценява всяко ниво за всяка фаза на жизнения цикъл на системата по модули, подсистеми и система. Практически дефинира различни оценки за всяко ниво и фаза на жизнения цикъл на системата.

Най-често срещаният процес на оценка е: Аналогия – Експертно мнение – Параметрични модели

Ревизия на оценките

След първоначално оценяване се извършва нова итерация на процеса с цел ревизия на оценките, за да се стигне до по-реална крайна оценка.

Ревизията се извършва на базата на:

- Размера на задачите
- Оценка на производителността, постигната в предишни проекти
- Аналогия с предишни проекти със същата големина и функционалност
- Установена метрика за оценка на функционалността и големината на проекта

Бюджет на проекта

Така дефинирани, всички разходи по проекта се сумират и определят бюджет на проекта. Той е разпределен по различни задачи и продукти/резултати, за да се създаде основа за измерване и оценка на проекта според неговите разходи. Бюджетът на проекта (разходите) се оценява на базата на определен период (например фаза от

проекта) и се контролира дали има преразход по проекта или спестени пари. След неговото съставяне е необходимо одобрение на бюджета на проекта.

Цена на проекта

Цената на проекта в повечето случаи е различна от неговия бюджет и обикновено би трябвало да е по-висока от разходите (бюджета на проекта). Но има случаи, особено при участието на търгове, когато се оферират дъмпингови цени на базата на „Цена на проекта, за да спечели търгът" и по-късно тези цени се компенсират с влошено качество, нереализирана функционалност или анекси към вече сключени договори. Цената на проекта освен разходи за извършването на работата по проекта може да съдържа и:

- Процент от административните разходи на организацията-изпълнител
- Процент от разходите за обучение и поддържане на знанията на персонала на организацията, вкл. и на членовете на екипа на проекта
- Извънредни разходи (overheads)
- Предвидени разходи за покриване на фирмен рис
- Процент печалба

Въпрос номер 21 от конспекта за държавен изпит на специалност «Софтуерно инженерство»

Използване на XML за структуриране, валидация, обработка и представяне на документно съдържание.

Изготвил: Станислава Бобчева, спец. Софтуерно инженерство, 4-ти курс

Съдържание

Добре структуриран XML

Произход на маркъп езиците и основни концепции в XML

Маркъп език (според Уикипедия) е модерна система за анотиране на текст по такъв начин, че маркирането синтактично се различава от самия текст. Терминът произлиза от редактирането на ръкописи, където с друг цвят например се пишат забележките от редакцията. Основните характеристики на маркъп езиците са:

- ✓ За стилистично оформяне – например от HTML <I>, <B>, <U>
- ✓ За структуриране – например от HTML <P>,
, <H2>
- ✓ За добавяне на семантично значение (мета данни) – например от HTML <TITLE><META NAME=keywords CONTENT="….">
- ✓ За добавяне на функционалност – например от HTML <BLINK><A HREF="[link]">Click here</A>

Маркъп съдържание има например във всеки doc файл, но то може да бъде обработено само от едно приложение (proprietary). Въпреки че такива файлове съдържат текст, за правилното визуализиране и работа с тях ни трябват точно определени програми. Това са binary файлове. Нуждата обаче от преносимост между различни платформи и приложения води до разработването на отворени маркъп езици.

През 1969 година е разработен един от първите маркъп езици – **Standard Generalize Markup Language (SGML)**, от който водят началото си най-разпространените маркъп езици днес – XML и HTML. Въпреки че от гледна точка на изчерпателност този език е много добър – с него може да се публикува почти всичко (математика, химия, UNIX документация, речници), той се оказва твърде сложен – средствата за обработка са много скъпи и сложни. Друг негов недостатък е, че не е разработен директно за WWW.

HTML (Hypertext Markup Language) се появява през 1989 и негов създател е Тим Бърнърс Ли. Неговият документен тип е описан чрез SGML. Използва се предефинирано множество от тагове (около 60). Има много частни разширения, което често води до несъвместимост на браузърите. Тъй като обаче таговете са предефинирани това води до ограничени възможности в експресивността на езика – повечето тагове са свързани със структурната организация на документа и неговото представяне, труден е за „разбиране" от компютри.

XML (eXtensible Markup Language) – позволява потребителят да си дефинира собствени тагове, затова е разширяем. Всеки документ съдържа четири типа информация:

- ✓ Съдържание – текст, графични елементи, звук
- ✓ Структура – на какви части се разделя съдържанието и как се подреждат те
- ✓ Външен вид – как се визуализират съдържанието и структурата на документа
- ✓ Меда данни – допълнителна информация за документа

XML йерархии, синтактични правила

Таг – това е маркъп, който се разделя от останалото съдържание с '<' и '>'. Съдържа името на елемент. Отварящ таг: <elementname>, затварящ таг </elementname>.

Елемент – означава се с отварящ и затварящ като между тях е съдържанието

- ✓ Именуване – винаги трябва да почва с буква или '_', след това може да има цифра, '-' или '.'. С ':' се разграничават името на пространството от имена и името на елемента. Имената не могат да започват с XML, XmL и т.н. Различават се малки и големи букви.

Атрибут – предоставят допълнителна информация за елемент

- ✓ Като им се задават стойност се огражда се с „” или ‘’
- ✓ Различават се главни и малки букви (case-sensitive)
- ✓ Character data or tokenized (value=”Blue Peter” character data, value=”blue” single token, value=”red green blue” tokens)

Елемент или атрибут

- ✓ Зависи от предназначението на конкретния документ.
- ✓ Трябва да се различи съдържанието от мета данните – съдържанието е това, което ще бъде видимо (представяно), останалата мета информация може да се зададе в атрибути.
- ✓ Правило: Като се махне всичкият маркъп, документът трябва да може да се чете и разбира.
- ✓ Използват се атрибути, когато се задава някакво свойство на даден елемент.
- ✓ Използват се атрибути за проста валидация на типа данни.
- ✓ Използват се елементи за сложна структурна валидация.

Структурата на XML документите е йерархична – всички елементи са вложени, т.е. не може елемент да има отварящ таг в рамките на един елемент и затварящ в рамките на друг. На всеки добре-структуриран XML документ може да се съпостави дърво. Има един елемент, който е корен на документа и всички останали са вътре в него. За всеки отварящ таг трябва задължително да има затварящ такъв.

План за създаване на XML документа:

- ✓ Въведение (prolog) - всеки XML документ започва с XML декларация – <?xml version="1.0" encoding="UTF-8" standalone=“yes"?>. XML декларацията има задължителен атрибут version за използваната версия на XML спецификацията и също така незадължителните encoding и standalone. Ако standalone е “yes”, то документът не зависи от други файлове. Кодирането на символите по подразбиране е UTF-16 или UTF-8.

Може да има и зададено стилово множество – <?xml:stylesheet type="text/css" href="s.css" **?>.** Също така може да се зададе и документният тип - <!DOCTYPE test SYSTEM "test.dtd">.

- ✓ Инстанция – това е йерархията от елементи
- ✓ Допълнения:
  - коментари – <!—comment -->
  - инструкции към процесора (<?PITarget Status="draft" ?>) – това е информация, необходима от външно приложение. XML декларацията също е инструкция към процесора.

Текстовото съдържание на елементите може да бъде два типа **PCDATA** (parsed character data) и **CDATA** (character data). В първия случай (PCDATA) това не се обозначава по никакъв начин и маркъп символи в съдържанието се разпознават от парсера. Ако в съдържанието на някой елемент искаме да включим знаците, използвани за маркъп, то трябва да ползваме специални последователности от символи:

- ✓ & - &
- ✓ < - <
- ✓ > - >
- ✓ ‘ - '

- ✓ “ - "

Ако искаме да избегнем това обаче, може да ползваме CDATA – започва с <![CDATA[и завършва с]]>. Съдържанието в такава секция не интерпретира по никакъв начин и се подава на приложението както е.

Пространства от имена

Поради използването на **едни и същи имена**, придавайки им **различно значение** (имат различна семантика), се налага употребата на пространства от имена за елементите и атрибутите. Пространство от имена е абстрактна нотация за група от имена. В рамките на един XML документ дадено пространство от имена може да се означава с префикс (ако не е това по подразбиране, когато префикс няма).

Qualified names – префикс:име

Уникална идентификация на пространството от имена – **URL**. Обосновка: може за различни пространства от имена в различни документи да се ползва един и същи префикс.

Задават се като **атрибут:** **xmlns**. Може да се зададе на всеки елемент. В един документ може да има информация, дефинирана в няколко пространства от имена.

XML валидация чрез Document Type Definitions (DTD)

Цели на валидирането

Когато се създават XML документи, те могат да се категоризират в групи от сходен тип в зависимост от елементите и атрибутите, които съдържат. Елементите и атрибутите, които се срещат в един документ, определят неговия речник. Използвайки DTD, лесно могат да се валидират XML документи спрямо дефиниран речник от елементи и атрибути. За да се провери дали съдържанието на даден XML документ е валидно според декларациите в съответния DTD документ, може да се ползва XML парсер.

DTD структура

DTD е файл, който съдържа формална дефиниция на конкретен тип документ.

Може да се намира вътре в XML документа, в отделен файл или и двете – при наличие и на двата вътрешният доминира.

Вътре в документа (само едно вътрешно DTD):

```
<!DOCTYPE MyDoc [
        <!-- DTD appears here -->
        <! … >
        <! … >
]>
<!-- Rest of XML file -->
<MyDoc> ……
</MyDoc>
```

Референция към **външен DTD файл** (за преизползване; декларациите, специфични за XML документа се оставят вътре в него):

```
<!DOCTYPE MyDoc
        SYSTEM "./MyDoc.dtd" [
        <!-- Extra declarations -->
```

```
        <! … >
        <! … >
    ]>
    <!-- Rest of XML file -->
```

SYSTEM – локални файлове и URL

PUBLIC – за публични каталози

DTD декларациите са инструкции към XML процесора. Означават се <! .. > или <! … [<! … >]>.

Документен тип – <!DOCTYPE My_XML_Doc […]>

- ✓ посочва името на елемента корен на XML документа
- ✓ след това в квадратните скоби се декларират останалите елементи

Character data – <![CDATA[…]]>

Семантични единици – <!ENTITY … >

- ✓ Семантична единица – един XML документ може да бъде разделен в няколко отделни файла, като всяка единица информация се нарича семантична единица
- ✓ всяка семантична единица има име, с което се идентифицира
- ✓ може да се реферира с entity reference
- ✓ видове семантични единици
    - Built-in – не е необходимо да се декларират (това са & < и т.н.)
    - Character – те също не се декларират; обикновено се ползват за символи, които трудно се изписват; реферират се или © , или с кода от Unicode представянето (&#x00A9)
    - General – декларират се в DTD; най-често се ползват за деклариране текст, който може лесно да бъде сменян; реферират се с &име;
    - Parameter – те са за употреба само в DTD; тук текстът може да съдържа части от DTD декларации за разлика от другите entities; след ключовата дума ENTITY се поставя % следван от името на entity и после е съдържанието; реферират се с %име.
- ✓ Външни entities – текстови: могат да се посочват с SYSTEM или PUBLIC идентификатор, binary: същото като текстовите + може с ключовата дума NDATA да се задава видът на файла

Нотация – <!NOTATION … >

- ✓ Задава описание на външна не-XML семантична единица
- ✓ Използва се в комбинация с NDATA

Елемент – <!ELEMENT … >

- ✓ използва се за декларация на нови елементи и модела на тяхното съдържание
- ✓ състои се от три части – декларацията ELEMENT, името на елемента и модела на съдържанието
- ✓ съдържанието може да бъде
    - елементно – (name, e-mail*), като всеки реферирян елемент също трябва да има дефиниция в DTD
    - смесено – (#PCDATA|a|b)*
    - празно – EMPTY
    - всякакво – ANY
- ✓ Индикатори за брой срещания (cardinality, quantity indicators):

- + - [1..*]
- * - [0..*]
- ? - [0..1]

✓ Последователност (sequence) – задава се ‘,’, и избор (choice) – задава се с ‘|’

Атрибути – <!ATTLIST … >

✓ Атрибутите могат да бъдат прикрепени към декларацията на елементите или да бъдат в отделна декларация.

✓ Декларацията се състои от три части:

- Ключовата дума ATTLIST
- Името на елемента, за който са тези атрибути
- Списък с декларациите на атрибутите – за всеки атрибут се задават име, тип и декларация на стойността.

  Пример: source CDATA #IMPLIED

  Декларацията на стойността може да бъде - стойност по подразбиране, фиксирана стойност, изисква ли се (Is required), is implied (or is optional).

  Заб: Декларациите на пространства от имена, които ще се ползват в XML документа, също трябва да се дефинират като атрибут, въпреки че според препоръката за пространствата от имена на W3C те са декларации, а не атрибути. Това се получава, тъй като тази препоръка излиза след завършването на DTD синтаксиса.

✓ Възможни типове на атрибутите:

- **CDATA –** показва, че атрибутът има стойност от символи (character data), парсерът не интерпретира маркъп символите
- **ID –** показва, че стойността на атрибута уникално идентифицира елемента
- **IDREF –** стойността на атрибута е референция по ID към уникално идентифициран елемент
- **IDREFS –** стойността на атрибута е списък от разделени с празно място IDREF стойности
- **ENTITY –** стойността на атрибута е референция към външна (unparsed) семантична единица. Това може да бъде картинка, MP3 или друго binary.
- **ENTITIES –** списък от ENTITY стойности разделени с празно място
- **NMTOKEN –** стойността е name token – това е стринг от група стандартни стрингови имена.
- **NMTOKENS –** списък от NMTOKEN стойности разделени с празно място
- **Enumerated List** – може да се декларира енумерация от възможни стойности. *Пример*: (alpha|num).

<![INCLUDE[…]]> и <![IGNORE[…]]>

✓ Могат да се ползват като switch включване и изключване на декларации

Недостатъци на валидацията с DTD

✓ Синтаксисът не е XML

✓ XML пространства от имена – всеки път, когато елемент или атрибут се добавят в DTD трябва да се включи и пространството от имена към името. Освен това в DTD пространствата от имена се третират като атрибути, въпреки че според препоръката на

W3C те са декларации. Това може да доведе до някои объркванията (например ако се съединят два XML документа с едно и също пространство от имена, което обаче има различен префикс в различните документи).

- ✓ Ограничени са типовете данни – единствените типове в DTD се използват при декларацията на атрибути и те са крайно недостатъчни. Няма и начин да се ограничи типът на текстов елемент – например не може да се зададе даден елемент да бъде число.
- ✓ Ограничения в модела на съдържанието – не могат да се задават връзки от тип наследяване; не могат да се задават ограничения в броя срещания на даден елемент (например от 1 до 10).

XML валидация чрез XML Schema

Схема е всеки документ, който се използва за моделирането на структурата на нещо. Днес терминът XML Schema се използва по-конкретно за W3C XML Schema технологията. XML Schema, подобно на DTD, позволява да се опише структурата на един XML документ. Самата XML Schema се валидира от DTD.

Спецификации

Спецификацията на XML Schema е разделена на три части.

- ✓ Част 0: Въведение – включва въведение в основните концепции, не е със задължителен характер.
- ✓ Част 1: Структури – дефинира структурите, които се ползват в XML Schema
- ✓ Част 2: Типове данни – специфицира какви типове могат да се ползват за елементи и атрибути

Структури

- ✓ Дефиниции на типове - <simpleType>, <complexType>
- ✓ Декларации на атрибути - <attribute> (декларациите на атрибути винаги са последни, след декларациите на елементи)
- ✓ Декларации на елементи - <element>
- ✓ Дефиниции на групи от атрибути - <attributeGroup>
- ✓ Дефиниции на групи - <group>
- ✓ Декларации на нотации - <notation>
- ✓ Анотации - <annotation>

Заб: Декларациите се правят на нещата, които ще използват в инстанция на XML документа, а дефинициите на нещата, които се ползват в самата схема (например дефинира се нов тип).

Глобални и локални декларации – глобалните са тези, които са директни деца на <schema> елемента, а локалните нямат като родител този елемент. Глобално декларирани елементи могат да се ползват като се зададе стойност на ref атрибута на декларацията element (ако targetNamespace не е namespace по подразбиране, то елементът трябва да се реферира с qualified името си). Въобще всички глобални декларации отиват в target пространството от имена и когато се ползват трябва да се дава неговият префикс (ако то не е пространството от имена по подразбиране).

<schema>

- ✓ Коренът на схема документа

Атрибути:

- ✓ targetNamespace – това е пространството от имена на елементите, които ще бъдат в инстанциите

<element>

Атрибути:

- ✓ minOccurs, maxOccurs – брой срещания (не са позволени в глобално декларирани елементи, могат да се ползват при реферирането на глобално дефиниран елемент)
- ✓ default – позволява задаването на стойност по подразбиране за елемента; ако елементът няма стойност, то той ще се третира от валидатора със стойността по подразбиране
- ✓ fixed **–** за задаване на фиксирана стойност

<any> - позволява даден XML документът да включва елементи, които не са зададени в схемата му; използва се само в модела на съдържанието, не може да е глобална декларация

За задаването на модела на съдържанието на елемент може да се ползват

<complexType> - с него се задава моделът на съдържанието на елемент; първо се задават винаги декларациите на елементи, а след това на атрибути

Атрибути:

- ✓ mixed – true или false; дали може да има само текст или текст и елементи
- ✓ name – *заб:* локалните декларации не могат да имат име (такива сложни типове се наричат анонимни); глобалните декларации за разлика, винаги трябва да имат име

<group> - позволява да се групират елементи и така получената група да се преизползва на различни места; използва се само за елементи, не са позволени декларации на атрибути; когато трябва да се ползва дадена глобално дефинирана група, се използва референция към нея; редът на елементите в групата се задава с помощта на декларациите <sequence>, <choice>, <all>

<sequence> - съдържа декларации на елементи, които трябва да присъстват в XML документа в посочения ред

<choice> - съдържа декларации на елементи, от които точно един може да присъства в XML документа

<all> - съдържа декларации на елементи, които трябва да присъстват в XML документа независимо в какъв ред

Горните четири декларации могат да се ползват по различни начини, за **да се зададе модела на съдържанието** – той може да бъде зададен вътре в декларацията на самия елемент (inner content models), element declarations, element wildcards.

В сравнение с DTD схемата дава много повече възможности за **задаване на уникалност**:

- ✓ може да се зададе съдържанието на даден елемент да е уникално
- ✓ може да се зададе атрибути, които не са ID, да са уникални
- ✓ може да се зададе комбинация от елементно съдържание и атрибути да бъде уникална
- ✓ прави се разлика между уникален и ключ
- ✓ може да се декларира област от документа, в която нещо е уникално

<key> - заедно с <keyref> позволява да се дефинира връзка между два елемента; вътре се дефинират <selector> - посочва за кой елемент е ключът и <field> - посочва кой връх е ключът

<unique> - отново се задават <selector> и <field>; стойността на нещо, което е unique обаче може да е и null

<attribute> - използва се за декларация на атрибути; обикновено се среща или в <attributeGroup>, или в <complexType>; може също да се ползва и в <extension> или <restriction> при дефиниране на производен тип.
<anyAttribute> - позволява XML документът да се разширява с атрибути, които не са специфицирани в схемата му
<attributeGroup> - позволява декларацията на група от атрибути; ако е глобална трябва да има име
Има две основни декларации за работа с повече от един XML Schema документи:
<import> - позволява да се внесат глобални декларации от други XML Schema документи като се зададат атрибутите namespace – target namespace на внасяната схема, и schemaLocation – местоположението на внасяната схема; винаги е глобална декларация – директно дете на <shcema>
<include> - много е подобна на <import> декларацията, но разликата е, че внесените глобални декларации от новата схема директно се причисляват към target namespace-а; затова има само атрибут schemaLocation и с него се посочва къде е схемата, от която да се внесат глобалните декларации

Типове данни

Вградени (built-in) – те включват 2 групи:

- ✓ примитивни (основни) – не са дефинирани, използвайки други типове; служат като основа за дефинирането на всички останали типове
  - string
  - boolean
  - float
  - double
  - decimal
  - timeDuration
  - recurringDuration
  - binary
  - uriReference
  - ID
  - IDREF
  - ENTITY
  - NOTATION
  - QName
- ✓ вградени производни (built-in derived) – използвани са други примитивни и производни типове за да бъдат дефинирани (но са вградени в схемата) /например positiveInteger, recurringDay и т.н./

Дефинирани от потребителя (user-defined)
<simpleType> - използва се за дефиниране на потребителски тип данни, като новият тип винаги се базира на вече съществуващ тип данни – или от вградените типове, или вече дефиниран от потребител друг тип. За целта се използват: <restriction>, <list>, <union>
<restriction> - типът, който се получава, е подмножество на основния тип (<restriction base=”name of the simpleType you are deriving from”>); използва се с фасетите за ограничения

<list> - <list itemType="name of simpleType used for validating items in the list">; може и вътре да се дефинира simpleType (типът на нещата от списъка)
<union> - позволява стойностите в документа инстанция да се валидират спрямо няколко типа едновременно (<union memberTypes="whitespace separated list of types">)

Фасети

Фасет е свойство или характеристика на <simpleType>. (A *facet* is a single property or trait of a <simpleType>). Фасетите на даден тип могат да се ползват, за да се ограничи неговото множество от стойности. С помощта на фасетите потребителят може да си дефинира нови типове от вградените такива.
Фундаментални – не могат да бъдат променяни; дефинират типа (няма ги в Beginning XML 4th)

- ✓ Equal
- ✓ Order
- ✓ Bounds
- ✓ Cardinality
- ✓ Numeric

Фасети за ограничения – позволяват да се ограничи множеството от стойности; ползват се в <restriction> (в лекциите са различни от тези в Beginning XML 4th, тук съм сложила от книгата, защото мисля, че са по-актуални)

- ✓ minExclusive – Allows you to specify the minimum value for your type that excludes the value you specify
- ✓ minInclusive – Allows you to specify the minimum value for your type that includes the value you specify
- ✓ maxExclusive – Allows you to specify the maximum value for your type that excludes the value you specify
- ✓ maxInclusive – Allows you to specify the maximum value for your type that includes the value you specify
- ✓ totalDigits – Allows you to specify the total number of digits in a numeric type
- ✓ fractionDigits – Allows you to specify the number of fractional digits in a numeric type (e.g., the number of digits to the right of the decimal point)
- ✓ length – Allows you to specify the number of items in a list type or the number of characters in a string type
- ✓ minLength – Allows you to specify the minimum number of items in a list type or the minimum number of characters in a string type
- ✓ maxLength – Allows you to specify the maximum number of items in a list type or the maximum number of characters in a string type
- ✓ enumeration – Allows you to specify an allowable value in an enumerated list
- ✓ whitespace – Allows you to specify how whitespace should be treated within the type
- ✓ pattern – Allows you to restrict string types using regular expressions

Сравнение с DTD

- ✓ Ползва се основният XML синтаксис, докато DTD дефинира свой отделен
- ✓ Поддържа изцяло препоръката за пространствата от имена
- ✓ По-експресивен синтаксис от DTD
- ✓ Повече поддържащи средства (DOM, браузъри)

- ✓ DTD поддържа само 10 типа данни, докато схемата 37 вградени и дава възможност също за дефиниране на нови
- ✓ По-добър модел на съдържанието – например може да се задава по-точно ограничението за брой срещания на даден елемент; могат много неща да се преизползват с помощта на дефинирането на глобални типове, групи от елементи, групи от атрибути, елементи
- ✓ Не само глобални, но и локални имена
- ✓ Неограничена възможност за разширяване
- ✓ Повече речници (базирани на използването XML пространства от имена)
- ✓ В XML Schema не може да се дефинира ENTITY
- ✓ DTD може да се вгради директно в документа, който се валидира, докато XML Schema е винаги в отделен файл

Използване на XSLT (eXtensible StyleSheet Language Transformations) и XPath за алокиране, манипулиране и представяне на XML съдържание.

XML документите обикновено са предназначени предимно за компютърна обработка, а не да бъдат четени от хора. Често се налага обаче XML съдържанието да се представи в различен вид в зависимост от това за кого или какво е предназначено. Затова W3C разработват спецификацията XSL – Extensible Stylesheet Language. По време на разработката се оказва, че се опитват да покрият две различни функции в една спецификация – да дефинират елементи за представяне на съдържание и да дефинират синтаксис за трансформиране на елементите и документната структура. Затова спецификацията се разделя на две – XSL Formatting objects (или само XSL) и XSL Transformations. Първото е език за форматиране на XML данни за представянето им на екран, хартия или друг медиен носител. Второто е език за трансформиране на XML документи в други документи.

XSLT

XSL процесорът може да **трансформира** един входен XML документ в XML, HTML или текстов изходен документ, като използва правилата, зададени в друг XML документ (XSLT файл). За разлика от JavaScript XSL е декларативен, а не императивен език. На входния документ се съпоставя входно дърво, а на изходния – изходно. Може да се добавя текст преди и след дадено съдържание, да се махат, създават и пренареждат елементи.

В сравнение с DOM възможностите са по-ограничени. Друга основна разлика е, че XSL е скриптов език, а DOM програмен интерфейс.

Client-side XSLT – ако браузърът поддържа XSLT, то документът може да се превърне в XHTML директно през него. Също така може да се ползва JavaScript за извикване на трансформацията – позволява да се прави проверка за браузъра и да се ползват различни стилови множества спрямо различните браузъри или изискванията на потребителя

Server-side XSLT - Ако обработката на входния XML файл е тежка или ако браузърът не поддържа XSLT, тя може да се извършва на сървърната страна.

Има дефинирани **34 елемента**, но дори само с три от тях могат да се правят полезни стилови множества (style sheets) – stylesheet, template, apply-templates

<xsl:stylesheet> - коренът на документа

<xsl:template> – специфицира правило за трансформация; match атрибутът приема XPath израз; атрибут mode. Темплейтът ще бъде приложен, ако някой връх във входното дърво отговаря на match атрибута.
<xsl:import> - използва се, за да се включат няколко XSLT файла
<xsl:value-of> – използва се, за да се преобразува обектът, посочен в select атрибута, в string (не е рекурсивна функция); ако е посочен елемент, то се взима неговото съдържание и всички елементи, които съдържа
<xsl:apply-templates> – използва се в даден темплейт за извикване на други темплейти; предизвиква рекурсивна обработка на всички потомци на елемент, който е бил match-нат
<xsl:output> - дава контрол върху изходния документ; слага се директно след <xsl:stylesheet>
<xsl:sort>
<xsl:number> - за автоматично номериране
Атрибути:

- ✓ level – ‘single’, ‘any’, ‘multiple’
- ✓ count – “list-1|list-2”
- ✓ format – “1.A”
- ✓ from – “3”
- ✓ grouping-separator – “,”
- ✓ grouping-size – “3”
- ✓ value – “position()”

<xsl:variable> - могат да се декларират променливи; за да се достъпи стойност на променлива се ползва ‘$’ (<xsl:value-of select=”$colour”/>); също така стойността на променливата може да се ползва и в елементи от резултатното дърво (<ajr:glyph colour=”{$colour}”/>). По този начин могат да се добавят прости константи (например да си дефинираме числото пи).
<call-template> - позволява прилагането на даден темплейт като се задава неговото име (<xsl:call-template name=”CreateHeader”/>)
<xsl:param> - специални променливи; позволява да се задават параметри на даден темплейт (стойността, посочена при декларацията на този елемент се счита по подразбиране). При извикването на темплейта с <call-template> може да се ползва елемента <xsl:with-param> и там да се зададе нова стойност. Стойностите на параметрите могат да се достъпват по същия начин като стойностите на променливите.
<xsl:element> - използва се за създаване на нови елементи и позволява динамично създаване на елементи (например съдържанието на елементите от входното дърво може да служи за имена на елементите от изходното дърво или атрибутите да се превърнат в елементи).
<xsl:attribute> - за създаване на атрибути
<xsl:attribute-set> - за създаване на групи от атрибути
<xsl:copy> - копира съдържанието на контекстния връх без децата и атрибутите; има атрибут use-attribute-sets, в който могат да се посочат групи от атрибути за копирания елемент.
<xsl:copy-of> - може да копира фрагменти от входното дърво без да се загубват атрибутите и вложените елементи (дълбоко копие)
<xsl:for-each> - позволява да се итерира по елементите на дадено множество от върхове
<xsl:if> - съдържа темплейт, който ще се приложи само ако условието в атрибута test е изпълнено
<xsl:choose> - използва се в комбинация с <xsl:when> и <xsl:otherwise> за изразяване на условни твърдения (multiple conditional statements)

XPath

XPath – схема за локализиране и идентифициране на подструктури в документи; използва се от други XML технологии като XSLT, XPointer, XQL като средство за локализиране на определени части от XML съдържанието.

XPath модел на данните – при XPath повечето части на един сериализиран XML документ се представят като върхове на дърво. Коренът на дървото е самият XML документ – отбелязва се с '/'. За всеки елемент, атрибут, инструкция към процесора или коментар има съответен връх в дървото. Също така има текстови върхове, представящи текстовото съдържание на елементите. XPath се възползва от последователния и йерархичен характер на XML документите, за да открие елементи според техния контекст (например къде в йерархията се намира даден елемент)

Контекстен възел – селектира се с .; това е текущата секция в XML документа, от където започва XPath изразът

XPath връх – може да бъде всяка част от XML документ – елемент, атрибут, инструкция към процесора или нещо друго

Корен на документа (document root) – не е първият елемент, а самият XML документ; означава се с '/'

XPath израз – текстов стринг, с който се избира даден елемент, атрибут, инструкция или текст; може да се намира в URL или като стойност на атрибут (http://abc.com/getQuery?/book/intro/title или <xsl:pattern match="chapter/title">… </xsl:pattern>). Изразите локализират нещата по тяхното място в XML йерархията

4 типа изрази – булеви, множества от върхове, числа, стрингове (това са типовете, които могат да бъдат върнати вследствие на оценката на XPath израза)

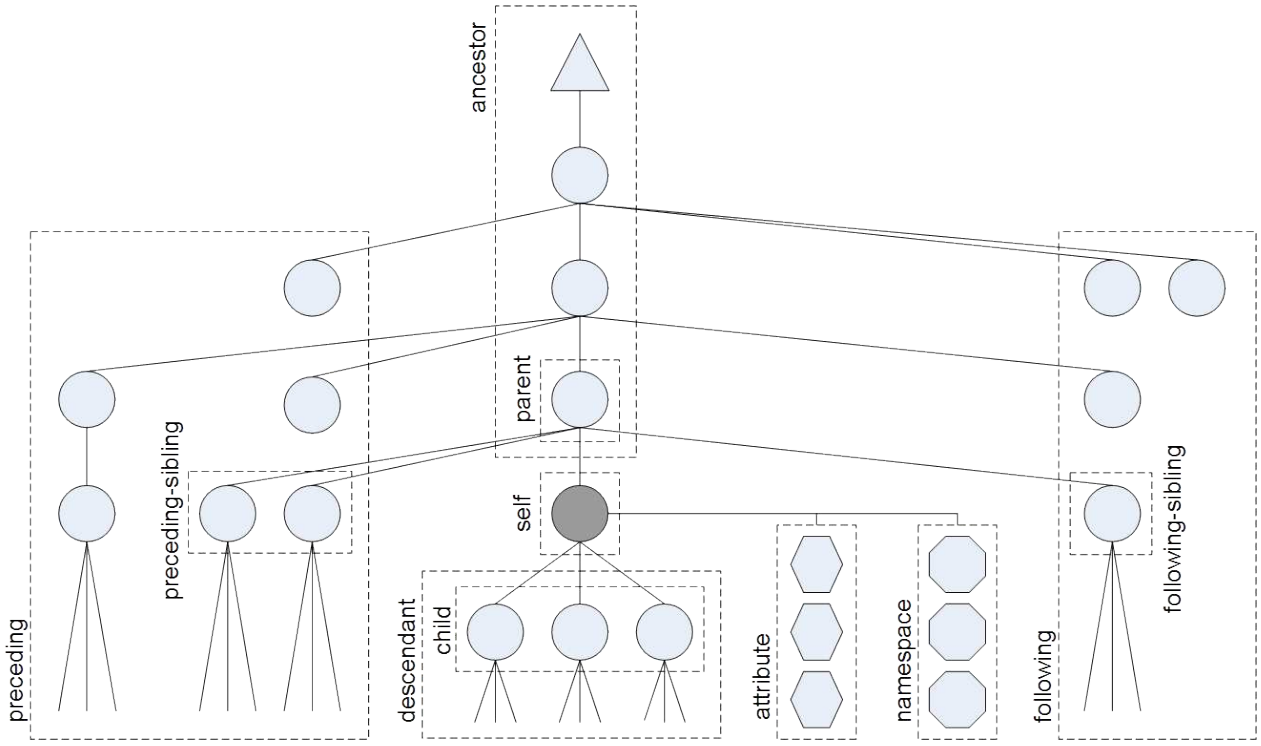
Location path – повечето XPath изрази връщат множество от върхове, тези изрази се наричат location paths. Пътищата, посочващи местонахождението (location paths), са съставени от стъпки. Всяка стъпка е съставена от ос, връх и предикат (опционален). Когато оста липсва, подразбира се child.

- ✓ Relative paths **–** започват от контекстния връх
- ✓ Absolute paths **–** започват от корена

Видове върхове

- ✓ Корен
- ✓ Връх на елемент
- ✓ Връх на атрибут – върхът на елемента, към който е този атрибут, се нарича връх родител. Всеки връх на атрибут има име и стойност.
- ✓ Текстов връх
- ✓ Връх за пространство от имена
- ✓ Връх за коментар
- ✓ Връх за инструкция към процесора

13 XPath оси – използват се така: име_на_оста::име_на_връх. Оста на контекстния връх се нарича self. Той може да се достъпи по два начина: '.' и self::node.



Заб: На картинката липсват descendant-or-self и ancestor-or-self. С тях не се селектират братя и сестри.

Рекурсивен оператор за достъп до всички потомци – '//'. Пример: chapter//para ще селектира всички para елементи, които са по-ниско в йерархията от chapter. Несъкратеният запис е: child::chapter/descendant-or-self::node()/child::para. Пример: За избиране на всички para елементи в текущия – .//para или descendant-or-self::node()/child::para.

С '..' се селектира родителят на контекстния връх, може да се достъпи и с parent оста.

С '@' се селектира стойността на атрибут

Функции

- ✓ node()
- ✓ text()
- ✓ position()
- ✓ last()
- ✓ count()
- ✓ id()
- ✓ concat()

Предикати – задават се с начупени скоби ([]). Предикат може да се сложи след всяка стъпка в XPath израза, като на една стъпка могат да бъдат зададени и няколко предиката на стъпка.

Използване на DOM (Document Object Model) и SAX (Simple API for XML) за обработка на XML документи – основни интерфейси на DOM и SAX и начини за използването им. Сравнение между DOM и SAX.

Има три основни стъпки при използването на XML парсер:

1. Създаване на обект от типа на парсера

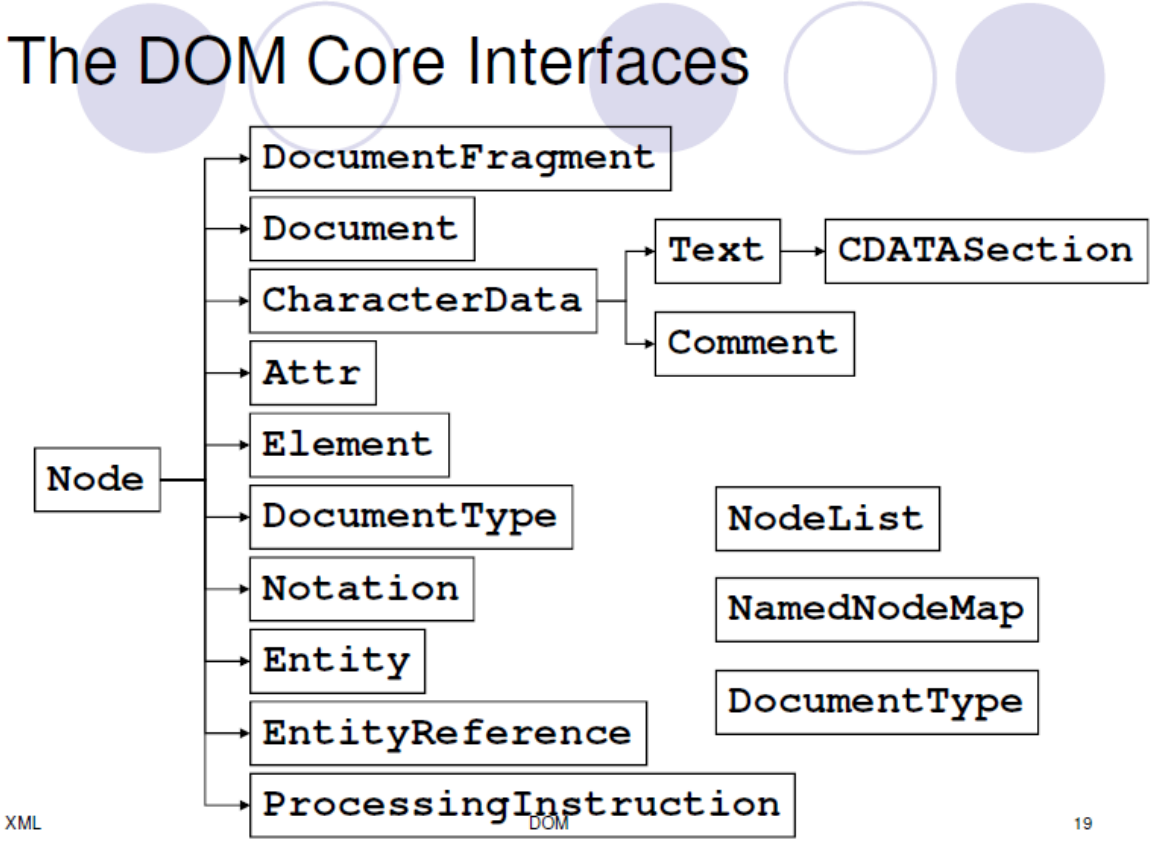
2. Подаване на XML документа към парсера
3. Обработка на резултатите

В общия случай генерирането на XML съдържание е извън възможностите на парсерите, но има и такива, които могат да поддържат и такава функционалност.
Двата главни програмни интерфейса (APIs) за парсване на XML са SAX и DOM.

DOM (Document Object Model)

DOM (Document Object Model) е W3C стандарт, който дефинира стандартен начин за работа с документи като XML и HTML. XML DOM дефинира обекти и свойства за всички XML елементи, както и методи (интерфейс) за достъп до тях. Платформено-независим модел. Използва се от програмистите основно като начин за манипулация на съдържанието на XML документи – може да се създава такова съдържание, да се навигира, да се модифицират, добавят и изтриват части от XML документи. Характерно за DOM е, че в паметта се зарежда целият документ, като за целта се парсва и му се съпоставят съответните обекти с техните свойства.
В DOM документът се представя като йерархия (дърво) от node обекти.
Спецификацията включва основна задължителна част от интерфейси, която трябва да бъде налична във всички имплементации, и различни разширения – за работа конкретно с XML, HTML.
Има три нива – всяко надгражда следващото; в DOM Level 2 е добавена функцията "getElementById", също така поддръжка на XML namespaces and CSS. DOM Level 3 е текущото ниво на спецификацията, добавена е поддръжка на XPath, обработка на събития от клавиатурата и интерфейс за сериализиране на XML документи.

Основни DOM интерфейси



Кой какви деца може да има:

- ✓ Document – Element (maximum of one), ProcessingInstruction, Comment, DocumentType (maximum of one)
- ✓ DocumentFragment – Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
- ✓ DocumentType – no children
- ✓ EntityReference – Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
- ✓ Element – Element, Text, Comment, ProcessingInstruction, CDATASection, EntityReference
- ✓ Attr – Text, EntityReference
- ✓ ProcessingInstruction – no children
- ✓ Comment – no children
- ✓ Text – no children
- ✓ CDATASection – no children
- ✓ Entity – Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
- ✓ Notation – no children

Node – това е основният тип данни за целия документен обектен модел. Представя един връх в документното дърво. Методите му се разделят в три категории:

- ✓ Характеристики на самия връх – име, тип, стойност
- ✓ Разположение в дървото и достъп до съседни върхове – родители, братя и сестри, деца, предшественици, потомци
- ✓ Модификация на върхове – редактиране, изтриване, пренареждане на върхове деца

Document – представя целия XML документ (корена на дървото); съдържа методи фабрики за създаване на останалите обекти (елементи, текстови върхове, коментари, инструкции за процесора.

DocumentFragment – фрагмент от документа може временно да се съхрани в такъв връх. Обектите от тип са „леки” (lightweight) и могат да се ползват примерно при разместване на части от XML документа. Въпреки че за целта би могло също да се ползват и обекти от тип Document, то тези обекти могат да бъдат „тежки” в зависимост от имплементацията. Някои методи (например за добавяне на деца на някой връх) могат да приемат DocumentFragmen обекти като аргументи. По този начин децата на обекта DocumentFragment се добавят към списък от деца на върха. DocumentNode обектите не са задължително добре-структурирани XML документи – например може да има само едно дете и то да е връх от тип Text.

Element – два типа методи общи и свързани с атрибутите на даден елемент. Методите, предоставящи интерфейс към атрибутите на елемента са относително просто – трябва да знаеш името на атрибута, не могат да се различат стойността по подразбиране, зададена в DTD, и реалната стойност в XML документа, не може да се определи кога типът на атрибута е [String] (Need name of attribute; Cannot distinguish between default value specified in DTD and given in XML file; Cannot determine attribute type [String]). По-добре е да се ползва методът, който връща NamedNodeMap от обекти Attr с атрибутите на дадения елемент.

Attr – предоставя интерфейс към обекти, съдържащи информация за атрибути

- ✓ String getName();
- ✓ String getValue();
- ✓ Void setValue(String value);

- ✓ Boolean getSpecified(); //показва къде е зададена стойността – дали в самия XML документ или е стойността по подразбиране от DTD

Референциите към Entity са деца на атрибутите. Атрибутите не се считат за част от дървото и затова parentNode, previousSibling и nextSibling имат стойност null за обектите от тип Attr. Обекти от този тип могат да се създават с метод фабрика в интерфейса на Document: Attr createAttribute(string Name);

CharacterData – има полезни методи за обработка на текст; не се използва директно, а през класовете наследници Text или Comment

- ✓ String getData() throws DOMException;
- ✓ void setData(String data) throws DOMException;
- ✓ int getLength()
- ✓ void appendData(String data) throws DOMException;
- ✓ String substringData(int offset, int length) throws DOMException;
- ✓ void insertData(int offset, String data) throws DOMException;
- ✓ void deleteData(int offset, int length) throws DOMException;
- ✓ void replaceData(int offset, int length, String data) throws DOMException;

Text – представя текстовото съдържание на обект от тип Element или Attr. Обикновено обектите от тип Text са деца на обекти от тип Element или Attr и винаги са листа в дървото. Има само още един метод в допълнение на интерфейса на CharacterData: Text splitText(int offset) throws DOMException;

За създаването на обекти от тип Text се използва метод фабрика от интерфейса на Document.

С извикването на normalize() метода на даден обект от тип Element се сливат всички текстови обекти, деца на елемента.

CDATASectioin – представя CDATA – текст, в който няма се интерпретира markup, разпознава се единствено “]]>”, който е краят на CDATA секцията. Няма добавени методи към интерфейса на CharacterData.

Comment – използва се да се представят коментарите в XML документа. Няма добавени методи към интерфейса на CharacterData.

ProcessingInstruction – представя декларациите на инструкции към процесора в XML документа. Тук името на върха е името на приложението, което трябва да се стартира, а стойността на върха е командата за приложението.

- ✓ String getDaga();
- ✓ Void setData(String data);
- ✓ String getTarget();

Entity – представя (represents) семантична единица (такава, която може да се парсва или не) в XML документа. XML процесорът може или да замени референция към семантична единица със самата нея, или да създаде връх от тип EntityReference. При семантични единици, които не могат да се парсват, референциите се запазват, а не се заменят със самата семантична единица. Към методите от интерфейса на Node са добавени още:

- ✓ String getPublicId();
- ✓ String getSystemId();
- ✓ String getNotationName();

Последният се ползва за семантични единици, които не могат да се парсват, в противен случай връща null. Entity, което може да се парсне, може да има деца, които представляват

стойността, с която може да бъде заменено entity. Всички семантични единици в даден XML документ могат да се вземат с getEntities() от интерфейса на DocumentType

EntityReference

Notation – всяка декларация на нотация в DTD се представя с Notation връх. Към методите на Node са добавени:

- ✓ String getPublicId(); //връща съдържанието на PUBLIC идентификатора
- ✓ String getSystemId(); //връща съдържанието на SYSTEM идентификатора

Всички нотации в даден документ могат да се вземат с getNotations() от DocumentType

NodeList –колекция от подредени обекти от тип Node

- ✓ int getLength();
- ✓ Node item(int index);

NamedNodeMap

- ✓ NamedNodeMap myAttributes = myElement.getAttributes();
- ✓ NamedNodeMap myEntities = myDocument.getEntities();
- ✓ NamedNodeMap myNotations = myDocument.getNotations();

Методи:

- ✓ int getLength();
- ✓ Node item(int index);
- ✓ Node getNamedItem(String name);
- ✓ Node setNamedItem(Node node) throws DOMException;
- ✓ Node removeNamedItem(String name) throws DOMException;

DOMImplementation – използва се, за да се определи нивото на поддръжка от DOM парсера

- ✓ hasFeature(String feature, String version);

DocumentType – съдържа DTD информацията за документа. В DOM 1.0 не се позволява промяна на този връх

DOM обектите могат да се сериализират, ако ще се предават отдалечено, за да се избегне повторно парсване на XML документа за създаване на обекти. Недостатъкът е, че обектите могат да бъдат по-големи от самия XML документ.

SAX (Simple API for XML)

- ✓ Разработен, за да позволи по-ефективна обработка и анализ на големи XML документи. Целта е да се разреши един от главните проблеми на DOM – за да се обхожда XML дървото, целият документ трябва да се зареди в паметта. Това води до заемане на голямо количество памет и също така отнема време за създаването на обектното дърво.
- ✓ Не се притежава от никой, възниква от една мейлинг листа (XML-DEV), която в момента се хоства от OASIS.
- ✓ Event-driven API
  - XML файлът се изпраща на SAX парсера
  - XML файлът се чете последователно
  - Парсерът нотифицира даден клас, когато се случи някакво събитие (включително грешки)
  - За обработка на събитията се имплементират методи от API-то
- ✓ SAX парсерът генерира събития:

- В началото и в края на документа
- В началото и в края на всеки елемент
- Когато намери символи в елемент
- Когато се натъкне на грешки
- Когато се натъкне на negligible whitespace
- В още няколко други случая

✓ Използва се callback механизъм за да се нотифицира приложението
✓ Всички SAX интерфейси се предполага, че са синхронни

- parse методите не трябва да връщат стойност преди парсването да е завършило
- readers трябва да чакат обработчика на събитието (това е callback) да върне резултат преди да продължат с докладването за следващото събитие

✓ Ползва се само за четене, но не и за създаване на XML съдържание
✓ Не е подходящ, ако трябва да се манипулира структурата на документа

Основни SAX интерфейси

DocumentHandler – за контролиране на събитията, свързани с обработката на документа; има го само в SAX1, в SAX2 отпада и е заменен от ContentHandler, който има поддръжка на пространства от имена.

ContentHandler – Използва се за получаване на основни събития от парсера. Има имплементация DefaultHandler (в SAX1 HadnlerBase) с празни имплементации на handler-ите; тя е за удобство, за да не се налага да се предоставят имплементации на всички методи от интерфейса, а само на тези, които са необходими в конкретното приложение. В DefaultHandler има имплементации на всички callbacks от четирите основни handler интерфейса:

✓ EntityResolver
✓ DTDHandler
✓ ContentHandler
✓ ErrorHandler

XMLFilterImpl – имплементация на ContentHandler; стои между XMLReader и обработчиците на събития в приложението – подава заявки към reader-а и събития към handler-ите без да ги модифицира, но подкласове могат да override-ват методите.

XMLReaderAdapter – wrapper за XMLReader – прави го като Parser от SAX1

XMLReader – интерфейс за четене на XML документ, използвайки callback функции. Въпреки името си, интерфейсът не разширява стандартния Java Reader интерфейс, защото тук става въпрос за различен тип действие от четенето на символни данни.

Позволява на приложението да задава и проверява характеристики и свойства на парсера, да регистрира обработчици на събития при обработката на документа и да инициира парсването на документа.

XMLReader замества отпадналия интерфейс Parser в SAX1. XMLReader има две съществени подобрения спрямо стария Parser интерфейс:

✓ Добавя стандартен начин за проверяване и задаване на характеристики и свойства на парсера.
✓ Добавена е поддръжка на пространства от имена.

void setContentHandler(ContentHandler handler)
void setDTDHandler(DTDHandler handler)

void setErrorHandler(ErrorHandler handler)
void setEntityResolver(EntityResolver resolver)
Парсването се започва с метода parse().
Ако приложението не регистрира DTD handler, то всички DTD събития просто се игнорират.
Handler-ите могат да се сменят по време на самото парсване и SAX парсерът веднага започва да ползва новите.
Когато има нещо, за което се генерира събитие, четенето спира и се изчаква обработчикът да върне резултат, след което се продължава отново.

lputSource

- ✓ void setByteStream(InputStream byteStream)
- ✓ void setCharacterStream(Reader characterStream)
- ✓ void setEncoding(String encoding)

EntityResolver – приложението не е наясно с физическата структура на XML данните. Парсерът има entity manager, който скрива сложността на физическата структура от приложението, което вижда всичко като един поток от данни.
Can intercept references to entities by implementing the EntityResolver interface
Ако приложението има нужда да имплементира собствена обработка на външни entities, трябва да имплементира този интерфейс и да регистрира инстанция, използвайки setEntiryResolver. Когато парсерът се натъкне на entity, той подава системния и/или публичния идентификатор на приложението: public InputSource resolveEntity(String publicID, String systemID)

ParserFactory – помощен клас, предоставя удобни методи за динамично зареждане на SAX парсери

- ✓ makeParser()
- ✓ makeParser(String className)

21. Използване на XML за структуриране, валидация, обработка и представяне на документно съдържание.

Добре структуриран XML

Основни концепции:

- XML таг – markup на документа, отделен от останалата част със символите < и >. Съдържа името на елемента
    - Start tag <elementname>
    - End tag </elementname>
- XML елемент – информацията между отварящ и затварящ таг включително. Информацията между таговете се нарича съдържание на елемента.
- XML Markup – състои се от таговете в документа.
- XML документ - съдържа текст под формата на един или повече елементи.
- XML атрибути – в допълнение към тагове и елементи, XML документите могат да съдържат и атрибути. Атрибутите са двойки име/стойност (стойността е задължителна дори да е празен стринг), асоциирани с елемент. Те се добавят към отварящия таг - <name nickname=”Smth”> и стойностите им задължително се поставят между ‘’ или „“.
- Processing instructions - <?....?> - представят информация, необходима на външни приложения.
- XML декларация - <?xml version=’1.0’ encoding=’UTF-16’ standalone=’yes’ ?> - указва версията на XML спецификацията, към която се придържа документа, енкодинга му и дали с него е асоцииран външен DTD.

XML Йерархии:

XML документите представляват йерархични дървета. Всеки документ може да бъде представен като дърво, в което отделните възели са XML елементи.
Root елемент – всеки XML документ има точно един елемент, който няма родител. Той е първият елемент в документа и съдържа всички други елементи.
Всеки XML документ съдържа следното:

- Prolog – stylesheet & document type декларации
- Instance – йерархията от елементи
- Additions – коментари <!—comment--> и processing instructions <?PITarget Status=”draft”?>

Синтакични правила

XML документите трябва да се придържат към определени правила, за да бъдат добре структурирани:

- Всеки отварящ таг трябва да има съвпадащ затварящ таг или да бъде self-closing таг.
- Таговете не могат да се застъпват, елементите трябва да са properly nested.
- XML документите могат да имат само един root елемент.
- Имената на елементите трябва да спазват XML конвециите.
- XML е case sensitive.
- XML запазва whitespace в PCDATA.
- В съдържанието на документите не се позволяват запазените от XML символи. Те се заменят с predefined entities или се поставят в <![CDATA[.....]]> секция.

XML пространства от имена

Namespace/пространство от имена – абстрактна нотация (категория) за група от имена. Едно име може да принадлежи само към една група. Създадени са с цел да се осигури уникалност между XML

елементите. За име на namespace в XML обикновено се използва URI. Във всеки документ може да се съдържа информация за множество пространства от имена.
Декларация на namespace – за да се асоциира определен елемент с namespace в деклаацията му се добавя атрибут xmlns - <pers:person xmlns:pers="http://www.wiley.com/pers"/>. Тази декларация важи за елемента, в който е поставена, както и за всички негови наследници. В един елемент могат да бъдат деклрирани няколко namespace-а.

XML валидация чрез Document Type Definitions (DTD)

Цели на валидирането

Синтактичните правила в XML се налагат от парсерите, но често е необходимо да има начини за разграничаване на различните XML-базирани езици(речници). Всеки документ обикновено се опитва да следва определен език/речник и са необходими методи за определяне на нивото на съответствие. За тези цели се използва валидация чрез DTD или XML Schema.
XML документ е валиден, ако съдържанието му отговаря на дефиницията на позволените елементи, атрибути и други части от документа.
Целта на един DTD е да декларира всички имената на всички елементи и атрибути, които се използват в XML документите, отговарящи на това DTD, типовете им, видовете елементи, как те се изолзват заедно и йерархията на документа. DTD-тата представляват шаблони за markup-а на документите и съдържат формалната дефиниция на съответния тип документ. DTD може да бъде механизъм за стандартизация и манипулация на документи/данни и обмяната им.

DTD структура

DTD се съхраняват във външни файлове, или в самия XML файл или и двете.
DTD се състоят от:

- Document Type Declaration (DOCTYPE) – информира парсера, че с документа има асоцииран DTD. Задължително трябва да е поставена в началото на документа
- Тяло на DTD – поставя се след DOCTYPE декларацията и съдържа декларациите на елементи, атрибути, entities и нотации.
- Затваряне на декларацията на DTD с "]>"

```
<!DOCTYPE name [                                    // Document Type Declaration
        <!ELEMENT name (first,  last)>              // Body
        <!ELEMENT first (#PCDATA)>
        <!ELEMENT last (#PCDATA)>
]>                                                  // End of DTD
```

DTD синтаксис

DTD-тата се състоят от декларации на елементи, атрибути към тях, entity-та и нотации. Форматът на декларациите е <! ...> или <! … [<! … >]>

Document Type Definition

<![DOCTYPE root_element SYSTEM file […] > и *<![DOCTYPE root_element PUBLIC fpi (system_id)[…] >* указва на парсера, че с XML документа е асоцииран DTD. Съдържа <![DOCTYPE, името на root елемента на документа, и идентификатор към файла с DTD-то. SYSTEM и име на файл указва, че DTDто, асоциирано с документа, се намира във file, а PUBLIC – указва публично DTD, достъпно на fpi и опционално задава локално DTD, което да се използва, в случай че публичното не е достъпно.

Декларация на елемент

<!ELEMENT first (#PCDATA)>

Състои се от три основни части

- ELEMENT декларация
- Име на елемент
- Модел на съдържание на елемента – дефинира допустимото съдържание на елемента. Има 4 вида модели на съдържание:
    - Елемент *<!ELEMENT name (surname) >*
    - Смесен - *<!ELEMENT name (#PCDATA)>*
    - Празен *<!ELEMENT br EMPTY>*
    - Any *<!ELEMENT description ANY>* - всеки елемент, деклриран в DTD може да бъде използван в description в какъвто и е да е ред, колкото и да е пъти.

Всеки елемент, който е посочен в модела на съдържание на елемента, трябва да има своя дефиниция в DTD.
Има 2 начина за определянето на елементите наследници:

- Последователности (sequences)- *<!ELEMENT contact (name, location, phone, knows, description)>* - указва, че елементът contact съдържа елементите name, location, phone, knows, description.
- Избор (choices) - *<!ELEMENT location (address|GPS)>* - указва, че елементът location съдържа елементът address или елементът GPS.

Кардналност на елемент (cardinality) – дефинира колко пъти елементът може да се появи в модела на съдържание. Всеки елемент може да има индикатор колко пъти се среща. Допустимите индикатори са [none], ?, +, *.

Декларация на атрибут

<!ATTLIST contacts source CDATA #IMPLIED>

Декларацията на атрибут се състои от следните 3 части:

- Ключовата дума ATTLIST
- Името на елемента, чийто атрибути се дефинират
- Списък с атрибути – състои се от:
    - Име на атрибута
    - Тип
    - Декларация на стойността

Типовете атрибути са CDATA, ELEMENT, ID, IDREF, IDREFS, ENTITY, ENTITIES, NMTOKEN, NMTOKENS.
Атрибутът може и да се дефинира като списък със допустими стойности.
<!ATTLIST doc lang (en|bg|ge) #IMPLIED>
DTD позволява да се зададе, че атрибутът има:

- Стойност по подразбиране – default - *<!ATTLIST doc lang (en|bg|ge) en >*
- Фиксирана стойност - #FIXED - *<!ATTLIST doc ver CDATA '1.0'>*
- Е задължителен - #REQUIRED *<!ATTLIST person nationality CDATA #REQUIRED>*
- Е опционален - #IMPLIED - *<!ATTLIST doc lang (en|bg|ge) #IMPLIED>*

Декларация на символни данни (character data)

Понякога текстът трябва да съдъжа неинтерпетирани markup символи. Това става със CDATA декларация:
<![CDATA[Press <<<ENTER>>>]]>

Декларация на Entities

XML документите могат да бъдат разпределени в някоко файла. Всяка единица информация се нарича entity, което има идентифициращо го име. Entities се дефинират с entity декларации и се използват с entity reference.
Има 4 вида entities:

- Вградени entities - *&, <, ', "*
- Character entities – подобни са на вградените, но не се декларират в DTD. Могат да се използват в документа без декларации чрз референции например - *©*
- General entities – декларират се в DTD преди да могат да бъдат използвани в документа. Обикновено се използват за създаването на заменим текст - *<!ENTITY source-text "Alabala">* и се използват чрез референция - *&source-text;*
- Parameter entities – не могат да се използват в съдържанието – единствено в DTD. Могат да е използват за създаването на DTD отняколко файла. Декларират се по следния начин - *<!ENTITY DefaultPhoneKind "Home">*. Реферират се чрез *%DefaultPhoneKind*

Условни секции

Състоят се от:

- Include декларации - *<!INCLUDE[…..]]>* - Включва секция DTD
- Ignore декларации - *<!IGNORE[…]]>* - Изключва секция DTD

Декларация на нотация

Описва външно не-XML entity. Може да указва и helper приложение и документация. Например

<!NOTATION name SYSTEM "external_id">
<!ATTLIST element_name attr_name NOTATION default_value>

XML валидация чрез XML Schema

Спецификации

XML Schema Спецификацията се разделя на 3 части:

- Part 0: Primer - съдържа въведение в XML Schema
- Part 1: Structures - дефинира структурата, дефинициите на типове, атрибути, елементи, групи от атрибути (attribute group), модели от групи (model group), нотации и анотации
- Part 2: Data types - определя типовете данни за елементи и атрибути. Дефинира фасети и възможните стойности за типовете.

Типове данни

XML Schema Recommendation дефинира следните видове данни:

- Built-in datatypes
  - примитивни типове данни – string, boolean, float, double, ID и пр.
  - Derived – language, IDREF, long, int, short и др.
- User-defined datatypes

Дефиниране на потребителски типове данни
Става по 3 начина:

- дефиниране от съществуваши типове данни чрез определянето на 1 или повече фасети.
  <datatype name="TelephoneNumber" source="string">
  <length value="8"/>
  <pattern value="\d{3}-\d{4}"/>
  </datatype>
- Derived Types - може да се прилага форма на subclassing на дефинициите на типове. Има 2 начина за това:
  - derive by extension - разширяване на родителския тип с повече елементи –
    <complexType name="Book" source="cat:Publication" derivedBy="extension">

- derive by restriction - ограничаване на родителския тип чрез ограничения върху възможните стойности или броя на occurrences.

  < complexType name= "SingleAuthorPublication" source="cat:Publication" derivedBy="restriction">

- <simpleType> декларации

Фасети

Фасетите контролират всички прости типове в XML Schemas. Фасет е качество или отличителна черта на <simpleType>. Има 12 ограничаващи фасети:

- minExclusive
- minInclusive
- maxExclusive
- maxInclusive
- totalDigits - брой цифри за числов тип
- fractionDigits - брой дробни цифри за числов тип
- length - брой елементи в списък или брой символи на стринг
- minLength
- maxLength
- enumeration
- whiteSpace - определя как ще се третира whitespace в типа
- pattern - позволява рестрикцията на символни типове чрез регулярни изрази
- precision
- scale
- encoding
- duration
- period

Др. вид фасети са фундаменталните (не могат да се променят):

- equal - всички типове осигуряват релация равенство
- order - някой типове осигуряват и order релация
- bounds - горна и долна граница
- cardinality
- numeric

Структури

Структурните типове се използват за описание на елементи, които имат асоциирани към тях елементи наследници или атрибути.

<element> декларации

При деклариране на елемент се задават името му и допустимото му съдържание:

<element
name=”name of the element”
type=”global type”
ref=”global element declaration”
form=”qualified or unqualified”
minOccurs=”non negative number”
maxOccurs=”non negative number or ‘unbounded’”
default=”default value”
fixed=”fixed value”>

Допустимото съдържание се определя от неговия тип. XML Schemas позволяват определянето на типа на елемента по 2 начина:

- създаване на локален тип - нямат <schema> като директен родител и могат да се ползват само в определен контекст.

- използване на глобален тип - глобалните декларации могат да се преизползват в XML Schema чрез добавянето на ref атрибут - <element ref="target:first"/>. Задължително глобалният тип трябва да е именован.

<complexType> декларации

Чрез тях се създават сложни потребителски типове. В рамките на дефиницията може да се укаже допустимото съдържание на елемента.

<complexType mixed="true or false" name="Name of complexType">

<simpleType> декларации

Винаги когато се декларира <simpleType> той трябва да се базира на съществуващ тип данни.

<simpleType name="name of the simpleType" final="#all or list or union or restriction">

```
<simpleType name="sku" base="xsd:string>
    <pattern value="\d{3}-[A-D]{4}"/>
</simpleType>
```

<group> декларации

Дефинират преизползваеми групи от елементи - чрез тях може лесно да се преизползват и комбинират цели модели на съдържание. Дефинициите на глобални групи трябва да са именовани - чрез добавянето на атрибут *name*. Могат да се дефинират групи единствено от елементи.

```
<group name="NameGroup">
        <!-- content model goes here -->
</group>
```

Към дефиницията на групата може да има атрибут order със следните допустими стойности:

- *choice* - инстанциите на групата съдържат някой от елементите
- *all* - инстанциите на групата съдържат всички елементи
- *seq* - - инстанциите на групата съдържат всички елементи в зададения ред

Модели на съдържание (Content Models)

В XML Schemas моделът на съдържание на елементите може да се дефинира чрез:

- <sequence> декларация - указва последователност от елементи, които се съдържат в реда на указването им.
  <sequence minOccurs="non negative number" maxOccurs="non negative number or unbounded">
- А <choice> декларация - могат да се зададат множество декларации наследници. В документа обаче само 1 от декларациите може да бъде използвана.
  <choice minOccurs="non negative number" maxOccurs="non negative number or unbounded">
- Референция към глобална <group> декларация -
  <group ref="global group definition"
  minOccurs="non negative number"
  maxOccurs="non negative number or unbounded">
- An <all> декларация - позволява декларирането на елементи, които могат да се появят в какъвто и да е ред.
  <all minOccurs="0 or 1" maxOccurs="1">
- Вътрешни модели на съдържание
- Element декларация
- Element wildcards
- any Елемент - Позволява всеки well-formed XML. Може да бъде ограничен чрез атрибут Namespace, който да указва от кой namespace са допустимите елементи.
  <xsd:any namespace="##any" minOccurs="0" maxOccurs="unbounded"/>

<attribute> декларации

Винаги са последни след декларациите на всички елементи.

<attribute name="name of the attribute"

type=”global type”
ref=”global attribute declaration”
form=”qualified or unqualified”
use=”optional or prohibited or required”
default=”default value”
fixed=”fixed value”>

Както при декларациите на елементи има 2 метода за деклариране на атрибути:

- създаване на локален тип
- използване на глобален тип

Декларациите на атрибути могат да са единствено прости типове (simple types). Може да се преизползват атрибути чрез реферирането на глобални декларации - <attribute ref=”contacts:kind”/>.

<attributeGroup> декларации

Чрез тях могат да се дефинират преизползваеми множества от атрибути. Глобалните <attributeGroup> декларации трябва да са именовани с name атрибут.
<attributeGroup name=”ContactsAttributes”>
<!-- attribute declarations go here -->
</attributeGroup>

Сравнение с DTD

DTD предоставят фундаменентална граматика за дефиниране на XML документ под формата на метаданни, които описват съдържанието на документа. XML Schema предоставя това плюс детайлен начин за дефиниране на данните, които могат и не могат да се съдържат.
Разликите между XML Schema и DTD са:

- DTD не поддържат namespace за разлика от XML Schema, която пълно поддържа Namespace Recommendation.
- DTD осигуряват възможности за дефиниране на заменим текст, външно съдържание и условна обработка (ENTYTY декларациите). Това не е възможно в XSD.
- DTD описват целия документ, schema-ите могат да дефинират само части от него.
- XSD има дефинирана система от типове и позволява дефинирането на потребителски типове данни, както и наследяване, енкапсулация и субституция.
- XSD е по-богат език за описание на елементи и атрибути. DTD поддържа 10 типа, XSD - 37+.
- DTD може да бъде част от XML документа, докато XSD е задължително в отеделен файл.
- Официалната дефиниция на “валиден XML” изисква DTD.
- XSD може да дефинира много повече ограничения от DTD.
- XSD използва XML синтаксис, DTD има отделен синтаксис. XSD синтаксисът е по-експресивен от този на DTD
- XSD има повече поддържащи средства (DOM, browsers)
- XSD има по-добър модел на съдържанието – напр. Може точно да се задава ограничението за брой срещания на даден елемент
- XSD позволява лесно създаване на сложни и преизползваеми модели на съдържание.
- XSD е съвместима с други XML технологии като Web Services, XQuery, XSLT...
- XSD поддържа DOM и DOM манипулации
- XSD поддържа не само глобални, но и локални имена
- XSD има неограничена възможност за разширяване

Използване на XSLT (eXtensible StyleSheet Language Transformations) и XPath за алокиране, манипулиране и представяне на XML съдържание.

XSLT се използва съвместно с XPath за алокиране, манипулиране и представяне на XML съдържание. XSLT представлява език за трансформация на XML документи в какъвто и да е текстово-базиран формат. Езикът позволява дефинирането на шаблони, коио да се приложат към определена част от XML документ. Към коя част да се приложат се задава чрез XPath изрази.

XSLT спецификацията се разделя на две – XSL Formatting objects (или само XSL) и XSL Transformations. Първото е език за форматиране на XML данни за представянето им на екран, хартия или друг медиен носител. Второто е език за трансформиране на XML документи в други документи.

Асоцииране на XML документ с XSLT Style sheet се извършва чрез инструкция за обработка - *<?xml-stylesheet href="myStyles.xsl" type="text/xsl" ?>*

XPath

XPath – схема за локализиране и идентифициране на подструктури в документи; използва се от други XML технологии като XSLT, XPointer, XQL като средство за локализиране на определени части от XML съдържанието.

XPath модел на данните – при XPath повечето части на един сериализиран XML документ се представят като върхове на дърво. Коренът на дървото е самият XML документ – отбелязва се с '/'. За всеки елемент, атрибут, инструкция към процесора или коментар има съответен връх в дървото. Също така има текстови върхове, представящи текстовото съдържание на елементите.

XPath се възползва от последователния и йерархичен характер на XML документите, за да открие елементи според техния контекст (например къде в йерархията се намира даден елемент)

Контекстен възел – селектира се с .; това е текущата секция в XML документа, от където започва XPath изразът

XPath връх – може да бъде всяка част от XML документ – елемент, атрибут, инструкция към процесора или нещо друго

Корен на документа (document root) – не е първият елемент, а самият XML документ; означава се с '/'

XPath израз – текстов стринг, с който се избира даден елемент, атрибут, инструкция или текст; може да се намира в URL или като стойност на атрибут (http://abc.com/getQuery?/book/intro/title или <xsl:pattern match="chapter/title">…</xsl:pattern>). Изразите локализират нещата по тяхното място в XML йерархията

4 типа изрази – булеви, множества от върхове, числа, стрингове (това са типовете, които могат да бъдат върнати вследствие на оценката на XPath израза)

Location path – повечето XPath изрази връщат множество от върхове, тези изрази се наричат location paths. Пътищата, посочващи местонахождението (location paths), са съставени от стъпки. Всяка стъпка е съставена от ос, връх и предикат (опционален). Когато оста липсва, подразбира се child.

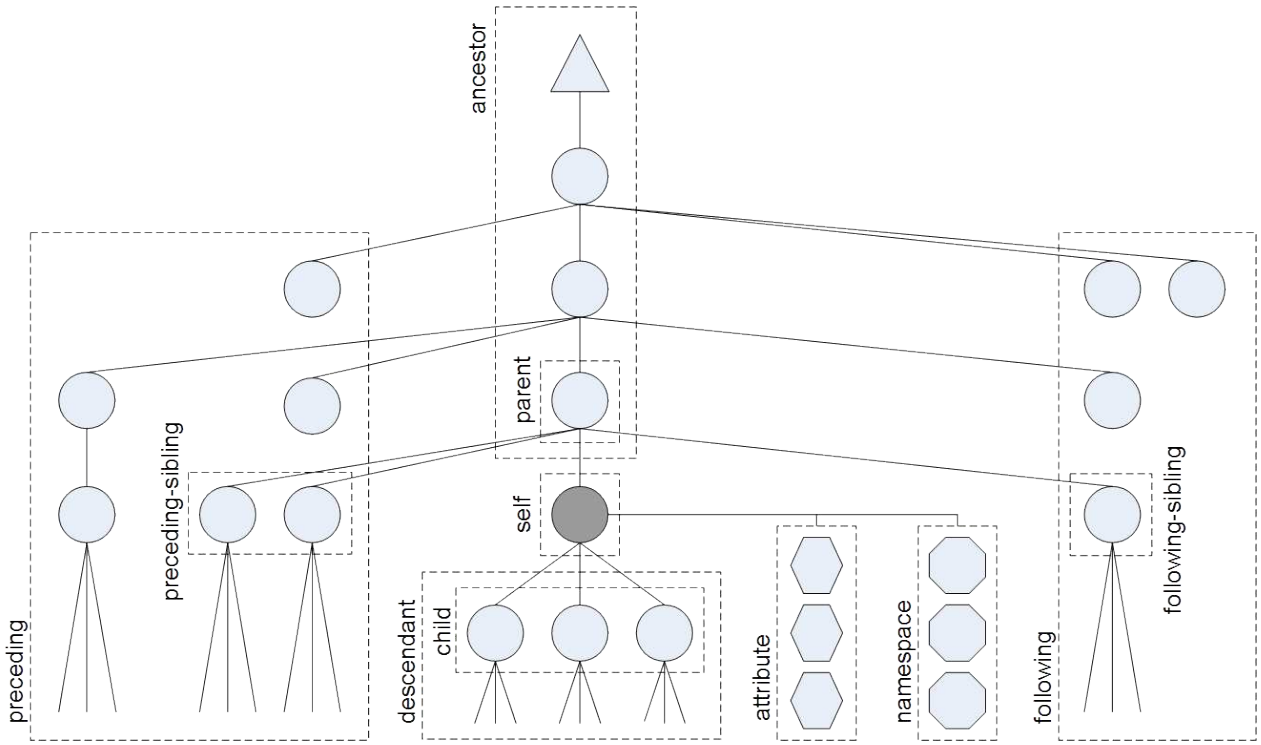
- ✓ Relative paths **–** започват от контекстния връх
- ✓ Absolute paths **–** започват от корена

Видове върхове

- ✓ Корен
- ✓ Възел на елемент
- ✓ Възел на атрибут – възелът на елемента, към който е този атрибут, се нарича възел родител. Всеки възел на атрибут има име и стойност.
- ✓ Текстов възел

- ✓ Възел за пространство от имена
- ✓ Възел за коментар
- ✓ Възел за инструкция към процесора

13 XPath оси – използват се така: име_на_оста::име_на_връх. Оста на контекстния връх се нарича self. Той може да се достъпи по два начина: '.' и self::node.



Заб: На картинката липсват descendant-or-self и ancestor-or-self. С тях не се селектират братя и сестри.

Рекурсивен оператор за достъп до всички потомци – '//'. Пример: chapter//para ще селектира всички para елементи, които са по-ниско в йерархията от chapter. Несъкратеният запис е: child::chapter/descendant-or-self::node()/child::para. Пример: За избиране на всички para елементи в текущия – .//para или descendant-or-self::node()/child::para.

С '..' се селектира родителят на контекстния връх, може да се достъпи и с parent оста.

С '@' се селектира стойността на атрибут

Функции

- ✓ node()
- ✓ text()
- ✓ position()
- ✓ last()
- ✓ count()
- ✓ id()
- ✓ concat()

Предикати – задават се с начупени скоби ([]). Предикат може да се сложи след всяка стъпка в XPath израза, като на една стъпка могат да бъдат зададени и няколко предиката на стъпка.

Манипулиране на XML съдържание

Реализира се чрез дефиниране на правилата за трансформация чрез елемент <xsl:template>. В атрибутът му match се задава XPath израз, който определя върху кои възли ще бъдат приложени шаблоните.

```
<xsl:template match="exon">
        We have found the EXON tag!
</xsl:template>
```

Елементът apply-templates позволява изливкването на шаблони от текущо изпълнявания такъв. Това предизвиква рекурсивната обработка на всички наследници на обработвания елемент.

```
<xsl:template match="p">
        <xsl:apply-templates/>
</xsl:template>
```

Извикване на темплейт се реализира чрез call-template елементът. XSLT позволява и предаването на параметри при извикването - <xsl:with-param name="prefix">new</xsl:with-param> - предава стойност на параметъра при извикването. Дефинирането на параметър става чрез param елемент.

```
<xsl:template match="title">
        <xsl:call-template name="CreateHeader" />
</xsl:template>
```

Изходните елементи могат да бъдат сортирани чрез xsl:sort елемент, който извършва сортировка по определено property.

```
<xsl:template match="list">
        <xsl:apply-templates>
                <xsl:sort select="@code"/>
        </xsl:apply-templates>
</xsl:template>
```

Итерация върху XML съдържание

<xsl:for-each> елементът може да селектира и итерира по всеки XML елемент от определено множество възли.

```
<xsl:for-each select="catalog/cd">
        .....................
</xsl:for-each>
```

Условна обработка на XML съдържание

Има 2 механизма за условна обработка:

- <xsl:if> елемент - съдържа шаблон, който да бъде приложен само ако зададеното условие е истина.

```
<xsl:if test="position() = 1">
        <xsl:attribute name="style">color: red</xsl:attribute>
</xsl:if>
```

- <xsl:choose> елемент заедно с <xsl:when> и <xsl:otherwise> - изразява множество условни тестове.

```
<xsl:choose>
        <xsl:when test="salary[number(.) &gt; 2000]">
                A big number
        </xsl:when>
        <xsl:when test="salary[number(.) &gt; 1000]">
                A medium number
        </xsl:when>
        <xsl:otherwise>A small number
        </xsl:otherwise>
</xsl:choose>
```

Деклариране на променливи

<xsl:variable> - могат да се декларират променливи; за да се достъпи стойност на променлива се ползва '$' (<xsl:value-of select="$colour"/>); също така стойността на променливата може да се ползва и в елементи от резултатното дърво (<ajr:glyph colour="{$colour}"/>). По този начин могат да се добавят прости константи (например да си дефинираме числото пи).

Алокиране на XML съдържание

XSLT позволява и създаването на елемент, атрибути и групи от атрибути в изходния документ. Това се реализира чрез xsl:element, xsl:attribute и xsl:attribute-set. Алокирането на динамични елементи и атрибути може да се реализира чрез задаването на XPath изрази за стойност или име.

```
<xsl:element namespace="html" name="lala">
<xsl:attribute name="style">purple</xsl:attribute>
```

Копиране на елементи

Има 2 възможности за копиране:

- Плитко – реализира се чрез copy елемент, който копира контекстния възел без наследниците и атрибутите му.

```
<xsl:template match="h1|h2|h3|h4|h5|h6|h7">
    <xsl:copy>
            Header: <xsl:apply-templates/>
    </xsl:copy>
</xsl:template>
```

- Дълбоко – елементът copy-of копира контекстния възел заедно с наследниците и атрибутите му.

```
<xsl:copy-of select="//h1 | //h2" />
```

Извличането на информация от XML дървото се реализира чрез елемента <xsl:value-of>. Той взима информация от обработвания XML и я добавя към резултатното дърво. В атрибута select се указва чрез XPath израз какво да бъде селектирано, като се включват всички наследници. Този елемент свежда селектирания обек до символен низ.

```
<xsl:template match="b">
```

```
        <xsl:value-of select="." />
</xsl:template>
```

Представяне на XML съдържание

Тъй като XML данните обикновено не съдържат елементи, които определят как да се презентират данние, е необходимо да се използват други методи за представяне. Един от тях е XSL документи, трансформиращи XML. Представянето на данните се реализира чрез асоцииране на XSL документ към XML документа и обработката на XML съдържанието от XSLT процесора по зададените критерии.

Контрол върху изходния документ се получава чрез <xsl:output>, който позволява задаването на формата, версията, енкодинга и др.

Използване на DOM (Document Object Model) и SAX (Simple API for XML) за обработка на XML документи

Основни интерфейси на DOM и SAX и начини за използването им

Има три основни стъпки при използването на XML парсер:

1. Създаване на обект от типа на парсера
2. Подаване на XML документа към парсера
3. Обработка на резултатите

В общия случай генерирането на XML съдържание е извън възможностите на парсерите, но има и такива, които могат да поддържат и такава функционалност.

Двата главни програмни интерфейса (APIs) за парсване на XML са SAX и DOM.

DOM интерфейси

DOM дефинира няколко следните Java интерфейси:

- DOMImplementation - Интерфейс за създаване на началния Document възел на DOM дървото.
- Node Interface - Представя основния обект в DOM - единичен възел в DOM дървото. Той е базовият интерфейс за всички други в DOM и осигурява достъп до общи пропъртита на възли, методи за обхождане и модификация на дървото.
- Document::Node Interface - Представлява целия XML документ (корена на дървото). Освен това се използва за factory за други типове възли - елементи, текстови възли, коментари, CDATA секции, processing инструкции, атрибути, entity референции. Има методи за достъп до DOCTYPE, до информация за възможностите на DOM имплементацията, до корена на дървото и пр.
- DocumentType::Node interface - Предоставя достъп до Entity и Notation колекциите в документа, както и до някой аспекти на външните и вътрешни DTD.
- Element::Node Interface - Съдържа 2 категории методи:
  - общи методи за елементи - за получаване на името на таг, на елементите по таг, нормализиране
  - методи за манипулиране на атрибути - за взимане и промяна на атрибут, премахване на такъв и пр.
- Attr::Node Interface - моделира атрибутите в XML документ, осигурявайки достъп до различни properties на атрибута. Поддържа методи за достъп до име и стойност и промяната им. Въпреки че extend-ва Node интерфейсът, атрибутите не се считат за част от DOM дървото.
- Entity::Node Interface - представя вътрешно или външно entity в XML документ.

- DocumentFragment::Node Interface - е signature interface. Той съхранява части от Document, когато се добави към друг Node се премахва. Не е нобходимо да се съобразява с правилата за XML.

SAX Интерфейси

SAX API са разделени на следните 4 области:

- core interfaces - облекчават работата с core информацията в XML документ
- core classes - облекчават работата с core информацията в XML документ
- extended interfaces - моделират аспекти на документа, които не представляват интерес за повечето програмисти - DTD декларации, коментари и пр.
- helper classes - съдържат няколко класа за удобство, както и имплементациите по подразбиране на някои от core интерфейсите.

Attributes - моделира атрибутите на елементи. Те се предоставят чрез ненаредено множество от свойства, което може да бъде обхождано по име и позиция.

ContentHandler - основен SAX интерфейс. Той моделира core информацията за XML документ като подредена последователност от извиквания на методи и предоставя интерфейс за получаване на основни markup събития от парсера.

DTDHandler - предоставя callback методи за получаване на нотификации за DTD събития.

EntityResolver - интерфейс, позволяващ на имплементациите да осигуряват custom резолюция на външни entities.

ErrorHandler - моделира well-formed грешки, грешки при валидация и warrinig-и. Консуматорът на ContentHandler имплементацията използва този интерфейс да преустанови потокът от извиквания на методи, които довеждат до грешки. ErrorHandler дефинира 3 нива exceptions - error, fatalError, warning.

Locator - парсерът може да предостави Locator обект, който да бъде използван в event методите, за да се разбере къде точно в документа е текущия метод.

XMLFilter - подинтерфейс на XMLReader. Повечето SAX интерфейси позволяват pipeline обработка, където имплементацията на ContentHandler може да получава някой информационни елементи, които разпознава, но да предава неразпознатите елементи на следващият процесор във веригата. Това се реализира чрез XMLFilter интерфейс. Той работи върху събия вместо върху документа и се регистрира при предишния филтър

XMLReader - представлява интерфейс за четене на XML документ чрез използването на callback механизъм. Позволява на приложенията да задават и получават свойства в парсера, да се регистрират event handler-и за обработка на документ и да се инициира парсването на документа.

Сравнение между DOM и SAX

DOM създава дърво за целия документ в паметта, докато SAX парсва документа възел по възел. DOM е подходящ, в случай че е необходимо да се обработва голяма част от елементите в документа и да се извършват операции върху XML. Подходящ е и при необхдимост за множество достъпи до XML. SAX е подходящ при големи XML документи, в които за обработката са необходими малко елементи, идеален е за извършването на прости операции върху XML файлове. SAX е разработен с цел да предостави по-ефективен анализ на големи документи. Той е подходящ за обработка по време на парсването на документа, тъй като единствено последния event е в паметта. Обработката със SAX изисква по-малко място и време, тъй като не се построява карта на целия документ в паметта, и е подходящ за сложни структури. Тъй като SAX е създаден единствено за четене на XML, но не и за писане при необходимост от това се използва DOM.
DOM е подходящ за приложения, който изискват структурни манипулации върху много XML tokens, а SAX - за приложения с ограничения в паметта.

27. Теорема за средните стойности (Рол, Лагранж, Коши). Формула на Тейлър

$\xi \subset M$

Дефиниция. Всеки отворен интервал, съдържащ точката a ще наричаме околност на точката a. Ще казваме, че точката a е вътрешна точка за интервала M, ако съществува околност ξ на точката a и ;

$(f(x) \geq f(c), \forall x \in \delta)$

Дефиниция. Ще казваме, че функцията f(x) има локален максимум (минимум) в точка C от своята дефиниционна област, ако съществува околност δ на точката C, за която . Ако функцията f има локален максимум или локален минимум ще казваме, че f има локален екстремум.

Теорема на Ферма.

Ако функцията y = f(x) е диференцируема в една вътрешна точка C от своята дефиниционна област и има локален екстремум в тази точка, то f’ (c) =0.

Теорема на Вайерщрас.

Ако дадена функция f(x) е непрекъсната в крайния и затворен интервал [a, b], тя достига в този интервал точната си горна и точната си долна граница.

Теорема на Рол.

Ако функцията f е непрекъсната в крайния и затворен интервал [a, b], диференцируема в отворения интервал (a, b) и f(a) = f(b), тогава съществува ξ принадлежаща на (a, b) и f’(ξ) = 0.

Доказателство:

Тъй като функцията f(x) е непрекъсната в [a, b], то тя е ограничена. Да означим съответно с L и l точната й горна и точната й долна граница в интервала [a, b].

Ако l = L, то поради неравенството l<=f(x)<=L, изпълнени за всяко x от интервала [a, b], функцията f(x) ще бъде константа в този интервал. Тогава нейната производна е нула в целия интервал (a, b) и теоремата е доказана.

Остава да разгледаме случая, когато l < L. Съгласно теоремата на Вайерщрас съществуват две точки x_1 и x_2 от интервала [a,b] такива, че $f(x_1)=l$ и $f(x_2) = L$. Поне една от тези две точки е вътрешна за интервала [a,b]. И наистина ако допуснем противното, т.е ако имаме $x_1 = a$, $x_2 = b$ или пък $x_1 = b$, $x_2 = a$, то от условието бихме получили l = L. Нека x_1 е вътрешна за интервала [a,b]. Но $f(x_1)=l$. Тогава функцията f(x) ще има локален минимум в точката x_1, поради което съгласно теоремата на Ферма ще имаме $f'(x_1) = 0$. Ако пък x_1 е крайна точка за интервала [a,b], то точката x_2 ще бъде вътрешна. В такъв случай това ще бъде една точка на локален максимум и следователно пак по теоремата на Ферма ще имаме $f'(x_2) = 0$. И така във всички случаи съществува точка ξ принадлежаща на [a, b], за която f’(ξ) = 0.

Теорема на Лагранж (формула на Лагранж за крайните нараствания)

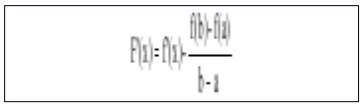
$\in (a,b)$

Ако функцията f е непрекъсната в крайния и затворен интервал [a, b], диференцируема в отворения интервал (a, b), то съществува ξ , за която f(b) – f(a) = (b-a)f’(ξ).

Доказателство:

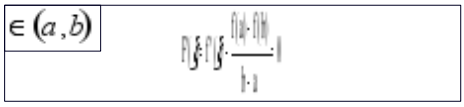
$\frac{(\cdot)((b)-f(a))}{b-a}$

Нека F(x) = f(x) – f(a) –



F(x) е непрекъсната в [a, b], като разлика на непрекъсната в този интервал функция f(x) и линейна функция и във всяка вътрешна точка на [a, b] има производна равна на:

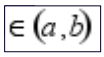
Имаме F(a) = F(b) = 0 и за функцията F(x) са изпълнени всички условия от теоремата на Рол.



Следователно съществува ξ такава, че

Или f(b) – f(a) = (b-a)f'(ξ), с което теоремата е доказана.

<u>Теорема на Коши за крайните нараствания.</u>



$\neq 0$

Ако f(x) и g(x) се непрекъснати в [a, b] и диференцируеми в (a, b) и g'x за всяко x (a, b),

то съществува ξ , за която

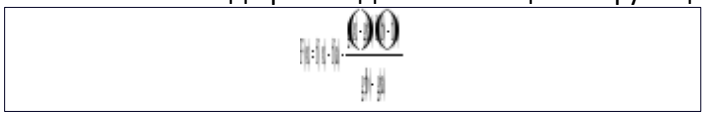
$$\frac{f(b)-f(a)}{g(b)-g(a)}=\frac{f'(\xi)}{g'(\xi)}$$

- обобщена формула на крайните нараствания на Коши.

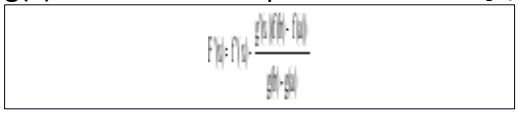
<u>Доказателство:</u>

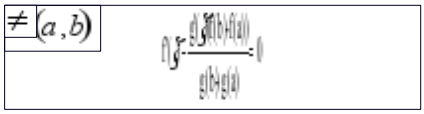
Ще докажем, че g(a) g(b). Да приемем, че g(a) = g(b), следователно за g(x) в интервала [a, b] и е в сила теоремата на Рол. Следователно съществува точка ξ (a, b), такава че g'(ξ) =0, което е противоречие с условието. Следователно g(a) g(b).

Можем да разгледаме помощната функция F(x).



F(x) е непрекъсната в [a, b], като разлика на непрекъснати в този интервал функции f(x) и g(x) и във всяка вътрешна точка на [a, b] има производна равна на:





Имаме F(a) = F(b) = 0 и за функцията F(x) са изпълнени всички условия от теоремата на Рол. Следователно съществува ξ такава, че от условието g'(ξ)0

$$\frac{f'(\xi)}{g'(\xi)}=\frac{f(b)-f(a)}{g(b)-g(a)}$$

, което доказва теоремата.

Формулата на Лагранж е частен случай от формулата на Коши при g(x) =x.

Теорема на Тейлър.

Да предположим, че функцията f(x) притежава производна до ред (n+1) в н якоя околност δ на една точка a от нейната дефиниционна област. За всяко x принадлежащо на δ съществува точка ξ между x и a такава, че:

$$\frac{x-a}{1!}f'(a)+\frac{(x-a)^2}{2!}f''(a)+\ldots+\frac{(x-a)^n}{n!}f^n(a)+\frac{(x-a)^{n+1}}{(n+1)!}f^n(\xi)$$

f(x) =f(a) + - формула на Тейлър.

Доказателство:

Да разгледаме функциите:

$$g(t)=f(t)+\frac{f'(t)(x-t)}{1!}+\frac{f''(t)(x-t)^2}{2!}+\ldots+\frac{f^n(t)(x-t)^n}{n!}$$

$h(t) = (x-t)^{n+1}$

$$g'(t)=f'(t)+\frac{f''(t)(x-t)}{1!}-\frac{f'(t)}{1!}+\frac{f'''(t)(x-t)^2}{2!}-\frac{f''(t)(x-t)^2}{2!}+\ldots$$
$$+\frac{f^{n+1}(t)(x-t)^n}{n!}-\frac{f^n(t)(x-t)^n}{(n-1)!}=\frac{f^{n+1}(t)(x-t)^n}{n!}$$

$h'(t) = -(n+1)(x-t)^n$

За функциите h(t) и g(t) прилагаме теорема на Коши за крайните в интервал (a,x) или (x,a).

$$\frac{g(x)-g(a)}{h(x)-h(a)}=\frac{g'(\xi)}{h'(\xi)}$$

h'(t) 0 за всяко t (a, x) или (x, a) и h(t) и g(t) са непрекъснати и диференцируеми в (a, x) и (x, a) следователно съществува т.ξ (a, x) или (x, a), така че: от дефиницията:

$$f(x)-f(a)-\frac{x-a}{1!}f'(a)-\frac{(x-a)^2}{2!}f''(a)-\ldots-\frac{(x-a)^n}{n!}f^n(a)-\frac{(x-a)^{n+1}}{(n+1)!}f^n(\xi)$$

g(x) – g(a) =

$h(x) – h(a) = -h(a) = -(x-a)^{n+1}$

$$g'(\xi)=\frac{f^{(n+1)}(\xi)(x-\xi)^n}{n!}$$

$h'(\xi)= -(n+1)(x-\xi)^n$

$$\Rightarrow g(x)-g(a)=\frac{g'(\xi)}{h'(\xi)}(h(x)-h(a))$$

или

$$\frac{x-a}{1!}f'(a)+\frac{(x-a)^2}{2!}f''(a)+\ldots+\frac{(x-a)^n}{n!}f^n(a)+\frac{(x-a)^{n+1}}{(n+1)!}f^n(\xi)$$

f(x) =f(a) + , което доказва теоремата

Определен интеграл. Дефиниция и свойства. Интегруемост на непрекъснати функции. Теорема на Нютон-Лайбниц.

Фиксирана е функцията $f:[a,b]\to\mathbb{R}$.

Под <u>разбиване</u> на интервала $[a,b]$ разбираме редицата от точки $x_0=a, x_1, \dots, x_n=b$, такива че $x_0<x_1<x_2<\dots<x_n$. Под <u>диаметър</u> на разбиването разбираме числото $d(\{x_i\}) = \max\limits_{1\le i\le n}(x_i - x_{i-1})$, т.е. дължината на най-голямия подинтервал, определен от разбиването. Сумата $\sigma(\{x_i\},\{t_i\}) = \sum\limits_{i=1}^{n} f(t_i).(x_i - x_{i-1})$, където $\{x_i\}$ е някакво разбиване на интервала $[a,b]$, а t_i са междинни точки, т.е. $t_i\in[x_{i-1},x_i]$ за $i=1..n$, се нарича <u>Риманова интегрална сума</u> на f, определена от разбиването $\{x_i\}$ и междинните точки $\{t_i\}$.

Геометрично е ясно, че Римановата интегрална функция представлява сума от лица на правоъгълници, които апроксимира лицето на криволинейния трапец, определен от графиката на f, абсцисната ос и правите успоредни на ординатната ос, преминаващи през точките $(a,0)$ и $(b,0)$.

Казваме, че функцията f е <u>интегруема по Риман</u> (интегруема в Риманов смисъл) в интервала $[a,b]$, ако съществува число I, такова че Римановите интегрални суми клонят към I, при условие, че диаметърът на използваните разбивания клони към 0, т.е. за $\forall\varepsilon>0$, съществува $\delta>0$, такова че за всяко разбиване $\{x_i\}$ с $d(\{x_i\})<\delta$ и всеки избор на междинните точки t_i е изпълнено неравенството:

$$|\sigma(\{x_i\},\{t_i\}) - I| < \varepsilon$$

Ако f е интегруема в Риманов смисъл, лесно се вижда (както единствеността на границата на една сходяща редица), че числото I е еднозначно определено и се нарича Риманов (определен) интеграл на f от a до b и се означава

$$I = \int_a^b f(x)dx$$

Твърдение:
Ако f е интегруема в Риманов смисъл в интервала $[a,b]$, то f е ограничена в $[a,b]$.

Доказателство:
Нека ε е произволно положително число и $I = \int_a^b f(x)dx$. Образуваме разбиване $\{x_i\}$ на $[a,b]$, такова че да имаме $|\sigma(\{x_i\},\{t_i\}) - I| < \varepsilon$ при всеки избор на междинните точки $\{t_i\}$. От неравенството на триъгълника имаме че

$$|x-y| + |y| \geq |(x-y)+y| = |x|, \text{ т.е. } |x-y| \geq |x| - |y|.$$

Получаваме:

$$\varepsilon > |\sigma(\{x_i\},\{t_i\}) - I| = \left|\sum_{i=1}^{n} f(t_i)(x_i - x_{i-1}) - I\right| =$$

$$= \left| f(t_k).(x_k - x_{k-1}) + \sum_{i=1}^{k-1} f(t_i)(x_i - x_{i-1}) + \sum_{i=k+1}^{n} f(t_i)(x_i - x_{i-1}) - I \right| =$$

$$= \left| f(t_k).(x_k - x_{k-1}) - \left(I - \sum_{i=1}^{k-1} f(t_i)(x_i - x_{i-1}) - \sum_{i=k+1}^{n} f(t_i)(x_i - x_{i-1})\right)\right| \geq$$

$$\geq \left| f(t_k).(x_k - x_{k-1})\right| - \left| I - \sum_{i=1}^{k-1} f(t_i)(x_i - x_{i-1}) - \sum_{i=k+1}^{n} f(t_i)(x_i - x_{i-1})\right|$$

$$\Rightarrow |f(t_k)| \leq \frac{\varepsilon + \left| I - \sum_{i=1}^{k-1} f(t_i)(x_i - x_{i-1}) - \sum_{i=k+1}^{n} f(t_i)(x_i - x_{i-1})\right|}{x_k - x_{k-1}}$$

Сега фиксираме t_i при $i \neq k$, и оставяме t_k да се мени в интервала $[x_{k-1}, x_k]$. От получеността неравенство получаваме, че функцията f е ограничена в подинтервала $[x_{k-1}, x_k]$, определен от разбиването $\{x_i\}$. Тъй като можем да даваме на k стойности $1, 2, \ldots, n$ заключаваме, че f е ограничена в целия интервал $[a,b]$.

Предполагаме, че е фиксирана функция f, дефинирана и ограничена в интервала $[a,b]$. Нека имаме разбиване $\{x_i\}$ на $[a,b]$. Полагаме:

$$m_i = \inf_{x\in[x_{i-1},x_i]} f(x), \quad i=1,2,\dots,n$$

$$M_i = \sup_{x\in[x_{i-1},x_i]} f(x), \quad i=1,2,\dots,n$$

Ще отбележим, че числата m_i и M_i са добре дефинирани, тъй като f е ограничена в $[a,b]$, а следователно и във всеки подинтервал на $[a,b]$, определен от разбиването $\{x_i\}$.

Сумите $s(\{x_i\}) = \sum_{i=1}^{n} m_i(x_i - x_{i-1})$ и $S(\{x_i\}) = \sum_{i=1}^{n} M_i(x_i - x_{i-1})$ наричаме съответно малка и голяма сума на Дарбу.

Тъй като $m_i \le f(t_i) \le M_i$ за всяко $t_i \in [x_{i-1}, x_i]$, то имаме че $m_i(x_i - x_{i-1}) \le f(t_i)(x_i - x_{i-1}) \le M_i(x_i - x_{i-1})$
$\Rightarrow s(\{x_i\}) \le \sigma(\{x_i\}, \{t_i\}) \le S(\{x_i\})$.

Твърдение:

Нека функцията f е дефинирана и ограничена в интервала $[a,b]$. Нека τ е разбиване, определено от точките $\{x_i\}$ и x' е точка различна от x_i за $i=1,2,\dots,n$. Нека $\tau^* = \tau \cup \{x'\}$. Тогава:

1) $s(\tau^*) \ge s(\tau)$

2) $S(\tau^*) \le S(\tau)$

3) Ако $M = \sup_{x\in[a,b]} f(x)$, то $S(\tau) - S(\tau^*) \le 2.M.d(\tau)$ и $s(\tau^*) - s(\tau) \le 2.M.d(\tau)$.

4) Ако $M = \sup_{x\in[a,b]} f(x)$ и $m = \inf_{x\in[a,b]} f(x)$, то е вярно че: $S(\tau) - S(\tau^*) \le (M-m)d(\tau)$ и $s(\tau^*) - s(\tau) \le (M-m)d(\tau)$

Доказателство:

Нека $x' \in [x_{i-1}, x_i]$. Такова i има единствено. Нека $m' = \inf_{x\in[x_{i-1},x']} f(x)$, $m'' = \inf_{x\in[x',x_i]} f(x)$. Ясно е, че $m' \ge m_i$ и $m'' \ge m_i$. Получаваме:

$$s(\tau^*) - s(\tau) = m'(x' - x_{i-1}) + m''(x_i - x') - m_i(x_i - x_{i-1}) =$$
$$= (m' - m_i)(x' - x_{i-1}) + (m'' - m_i)(x' - x_{i-1}) \ge 0$$

Тъй като $m' - m_i \le M - m$, $m'' - m_i \le M - m$, $x' - x_{i-1} < x_i - x_{i-1} \le d(\tau)$ и $x_i - x' < x_i - x_{i-1} \le d(\tau)$, получаваме че

$$s(\tau^*) - s(\tau) = (m' - m_{i,m})(x' - x_{i-1}) + (m'' - m_i)(x_i - x') \le$$

$$\le (M-m)(x' - x_{i-1}) + (M-m)(x_i - x') =$$

$$= (M-m)(x' - x_{i-1} + x_i - x') = (M-m)(x_i - x_{i-1}) \le (M-m)d(\tau)$$

$$\Rightarrow 0 \le s(\tau^*) - s(\tau) \le (M-m)d(\tau).$$

Аналогично, нека $M' = \sup_{x \in [x_{i-1}, x']} f(x)$ и $M'' = \sup_{x \in [x', x_i]} f(x)$.

Ясно е, че $M'' \le M_i$ и $M' \le M_i$. Получаваме:

$$S(\tau) - S(\tau^*) = M_i(x_i - x_{i-1}) - M'(x' - x_{i-1}) - M''(x_i - x') =$$

$$= (M_i - M')(x' - x_{i-1}) + (M_i - M'')(x_i - x') \ge 0.$$

Тъй като $M_i - M' \le M - m$, $M_i - M'' \le M - m$ и $x_i - x_{i-1} \le d(\tau)$ получаваме:

$$S(\tau) - S(\tau^*) = (M_i - M')(x' - x_{i-1}) + (M_i - M'')(x_i - x') \le$$

$$\le (M-m)(x' - x_{i-1}) + (M-m)(x_i - x') =$$

$$= (M-m)(x' - x_{i-1} + x_i - x') = (M-m)(x_i - x_{i-1}) \le (M-m)d(\tau)$$

$$\Rightarrow 0 \le S(\tau) - S(\tau^*) \le (M-m)d(\tau),$$ с което твърдението е доказано.

По-общо, ако τ^* се получава от τ чрез добавяне на k нови точки, то $0 \le S(\tau) - S(\tau^*) \le k(M-m)d(\tau)$ и $0 \le s(\tau^*) - s(\tau) \le k(M-m)d(\tau)$.

Твърдение:
Всяка малка сума на Дарбу не надминава коя да е голяма сума на Дарбу.

Доказателство:
Нека $s(\{x_i\})$ е малка сума на Дарбу, определена от разбиването $\{x_i\}$. Нека $S(\{y_j\})$ е голяма сума на Дарбу, определена от разбиването $\{y_j\}$. Нека τ е обединението на двете разбивания. Тогава с помощта на предходното твърдение получаваме следните неравенства:

$s(\{x_i\}) \le s(\tau)$ и $\forall S(\tau) \le S(\{y_j\})$

От друга страна е ясно че $s(\tau) \le S(\tau)$ за всяко разбиване τ. Следователно:

$s(\{x_i\}) \le s(\tau) \le S(\tau) \le S(\{y_j\})$, с което теоремата е доказана.

От горното твърдение, че множеството от големите суми на Дарбу за f е ограничено отдолу от кое да е малка сума на Дарбу, освен това то е непразно, така че то има точна долна граница. Точната ~~горна~~ долна граница на големите суми на Дарбу за f се нарича **горен интеграл** на функцията f в интервала $[a,b]$ и се означава с $\overline{I}$.

Аналогично, от горното твърдение получаваме, че множеството от всички малки суми на Дарбу е ограничено отгоре от кое да е голяма сума на Дарбу. Освен това то ~~не~~ е празно, следователно има точна горна граница. Тази граница се нарича **долен интеграл** на функцията f в интервала $[a,b]$ и се означава с $\underline{I}$.

Казваме, че една функция f е интегруема в смисъл на Дарбу, ако горният интеграл на f е равен на долния, т.е. $\underline{I} = \overline{I}$.

Твърдение:

Нека f е дефинирана и ограничена в интервала $[a,b]$. Тогава горният интеграл на f в интервала $[a,b]$ е равен на границата на големите суми на Дарбу, когато диаметърът на разбиването на интервала $[a,b]$ клони към 0, т.е. за всяко $\varepsilon > 0$ съществува $\delta > 0$, такова че за всяко разбиване ~~δ~~ τ на интервала $[a,b]$ с $d(\tau) < \delta$ имаме $S(\tau) - \overline{I} < \varepsilon$.

Доказателство:

Фиксираме $\varepsilon > 0$. Тогава $\overline{I} + \frac{\varepsilon}{2}$ не е точна горна граница на ~~не~~ големите суми на Дарбу

$\Rightarrow$ съществува разбиване τ_0 на $[a,b]$, такова че $S(\tau_0) < \overline{I} + \frac{\varepsilon}{2}$. Нека τ_0 има k делящи точки. Да изберем числото δ така че да бъде изпълнено:

$$0 < \delta < \frac{\varepsilon}{2k(M-m)}$$

където $M = \sup_{x\in[a,b]} f(x)$ и $m = \inf_{x\in[a,b]} f(x)$.

Нека τ е такова разбиване, че $d(\tau) < \delta$. Като използваме бележката след по-предното твърдение получаваме:

$$S(\tau) - S(\tau \cup \tau_0) \leq k(M-m)\,d(\tau) < k(M-m).\delta < k(M-m).\frac{\varepsilon}{2k(M-m)} = \frac{\varepsilon}{2}$$

Освен това, $S(\tau \cup \tau_0) \leq S(\tau_0) < \overline{I} + \frac{\varepsilon}{2}$

$$\Rightarrow S(\tau \cup \tau_0) - \overline{I} < \frac{\varepsilon}{2}$$

Сега като съберем двете неравенства получаваме че:

$$S(\tau) - \overline{I} < \frac{\varepsilon}{2} + \frac{\varepsilon}{2} = \varepsilon.$$

Твърдение:

Нека f е дефинирана и ограничена в $[a,b]$. Тогава долният интеграл $\underline{I}$ на f в интервала $[a,b]$ е равен на границата на малките суми на Дарбу, когато диаметърът на разбиването на интервала $[a,b]$ клони към 0, т.е. за всяко $\varepsilon > 0$, съществува $\delta > 0$, такова че за всяко разбиване τ на интервала $[a,b]$ с $d(\tau) < \delta$ имаме $\underline{I} - s(\tau) < \varepsilon$.

Доказателство:

Аналогично на доказателството на предходното твърдение.

Твърдение:

Нека f е дефинирана и ограничена в $[a,b]$. Тогава f е интегруема в смисъл на Дарбу $\Leftrightarrow$ за всяко $\varepsilon > 0$, съществува разбиване τ, такова че $S(\tau) - s(\tau) < \varepsilon$.

Доказателство:

$\Rightarrow$ Нека f е интегруема в смисъл на Дарбу. и $I = \overline{I} = \underline{I}$. Избираме $\varepsilon > 0$. От дефиницията за горен и долен интеграл следва че съществува разбиване τ', такова че $S(\tau') - \overline{I} < \varepsilon/2$ и разбиване τ'', такова че $I - s(\tau'') < \varepsilon/2$.

Нека $\tau = \tau' \cup \tau''$. Тогава $S(\tau) \le S(\tau') \Rightarrow S(\tau) - I < \frac{\varepsilon}{2}$

Също така $s(\tau) \ge s(\tau'') \Rightarrow I - s(\tau) < \frac{\varepsilon}{2}$

Следователно (след събиране на двете неравенства):

$$S(\tau) - s(\tau) < \frac{\varepsilon}{2} + \frac{\varepsilon}{2} = \varepsilon.$$

($\Leftarrow$) Нека $\varepsilon > 0$ и да намерим разбиване τ, такова че $S(\tau) - s(\tau) < \varepsilon$. Изпълнени са равенствата

$$s(\tau) \le \underline{I} \le \overline{I} \le S(\tau)$$

$$\Rightarrow \overline{I} - \underline{I} \le S(\tau) - s(\tau) < \varepsilon$$

Последното неравенство е изпълнено за всяко $\varepsilon > 0 \Rightarrow \underline{I} = \overline{I} \Rightarrow f$ е интегруема в смисъл на Дарбу

Теорема:
Нека f е дефинирана в интервала $[a, b]$. Тогава f е интегруема в Риманов смисъл ($\Leftrightarrow$) f е ограничена и интегруема в смисъл на Дарбу.

Доказателство:

($\Leftarrow$) Нека f е ограничена и интегруема в смисъл на Дарбу. Тогава $I = \underline{I} = \overline{I}$. В сила са неравенствата:

$$s(\{x_i\}) \le \sigma(\{x_i\}, \{t_i\}) \le S(\{x_i\})$$

за всяко разбиване $\{x_i\}$. В тях извършваме граничен преход при $d(\{x_i\}) \to 0$, използваме двете твърдения от по-горе и получаваме

$$s(\{x_i\}) \to I, \; S(\{x_i\}) \to I$$

$\Rightarrow \sigma(\{x_i\}, \{t_i\}) \to I$. Следователно f е интегруема по Риман.

($\Rightarrow$) Нека f е интегруема по Риман. Тогава от първото твърдение получаваме, че f е ограничена в $[a, b]$. Фиксираме $\varepsilon > 0$. Избираме $\delta > 0$, такова че за всяко разбиване $\{x_i\}$ с $d(\{x_i\}) < \delta$, да имаме

$$|\sigma(\{x_i\}, \{t_i\}) - I| < \frac{\varepsilon}{6} \Leftrightarrow I - \frac{\varepsilon}{6} < \sigma(\{x_i\}, \{t_i\}) < I + \frac{\varepsilon}{6}$$

за всеки избор на междинните точки $\{t_i\}$.
Нека $\alpha = \frac{\varepsilon}{3(b-a)} > 0$. Тъй като $M_i = \sup_{x \in [x_{i-1}, x_i]} f(x)$, то $M_i - \alpha$ не е супремум. Следователно съществуват междинни точки t_i', такива че $f(t_i') > M_i - \alpha$ за $\forall i$.

Образуваме съответната римanова сума със същото разбиване $\{x_i\}$ и междинните точки t_i'. Имаме:

$$\sigma(\{x_i\},\{t_i'\}) > S(\{x_i\}) - \alpha(b-a) = S(\{x_i\}) - \frac{\varepsilon}{3}$$

$$\Rightarrow S(\{x_i\}) - \frac{\varepsilon}{3} < I + \frac{\varepsilon}{6} \Rightarrow S(\{x_i\}) < I + \frac{\varepsilon}{2}.$$

Аналогично, тъй като $m_i = \inf_{x\in[x_{i-1},x_i]} f(x)$, тогава $m_i+\alpha$ не е инфимум. Следователно съществуват междинни точки t_i'', такива че $f(t_i'') < m_i+\alpha$ за всяко i. Образуваме съответната римanова сума със същото разбиване $\{x_i\}$ и междинни точки t_i''. Имаме:

$$\sigma(\{x_i\},\{t_i\}) < s(\{x_i\}) + \alpha\cdot(b-a) = s(\{x_i\}) + \frac{\varepsilon}{3}$$

$$\Rightarrow s(\{x_i\}) + \frac{\varepsilon}{3} > I - \frac{\varepsilon}{6} \Rightarrow s(\{x_i\}) > I - \frac{\varepsilon}{2}.$$

$$\Rightarrow \left| \begin{array}{l} S(\{x_i\}) < I + \frac{\varepsilon}{2} \\ s(\{x_i\}) > I - \frac{\varepsilon}{2} \end{array} \right. \Rightarrow S(\{x_i\}) - s(\{x_i\}) < \frac{\varepsilon}{2} + \frac{\varepsilon}{2} = \varepsilon.$$

~~Така получихме~~

С помощта на предходното твърдение и неравенството, което получихме сега, стигаме до извода, че f е интегруема по Дарбу.

С това теоремата е доказана.

От предходните теорема и твърдение получаваме, че следните условия за една функция f, дефинирана в интервала $[a,b]$, са еквивалентни:

1) f е интегруема в Римanов. смисъл
2) f е ограничена и интегруема в смисъл на Дарбу.
3) f е ограничена и за всяко $\varepsilon>0$ съществува такова разбиване τ на $[a,b]$, че $S(\tau)-s(\tau)<\varepsilon$.

Теорема на Кантор:

Ако $f(x)$ е дефинирана и непрекъсната в крайния затворен интервал $[a,b]$, то $f(x)$ е равномерно непрекъсната в $[a,b]$.

Теорема:

Нека $f(x)$ е дефинирана и непрекъсната в крайния затворен интервал $[a,b]$. Тогава $f(x)$ е интегруема в Риманов смисъл.

Доказателство:

Преди всичко от теоремата на Вайерщрас f е ограничена, така, че е достатъчно да докажем, че за $\forall \varepsilon > 0$ съществува разбиване τ, такова че

$$S(\tau) - s(\tau) < \varepsilon.$$

От теоремата на Кантор следва че $f(x)$ е равномерно непрекъсната в $[a,b]$. Фиксираме $\varepsilon > 0$. Да изберем такова $\delta > 0$, че от $x', x'' \in [a,b]$ и $|x'-x''| < \delta$ да следва че $|f(x') - f(x'')| < \frac{\varepsilon}{b-a}$. Образуваме разбиване $\{x_i\}$ на $[a,b]$, такова че $d(\{x_i\}) < \delta$. Разглеждаме интервала $[x_{i-1}, x_i]$. От теоремата на Вайерщрас $M_i = f(x')$ и $m_i = f(x'')$ за някои $x', x'' \in [x_{i-1}, x_i]$. Тъй като $x', x'' \in [x_{i-1}, x_i]$, то $|x'-x''| < d(\{x_i\}) < \delta$.

Следователно $|f(x') - f(x'')| < \frac{\varepsilon}{b-a} \Rightarrow M - m < \frac{\varepsilon}{b-a}$

Получаваме:

$$S(\{x_i\}) - s(\{x_i\}) = \sum_{i=1}^{n} (M_i - m_i)(x_i - x_{i-1}) <$$

$$< \frac{\varepsilon}{b-a} \sum_{i=1}^{n} (x_i - x_{i-1}) = \frac{\varepsilon}{b-a} \cdot (x_n - x_0) = \frac{\varepsilon}{b-a}(b-a) = \varepsilon.$$

Така доказахме, че $f(x)$ е интегруема в Риманов смисъл.

Свойства на Римановия интеграл:

① Ако интегрумата в интервала $[a,b]$ функция $f(x)$ удовлетворява неравенствата $m \le f(x) \le M$, то

$$m \le m(b-a) \le \int_a^b f(x)\,dx \le M(b-a)$$

② Ако функцията $f(x)$, дефинирана в интервала $[a,b]$, е интегруема, а C е константа, то $C \cdot f(x)$ е също интегруема и при това

$$\int_a^b C \cdot f(x)\,dx = C \cdot \int_a^b f(x)\,dx$$

③ Ако функциите $f(x)$ и $g(x)$, дефинирани в интервала $[a,b]$, са интегруеми, то и тяхната сума $f(x)+g(x)$ е интегруема функция и при това:

$$\int_a^b [f(x)+g(x)]\,dx = \int_a^b f(x)\,dx + \int_a^b g(x)\,dx$$

④ Ако функцията $f(x)$ и $g(x)$ са интегруеми в интервала $[a,b]$ и ако $f(x) \le g(x)$ за всяко x от този интервал, то

$$\int_a^b f(x)\,dx \le \int_a^b g(x)\,dx$$

⑤ Ако функцията $f(x)$ е интегруема в интервала $[a,b]$, то интегруема е и функцията $|f(x)|$, като при това:

$$\left| \int_a^b f(x)\,dx \right| \le \int_a^b |f(x)|\,dx$$

⑥ Ако $f(x)$ е дефинирана и ограничена в $[a,b]$, а c е една вътрешна точка за този интервал, то $f(x)$ е интегруема в $[a,b]$ тогава и само тогава, когато е интегруема в $[a,c]$ и $[c,b]$, като при това

$$\int_a^b f(x)\,dx = \int_a^c f(x)\,dx + \int_c^b f(x)\,dx$$

⑦ Ако функциите $f(x)$ и $g(x)$ са дефинирани в $[a,b]$ и интегруеми, то и тяхното произведение е интегруема функция.

Теорема:

Нека $f(x)$ е дефинирана и непрекъсната в затворения интервал $[a,b]$. Тогава съществува $t \in [a,b]$, такова че:

$$\int_a^b f(x)\,dx = f(t)\cdot(b-a)$$

Доказателство:

От теоремата на Вайерщрас, следва че f е ограничена. Нека $m = \inf_{x\in[a,b]} f(x)$, $M = \sup_{x\in[a,b]} f(x)$.

За всяко $x \in [a,b]$ имаме:

$$m \le f(x) \le M$$

$$\Rightarrow \int_a^b m\,dx \le \int_a^b f(x)\,dx \le \int_a^b M\,dx$$

$$\Rightarrow m(b-a) \le \int_a^b f(x)\,dx \le M\cdot(b-a)$$

$$\Rightarrow m \le \frac{\int_a^b f(x)\,dx}{b-a} \le M$$

От теоремата на Вайерщрас $m = f(x_1)$ и $M = f(x_2)$ за някои $x_1, x_2 \in [a,b]$. От теоремата за междинните стойности съществува t между x_1 и x_2 такова че $f(t) = \frac{\int_a^b f(x)\,dx}{b-a}$

$$\Rightarrow \int_a^b f(x)\,dx = f(t)\cdot(b-a).$$

Теорема (Лайбниц-Нютон)

Нека $f:[a,b]\to\mathbb{R}$ е непрекъсната функция. Тогава функцията $F(x) = \int_a^x f(t)\,dt$ е диференцируема и нейната производна е $f(x)$ за всяко $x\in[a,b]$.

Доказателство:

Нека $x\in[a,b]$. Записваме диференчното частно на $F(x)$: ($x+h\in[a,b]$)

$$\frac{F(x+h)-F(x)}{h} = \frac{\int_a^{x+h} f(t)\,dt - \int_a^x f(t)\,dt}{h} =$$

$$= \frac{1}{h}\cdot\left[\int_x^{x+h} f(t)\,dt\right]$$

Съгласно горната теорема съществува точка $u_h \in [x, x+h]$, такава че:

$$f(u_h) = \frac{1}{h} \cdot \left[\int_x^{x+h} f(t)\,dt \right]$$

Следвайки известния граничен преход при $h \to 0$ получаваме че

$u_h \to x \Rightarrow f(u_h) \to f(x)$, тъй като $f(x)$ е непрекъсната $\Rightarrow \frac{F(x+h) - F(x)}{h} \to f(x) \Rightarrow F(x)$ е диференцируема и $F'(x) = f(x)$ за всяко $x \in [a, b]$.

Да предположим, че трябва да пресметнем интеграла $\int_a^b f(x)\,dx$, където $f(x)$ е непрекъсната функция. Образуваме функцията $F(x) = \int_a^x f(t)\,dt$. Тогава очевидно търсеното число е $F(b)$.

Нека $\Phi(x)$ е примитивна на $f(x)$ в $[a, b]$. От теоремата на Нютон-Лайбниц $F(x)$ също е примитивна на $f(x)$ в $[a, b]$. Следователно съществува константа C, такава че $F(x) = \Phi(x) + C$, за всяко $x \in [a, b]$. При $x = a$ получаваме:

$F(a) = \Phi(a) + C \Rightarrow 0 = \Phi(a) + C \Rightarrow C = -\Phi(a)$

Получихме, че $F(x) = \Phi(x) - \Phi(a)$ за всяко $x \in [a, b]$.

Следователно $F(b) = \Phi(b) - \Phi(a)$

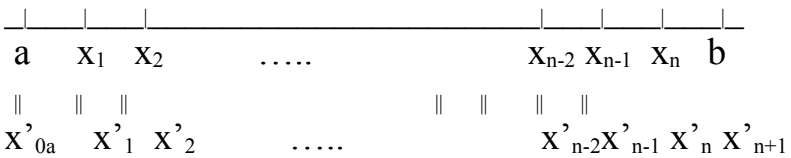
$$\Rightarrow \int_a^b f(x)\,dx = \Phi(x)\Big|_a^b,$$

където Φ е произволна примитивна функция на $f(x)$ в $[a, b]$

23. Определен интеграл. Дефиниция и свойства. Интегруемост на непрекъснати функции. Теорема на Нютон – Лайбниц.

Def. Нека f(x) е ограничена върху интервала [a, b] и $\tau = \{x_i\}^{\infty}_{i=0}$ е разбиване на интервала [a, b],
$m_i = \inf_{x\in[x_{i-1},x_i]} f(x)$
$M_i = \sup_{x\in[x_{i-1},x_i]} f(x)$, тогава:
$s_\tau = \sum_{i=1}^{n} m_i * \Delta x_i$ – малка сума на Дарбу
$S_\tau = \sum_{i=1}^{n} M_i * \Delta x_i$ – голяма сума на Дарбу

Th. За всяко разбиване τ и τ', за които τ < τ' (по броят на интервалите в разбиването) => малките суми на Дарбу са по – малки от големите суми на Дарбу, съответно за двете разбивания. ($s_\tau < s_{\tau'} < S_{\tau'} < S_\tau$)
Док.



Нека $\tau = \{x_i\}^{n}_{i=0}$ и $\tau' = \{x'_i\}^{n+1}_{i=0}$, но
$x'_i = x_i$ за всяко i = 0, 1, …, n-1, където $x'_n \in [x_{i-1}, x_i]$
⇨ $x'_{n+1} = x_n = b$
$s_\tau = \sum_{i=1}^{n} m_i * \Delta x_i$, където $m_i = \inf_{x\in[x_{i-1},x_i]} f(x)$ за всяко i = 0, 1, …, n-1
$S_\tau = \sum_{i=1}^{n} M_i * \Delta x_i$, където $M_i = \sup_{x\in[x_{i-1},x_i]} f(x)$ за всяко i = 0, 1, …, n-1
Тогава :
$s_\tau = \sum_{i=1}^{n+1} m'_i * \Delta x'_i = s_\tau = \sum_{i=1}^{n-1} m'_i * \Delta x'_i + m_n * \Delta x'_n + m_{n+1} * \Delta x'_{n+1}$
$m'_{n-1} = \inf_{x\in[x'_{n-1},x'_n]} f(x)$ и $m'_n = \inf_{x\in[x'_n,x'_{n+1}]} f(x)$
⇨ $\inf_{x\in[x'_{n-1},x'_{n+1}]} f(x) = m_n$
1. $m'_{n-1} >= m_n$
2. $m'_n >= m_n$

$s_{\tau'} >= \sum_{i=1}^{n-1} m_i * \Delta x_i + m_n * \Delta x'_n + m_n * \Delta x'_{n+1} = \sum_{i=1}^{n-1} m_i * \Delta x_i + m_n * (\Delta x'_n + \Delta x'_{n+1}) = \sum_{i=1}^{n-1} m_i * \Delta x_i + m_n * \Delta x_n = \sum_{i=1}^{n} m_i * \Delta x_i = s_\tau$
⇨ $s_\tau <= s_{\tau'}$
⇨ Аналогично се доказва, че $S_{\tau'} <= S_\tau$

Теорема1: Ако към един начин на делене на интервала [a, b] на подинтервали добавим краен брой нови точки, малката сума на Дарбу не намалява , а голямата не нараства.

Деф. Нека f(x) е дефинирана върху [a, b] и f(x) >= 0. Взимаме набор от точки $\tau = \{x_i\}^{\infty}_{i=0}$ и $a = x_0 < x_1 < \ldots < x_n = b$. Разбиване на интервали [a, b] : $[x_0, x_1], [x_1, x_2], \ldots, [x_{n-1}, x_n]$

$\Delta x_i = x_i - x_{i-1}$, за всяко i = 1, 2, …, n

$\delta_\tau = \max_{1<=i<=n} \Delta x_i$ – големина на разбиването τ

$c_i \in [x_{i-1}, x_i]$, за i = 1, 2, …, n

$\sum_{i=1}^{n} f(ci) * \Delta x_i = \sigma_\tau(f; c)$ – сума на Риман

Деф. Едно число I се нарича определенн интеграл на Риман от функцията f(x) върху интервала [a, b] ако за всяко ε > 0, съществува

$\delta = \delta(\varepsilon) > 0$ и за всяко $\tau = \{x_i\}^{\infty}_{i=0}$, $\delta_\tau < \delta$ и за всяко $c = \{c_i\}^{n}_{i=1}$

- $c_i \in [x_{i-1}, x_i]$, за i = 1, 2, …, n
- $|\sigma_\tau(f; c) - I| < \varepsilon$
- $I = \int_a^b f(x)dx$ - Определен интеграл на Риман

$I = \int_a^b f(x)dx = lim_{\delta\tau \to 0}\, \sigma\tau(f; c)$

Th. Критерий за интегруемост (Необходимо условие)

Нека f(x) е интегруема върху интервала [a, b] => f(x) е ограничена върху интервала [a, b].

Док. Допускаме, че f(x) не е ограничена върху интервала [a, b], тъй като f(x) е интегруема върху интервала [a, b].

- Съществува I ∈ R и за всяко ε > 0, съществува $\delta = \delta(\varepsilon) > 0$ и за всяко $\tau = \{x_i\}^{\infty}_{i=0}$, $\delta_\tau < \delta$ и за всяко $c = \{c_i\}^{n}_{i=1}$,

$c_i \in [x_{i-1}, x_i]$, за i = 1, 2, …, n

Нека за определеност ε = 1

- Съществува δ_1 и за всяко $\tau = \{x_i\}^{\infty}_{i=0}$, $\delta_\tau < \delta_1$ и за всяко $c = \{c_i\}^{n}_{i=1}$, $c_i \in [x_{i-1}, x_i]$, за i = 1, 2, …, n
- $|\sigma_\tau(f; c) - I| < 1$

  $I - 1 < \sigma_\tau(f; c) < I + 1$

  $I - 1 < \sum_{i=1}^{n} f(ci) * \Delta x_i < I + 1$

Фиксираме $\tau = \{x_i\}^{\infty}_{i=0}$, $\delta_\tau < \delta_1$, по допускане f(x) не е ограничена върху интервала [a, b] => f(x) не е ограничена върху поне един от интервалите $[x_{i-1}, x_i]$.

Без ограничение на общността нека считаме, че f(x) не е ограничена върху $[x_0, x_1]$

За всяко $\{c_i\}^{n}_{i=2}$ – фиксирано : $c_i \in [x_{i-1}, x_i]$ (i = 2, 3, … n)

За всяко $\hat{c}_1 \in [x_0, x_i]$ => $\hat{c} = \{ \hat{c}_1, c_2, \ldots, c_n \}$

- $I - 1 < \sigma_\tau(f; c) < I + 1$

  $I - 1 < f(\hat{c})* \Delta x_1 + \sum_{i=2}^{n} f(ci) * \Delta x_i < I + 1$, където $\sum_{i=2}^{n} f(ci) * \Delta x_i = \sum_0$

<=> $\frac{1}{\Delta x1}(I - 1 - \sum_0) < f(\hat{c}) < \frac{1}{\Delta x1}(I + 1 - \sum_0)$, но $I - 1 - \sum_0$ – фиксирано и $I + 1 - \sum_0$ - фиксирано

⇨ f(c) е ограничена върху интервала [a, b], противоречие с допуснатото.

Деф: $\bar{I}$ = inf S , S ∈ Q наричаме горен интеграл на Дарбу s <= $\bar{I}$, т.е P е ограничено отгоре и има точна горна граница.

Деф : $\underline{I}$ = sup s, s ∈ P наричаме долен интеграл на Дарбу.

Деф : Функцията f е ограничена и интегруема в Риманов смисъл в [a, b] ,ako $\bar{I} = \underline{I}$ и I = $\bar{I} = \underline{I} = \int_a^b f(x)dx$ се нарича определен Риманов интеграл на f в [a, b].

Th. Критерий за интегруемост на функция
Нека f(x) е дефинирана и ограничена върху крайния затворен интервал [a, b]. Тогава функцията f(x) е интегруема в Риманов смисъл върху крайния затворен интервал [a, b] ⇔ за всяко ε > 0, съществува δ = δ(ε) >0 и за всяко τ, $\delta_{(\varepsilon)} < \delta \Rightarrow S_\tau - s_\tau < \varepsilon$
Док.
=>|
Нека функцията f(x) е интегруема върху крайния затворен интервал [a, b], т.е.
Съществува I Є R, такова че за всяко ε > 0, съществува δ = δ(ε) >0 и за всяко τ = $\{x_i\}^n_{i=1}$ върху крайния затворен интервал [a, b] за всяко i = 0, 1, …, n
$\delta_\tau < \delta$ и за всяко c = $\{c_i\}^n_{i=}1$, такова че ci Є $[x_{i-1}, x_i]$, за i = 1, 2, …, n
⇨ $|\sigma_\tau(f; c) - I| < \varepsilon/3 \Leftrightarrow I - \varepsilon/3 < \sigma_\tau(f; c) < I + \varepsilon/3$
Фиксираме τ и разглеждаме c:
$\sup_c \sigma_\tau(f; c) <= I + \varepsilon/3$
⇨ $I - \varepsilon/3 <= \inf_c \sigma_\tau(f; c) = s_\tau <= S_\tau = \sup_c \sigma_\tau(f; c) <= I + \varepsilon/3$
⇨ $I - \varepsilon/3 <= s_\tau <= S_\tau <= I + \varepsilon/3$
⇨ $I - \varepsilon/3 <= s_\tau <= I + \varepsilon/3$
+
$I - \varepsilon/3 <= S_\tau <= I + \varepsilon/3$
⇨ $-2*\varepsilon/3 <= S_\tau - s_\tau <= 2*\varepsilon/3 < \varepsilon$
⇨ $S_\tau - s_\tau < \varepsilon$

<=|
Имаме, че за всяко ε > 0 - фиксираме, съществува δ = δ(ε) >0 и за всяко τ, $\delta_\tau < \delta \Rightarrow S_\tau - s_\tau < \varepsilon$
⇨ за всяко τ: $s_\tau <= \underline{I} <= I' <= S_\tau$

1. $I' <= S_\tau$
2. $-\underline{I} <= -s_\tau$

⇨ $0 <= I' - \underline{I} <= S_\tau - s_\tau$ (за всяко τ)

Ако допуснем, че $A = I' - \underline{I} <> 0 => A > 0$

Нека $\varepsilon = A/2 => A = I' - \underline{I} => S_\tau - s_\tau < A/2$ (за някое τ)

⇨ $\delta = \delta(A/2) > 0$ и за всяко τ, $\delta_{(\tau)} = \delta > 0 => S_\tau - s_\tau < A/2$ за всяко c = $\{c_i\}^n_{i=1}$, такова че ci Є $[x_{i-1}, x_i]$, за i = 1, 2, …, n

⇨ $s_\tau <= \sigma_\tau(f; c) <= S_\tau$

+

$-S_\tau <= -I <= -s_\tau$

⇨ $|\sigma_\tau(f; c) - I| <= S_\tau - s_\tau$

⇨ $|\sigma_\tau(f; c) - I| < \varepsilon$ // условие за интегруемост в Риманов смисъл.

Свойство 1: $\int_a^b f(x)dx = b - a$, при f(x) = 1

Свойство 2: Нека f(x) и g(x) са интегруеми върху интервала [a, b], тогава :

1. f(x) + g(x) - интегруема върху интервала [a, b]
   $\int_a^b [f(x) + g(x)]\,dx = \int_a^b f(x)dx + \int_a^b g(x)dx$, при f(x) = 1
2. d f(x), d Є R е интегруема върху интервала [a, b]
   $\int_a^b df(x)dx = d\int_a^b f(x)dx$

Свойство 3: Ако f(x) >= 0 и е интегруема върху интервала [a, b], тогава :
$\int_a^b f(x)dx >= 0$

Свойство 4: Ако f(x) <= g(x), то $\int_a^b f(x)dx <= \int_a^b g(x)dx$

Свойство 5: Нека f(x) >= 0 и е интегруема върху интервала [a, b], тогава :

1. |f(x)| - интегруема върху интервала [a, b]
2. $|\int_a^b f(x)dx| <= \int_a^b |f(x)|dx$

Свойство 6: Нека f(x) е интегруема върху интервала [a, b] и интервала [c, d] е произволен подинтервал на [a, b], за който
[c, d] < [a, b], тогава f(x) е интегруема върху интервала [c, d].

Свойство 7: Нека f(x) е интегруема върху интервала [a, b] и a < c < b, тогава:

$$\int_a^b f(x)dx = \int_a^c f(x)dx + \int_c^b f(x)dx$$

Свойство8: Ако f(x) е интегруема в интервала [a, b] и ако m и M са съответно една нейна долна и една нейна горна граница в този интервал, то $m(b - a) <= \int_a^b f(x)dx <= M(b - a)$

Th. Нека f(x) е непрекъсната върху интервала [a, b], тогава съществува поне една точка *c* Є [a, b], за която е изпълнено равенството :

$\int_a^b f(x)dx$ = f(c)(b – a)

Доказателство: Да означим с m и M съответно точната долна и точната горна граница на f(x) в интервала [a, b] , m<= f(x) <=M
От свойство 8 обаче =>

m(b - a) <= $\int_a^b f(x)dx$ <= M(b - a), или m < $\frac{\int_a^b f(x)dx}{b-a}$ <=M

От непрекъснатостта на f(x) ∃ две точки x_1 и x_2 от интервала [a, b], за които $f(x_1)=m$ и $f(x_2)=M$.
И така точките x_1 и x_2 определят един интервал.Но всяка непрекъсната функция приема всички стойности между максимума и минимума си

Следователно ∃ поне една c∈ [a, b] , така че f(c) = $\frac{\int_a^b f(x)dx}{b-a}$.

Теорема на Нютон-Лайбниц: Ако функцията f(x) е непрекъсната в един интервал, то функцията F(x) = $\int_a^x f(t)dt$ е деференцеруема в този интервал и за всяко x ∈ D е изпълнено равенството F’(x)=f(x).

Док-во: Нека x е произволна точка от интервала D.Ako x+h е друга точка от този интервал, то ще имаме

$$\frac{F(x+h)-F(x)}{h}=\frac{\left[\int_a^{x+h} f(t)dt-\int_a^x f(t)dt\right]}{h}=\frac{\left[\int_a^x f(t)dt+\int_x^{x+h} f(t)dt-\int_a^x f(t)dt\right]}{h}=$$

$$=\frac{\left[\int_x^{x+h} f(t)dt\right]}{h}=\frac{f(c)(x+h-x)}{h}=f(c), c\in[x, x+h]$$

Ако h $\to 0$ следва c $\to x$,

т.е F’(x) = $\lim_{h\to 0}$ $\frac{F(x+h)-F(x)}{h}=\lim_{h\to 0}$ f(c)= f(x).

Начин за пресмятане на определен интеграл чрез теоремата на Лайбниц – Нютон :

Нека f(x) е непрекъсната върху интервала [a, b] и Φ(x) е примитивна(Φ(x)'=f(x)) функция на f(x) върху интервала [a, b], тогава:

$\int_a^b f(x)dx$ = Φ(b) - Φ(a) = Φ(x) $|^b_a$

Док. F(x) = $\int_a^x f(t)dt$ - примитивна функция на f(x) върху интервала [a, b]

- ⇨ От основната теорема на интегралното смятане => съществува c – константа , такава че Φ(x) = F(x) + c, за всяко x Є [a, b]
  Φ(x) = $\int_a^x f(t)dt$ + c

1. x = a => Φ(a) = $\int_a^a f(t)dt$ + c = 0 + c = c – константа
   т.е. Φ(a) – константа
2. x = b => Φ(b) = $\int_a^b f(t)dt$ + Φ(a)

⇨ $\int_a^b f(t)dt$ = Φ(b) - Φ(a) (1)

Т.е. за да пресметнем определения интеграл трябва да пресметнем най-напред неопределения, т.е. да намерим една примитивна функция Φ(x) на f, след което да приложим формула (1).

Лекция 2

Хомогенни системи и подпространства на F^n. Теорема на Руше и решения на нехомогенна система

1. Пространството от решения на хомогенна система

Да разгледаме една линейна хомогенната система уравнения, с коефициенти от полето F:

$$(1)\ \begin{vmatrix} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = 0 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = 0 \\ \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\ a_{s1}x_1 + a_{s2}x_2 + \cdots + a_{sn}x_n = 0 \end{vmatrix} .$$

Нека $A = \begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \cdots\cdots & \cdots\cdots & \cdots & \cdots\cdots \\ a_{s1} & a_{s2} & \ldots & a_{sn} \end{pmatrix}$ е матрицата на системата и стълбовете на матрицата A са:

$$\mathbf{c}_1 = \begin{pmatrix} a_{11} \\ a_{21} \\ \ldots \\ a_{s1} \end{pmatrix},\ \mathbf{c}_2 = \begin{pmatrix} a_{12} \\ a_{22} \\ \ldots \\ a_{s2} \end{pmatrix},\ \ldots,\ \mathbf{c}_n = \begin{pmatrix} a_{1n} \\ a_{2n} \\ \ldots \\ a_{sn} \end{pmatrix}.$$

Твърдение 2.1. *Решението на всяка линейна хомогенна система с n неизвестни и коефициенти от полето F е линейно подпространство на n-мерното векторно пространството F^n.*

Доказателство. Нека да бележим с L множеството от решенията на системата. Множеството L винаги има поне един елемент и това е нулевото решение $\mathbf{0} = (0, 0, \ldots, 0)$. Векторът $\beta = (\beta_1, \ldots, \beta_n)$ е решението на системата (1), тогава и само тогава, когато за всеки индекс i е изпълнено $a_{i1}\beta_1 + a_{i2}\beta_2 + \cdots + a_{in}\beta_n = 0$. Като обединим тези равенства, ще получим следната зависимост за стълбовете на матрицата на системата:

$$\beta_1 \begin{pmatrix} a_{11} \\ a_{21} \\ \ldots \\ a_{s1} \end{pmatrix} + \beta_2 \begin{pmatrix} a_{12} \\ a_{22} \\ \ldots \\ a_{s2} \end{pmatrix} + \cdots + \beta_n \begin{pmatrix} a_{1n} \\ a_{2n} \\ \ldots \\ a_{sn} \end{pmatrix} = \beta_1 c_1 + \beta_2 c_2 + \cdots + \beta_n c_n = \mathbf{0}.$$

Нека $\beta = (\beta_1, \ldots, \beta_n)$ и $\gamma = (\gamma_1, \ldots, \gamma_n)$ са решения на системата (1), тогава за λ и μ - произволни елементи от полето F е изпълнено:

$$(\lambda.\beta_1 + \mu.\gamma_1)c_1 + \cdots + (\lambda.\beta_n + \mu.\gamma_n)c_n = \lambda(\beta_1 c_1 + \cdots + \beta_n c_n) + \mu(\gamma_1 c_1 + \cdots + \gamma_n c_n) = \mathbf{0},$$

следователно $\lambda.\beta + \mu.\gamma$ е също решение на системата (1). По този начин се получава, че множеството от решения е подпространство. □

Дефиниция 2.2. *Базис на пространството от решения на една хомогенна система линейни уравнения се нарича фундаментална система от решения на линейната система.*

Ясно е, че ако една хомогенна система има ненулево решение, тогава нейното подпространство от решения има базис и следователно за тази система съществува фундаментална система от решения.

ТЕОРЕМА 2.3. *Размерността на подпространството от решения на хомогенната система*

$$(1)\quad \begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = 0 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = 0 \\ \cdots\cdots\cdots\cdots\cdots\cdots\cdots \\ a_{s1}x_1 + a_{s2}x_2 + \cdots + a_{sn}x_n = 0 \end{cases}$$

е равна на $n - r$, където $r = r(A)$ е рангът на матрицата на системата

$$A = \begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \cdots\cdots & \cdots\cdots & \cdots & \cdots\cdots \\ a_{s1} & a_{s2} & \ldots & a_{sn} \end{pmatrix}.$$

Доказателство. Да бележим с L пространството от решенията на системата и нека рангът на матрицата A е $r = r(A)$. Следователно тя има r линейно независими реда и всеки друг ред е тяхна линейна комбинация. Това означава, че съществуват r линейно независими уравнения и можем да считаме, че това са първите r уравнения:

$$(2)\quad \begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = 0 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = 0 \\ \cdots\cdots\cdots\cdots\cdots\cdots\cdots \\ a_{r1}x_1 + a_{r2}x_2 + \cdots + a_{rn}x_n = 0 \end{cases}.$$

Системите (1) и (2) имат едни и същи рещения, т.е. те са еквивалентни.
Матрицата A има ранг r, колкото е и рангът на матрицата на системата (2). Нека индексите на r линейно независими стълба са $i_1, i_2, \ldots, i_r$, а останалите индекси на стълбове са $j_1, j_2, \ldots, j_{n-r}$. Изпълнено е $\{i_1, \ldots, i_r\} \cup \{j_1, \ldots, j_{n-r}\} = \{1, 2, \ldots, n\}$. Подматрицата

$$B = \begin{pmatrix} a_{1,i_1} & a_{1,i_2} & \ldots & a_{1,i_r} \\ a_{2,i_1} & a_{2,i_2} & \ldots & a_{2,i_r} \\ \cdots\cdots & \cdots\cdots & \cdots & \cdots\cdots \\ a_{r,i_1} & a_{r,i_2} & \ldots & a_{r,i_r} \end{pmatrix}$$

има линейно независими стълбове, следователно B е $r \times r$ матрица с ранг r. От Следствие 1.15 имаме, че само с елементарни преобразувания по редове матрицата B може да се преобразува до единична матрица. Ако приложим тези преобразувания към матрицата на системата (2), ще получим, че тази система се преобразува до следната система:

$$(3)\quad \begin{cases} x_{i_1} + a'_{1j_1}x_{j_1} + a'_{1j_2}x_{j_2} + \cdots + a'_{1j_{n-r}}x_{j_{n-r}} = 0 \\ x_{i_2} + a'_{2j_1}x_{j_1} + a'_{2j_2}x_{j_2} + \cdots + a'_{2j_{n-r}}x_{j_{n-r}} = 0 \\ \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\ x_{i_r} + a'_{rj_1}x_{j_1} + a'_{rj_2}x_{j_2} + \cdots + a'_{rj_{n-r}}x_{j_{n-r}} = 0 \end{cases}.$$

Построяваме решения на системата (2), като на неизвестните $x_{j_1}, x_{j_2}, \ldots, x_{j_{n-r}}$ задаваме стойности, които съответстват на векторите от стандартния базис на F^{n-r}. За $1 \leq s \leq n - r$ получаваме векторите $\beta^s = (\beta^s_1, \beta^s_2, \ldots, \beta^s_n)$, които имат следните координатите:

$$\beta^s_t = \begin{cases} 0 & \text{когато } t = j_p \text{ и } p \neq s, \\ 1 & \text{когато } t = j_s, \\ -a'_{k,j_s} & \text{когато } t = i_k. \end{cases}$$

Ясно е, че векторите β^s, за $s = 1, \ldots, n-r$, са линейно независими и са решения на системата, т.е. $\beta^s \in L$. Следователно линейната обвивка $l(\beta^1, \ldots, \beta^{n-r})$ се съдържа в пространството от решения L.
Нека е $\gamma = (\gamma_1, \gamma_2, \ldots, \gamma_n) \in L$ е едно произволно решение на системата. Разглеждаме вектора

$$\alpha = \gamma - \gamma_{j1}\beta^1 - \gamma_{j2}\beta^2 - \cdots - \gamma_{j_{n-r}}\beta^{n-r} \in F^n.$$

Векторът α е решение на системата, защото е линейна комбинация на решения. Координатата $\alpha_{j_s} = \gamma_{j_s} - \gamma_{j_s}.\beta^s_{j_s} = 0$ за $s = 1, \ldots, n-r$. Векторът α е решение на системата и следователно е решение и на всяко уравнение от системата (3). По този начин получаваме, че $\alpha_{i_t} = 0$, за $t = 1, \ldots, r$, следователно всички координати на вектора α са 0. По този начин получаваме:

$$\alpha = \mathbf{0} \quad \Rightarrow \quad \gamma = \gamma_{j1}\beta^1 + \gamma_{j2}\beta^2 + \cdots + \gamma_{j_{n-r}}\beta^{n-r} \in l(\beta^1, \ldots, \beta^{n-r}).$$

Това означава, че пространството L се съдържа в линейната обвивка, следователно решението е $L = l(\beta^1, \ldots, \beta^{n-r})$. От това получаваме, че

$$\dim L = \dim l(\beta^1, \ldots, \beta^{n-r}) = n - r. \qquad \square$$

2. Подпространства на F^n и хомогенни системи уравнения

ТЕОРЕМА 2.4. *Всяко подпространство на n-мерното векторно пространство F^n е решение на хомогенна система линейни уравнения.*

Доказателство. Нека U е произволно подпространство на F^n.
Ако U е нулевото пространство, тогава то е решение на следната система:

$$U : \left| \begin{array}{l} x_1 = 0 \\ x_2 = 0 \\ \ldots\ldots \\ x_n = 0 \end{array} \right. .$$

Ако U не е нулевото пространство и нека $\dim U = k$ и $a_1, a_2, \ldots, a_k$ е базис на пространството U. Нека

$$\begin{array}{l} a_1 = (a_{11}, a_{12}, \ldots, a_{1n}) \\ a_2 = (a_{21}, a_{22}, \ldots, a_{2n}) \\ \ldots\ldots\ldots\ldots\ldots\ldots \\ a_k = (a_{k1}, a_{k2}, \ldots, a_{kn}) \end{array} .$$

Разглеждаме следната хомогенна система уравнения:

$$(4) \quad \left| \begin{array}{l} a_{11}y_1 + a_{12}y_2 + \cdots + a_{1n}y_n = 0 \\ a_{21}y_1 + a_{22}y_2 + \cdots + a_{2n}y_n = 0 \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ a_{k1}y_1 + a_{k2}y_2 + \cdots + a_{kn}y_n = 0 \end{array} \right. .$$

Матрицата на тази система се състои от k линейто независими реда и има ранг k. Следователно пространството W от решения на системата (4) има размерност $t = n - k$. Нека $\beta_1 = (\beta_{11}, \beta_{12}, \ldots, \beta_{1n}), \ldots, \beta_t = (\beta_{t1}, \beta_{t2}, \ldots, \beta_{tn})$ са базис на пространството W от решения на системата (4). Тогава разглеждаме системата:

$$(5) \quad \left| \begin{array}{l} \beta_{11}x_1 + \beta_{12}x_2 + \cdots + \beta_{1n}x_n = 0 \\ \beta_{21}x_1 + \beta_{22}x_2 + \cdots + \beta_{2n}x_n = 0 \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ \beta_{t1}x_1 + \beta_{t2}x_2 + \cdots + \beta_{tn}x_n = 0 \end{array} \right. .$$

Рангът на матрицата на тази система е t, следователно размерността на решението на (5) е равна $k = n - t$.

За произволно $d \in \{1, \ldots, t\}$ е изпълнено, че векторът β_d е решение на системата (4), следователно за всяко $p \in \{1, \ldots, k\}$ имаме:

$$\beta_{d1}a_{p1} + \beta_{d2}a_{p2} + \cdots + \beta_{dn}a_{pn} = 0.$$

Това равенство задава, че векторът a_p е решение на d-тото уравнение от системата (5). Следователно всеки вектор от базиса $a_1, \ldots, a_k$ на пространството U е решение на системата (5). Размерността на решението на системата (5) е k, следователно решението съвпада с пространството U. □

3. Теорема на Руше и решения на нехомогенни системи

Нека разгледаме една линейна система с коефициенти от поле F.

$$(6) \left| \begin{array}{l} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ a_{s1}x_1 + a_{s2}x_2 + \cdots + a_{sn}x_n = b_s \end{array} \right. .$$

Матрицата $A = \begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \ldots\ldots & \ldots\ldots & \ldots & \ldots\ldots \\ a_{s1} & a_{s2} & \ldots & a_{sn} \end{pmatrix}$ се нарича матрица на системата, а разширена матрица на системата е $\bar{A} = \left(\begin{array}{cccc|c} a_{11} & a_{12} & \ldots & a_{1n} & b_1 \\ a_{21} & a_{22} & \ldots & a_{2h} & \vdots \\ \ldots\ldots & \ldots\ldots & \ldots & \ldots\ldots & \\ a_{s1} & a_{s2} & \ldots & a_{sn} & b_s \end{array} \right)$.

ТЕОРЕМА 2.5. **Теорема на Руше**
Една система линейни уравнения има решение тогава и само тогава, когато рангът на матрицата на системата съвпада с ранга на разширената матрица, т.е. $r(A) = r(\bar{A})$.

Доказателство. Нека стълбовете на матрицата на система(6) и стълба от свободните членове са съответно:

$$\mathbf{c}_1 = \begin{pmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{s1} \end{pmatrix}, \ \mathbf{c}_2 = \begin{pmatrix} a_{12} \\ a_{22} \\ \vdots \\ a_{s2} \end{pmatrix}, \ldots, \ \mathbf{c}_n = \begin{pmatrix} a_{1n} \\ a_{2n} \\ \vdots \\ a_{sn} \end{pmatrix}, \ \mathbf{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_s \end{pmatrix}.$$

Ако системата (6) има решение $\beta = (\beta_1, \beta_2, \ldots, \beta_n)$, тогава е изпълнено:

$$\beta_1 c_1 + \beta_2 c_2 + \cdots + \beta_n c_n = b,$$

следователно стълбът b е линейна комбинация на стълбовете $c_1, c_2, \ldots, c_n$. По този начин получаваме, че $l(c_1, c_2, \ldots, c_n) = l(c_1, c_2, \ldots, c_n, b)$ и $r(c_1, c_2, \ldots, c_n) = r(c_1, c_2, \ldots, c_n, b)$, което означава, че $r(A) = r(\bar{A})$.
Ако е изпълнено, че $r(A) = r(\bar{A})$, тогава $r(c_1, c_2, \ldots, c_n) = r(c_1, c_2, \ldots, c_n, b)$, откъдето следва, че $l(c_1, c_2, \ldots, c_n) = l(c_1, c_2, \ldots, c_n, b)$. По този начин се получава, че $b \in l(c_1, c_2, \ldots, c_n)$ и съществуват коефициенти $\beta_1, \beta_2, \ldots, \beta_n$ от полето F, за които е изпълнено

$$\beta_1 c_1 + \beta_2 c_2 + \cdots + \beta_n c_n = b.$$

Следователно $(\beta_1, \beta_2, \ldots, \beta_n)$ е решение на системата. □

ТВЪРДЕНИЕ 2.6. *Нека (6) е една линейна система, а (7) съответната ѝ хомогенна система:*

$$(6) \left| \begin{array}{l} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ a_{s1}x_1 + a_{s2}x_2 + \cdots + a_{sn}x_n = b_s \end{array} \right. \qquad (7) \left| \begin{array}{l} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = 0 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = 0 \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ a_{s1}x_1 + a_{s2}x_2 + \cdots + a_{sn}x_n = 0 \end{array} \right.$$

Тогава са изпълнени следните свойства:

а) Ако $\alpha = (\alpha_1, \ldots, \alpha_n)$ *и* $\beta = (\beta_1, \ldots, \beta_n)$ *са решения на нехомогенната система (6), тогава тяхната разлика*

$$\alpha - \beta = (\alpha_1 - \beta_1, \ldots, \alpha_n - \beta_n)$$

е решение на хомогенната система (7);

б) Ако $\alpha = (\alpha_1, \ldots, \alpha_n)$ *е решение на система (6) и* $\gamma = (\gamma_1, \ldots, \gamma_n)$ *е решение на хомогенната система (7), тогава тяхната сума*

$$\alpha + \gamma = (\alpha_1 + \gamma_1, \ldots, \alpha_n + \gamma_n)$$

е решение на системата (6);

в) Ако системата (6) е съвместима и $\alpha = (\alpha_1, \ldots, \alpha_n)$ *е едно нейно решение, тогава всяко решение на тази система е от вида*

$$\alpha + \gamma = (\alpha_1 + \gamma_1, \ldots, \alpha_n + \gamma_n),$$

където $\gamma = (\gamma_1, \ldots, \gamma_n)$ *е произволно решение на хомогенната система (7).*

Доказателство. а) Да заместим решенията $\alpha = (\alpha_1, \ldots, \alpha_n)$ и $\beta = (\beta_1, \ldots, \beta_n)$ на нехомогенната система (6) в уравнение с номер i и да извадим двете равенства:

$$\begin{array}{l} a_{i1}\alpha_1 + a_{i2}\alpha_2 + \cdots + a_{1n}\alpha_n = b_i \\ a_{i1}\beta_1 + a_{i2}\beta_2 + \cdots + a_{in}\beta_n = b_i \end{array} \Rightarrow\ a_{i1}(\alpha_1 - \beta_1) + \cdots + a_{1n}(\alpha_n - \beta_n) = 0.$$

По този начин получаваме, че разликата $\alpha - \beta$ е решение на хомогенната система (6).

б) Твърдението се получава, когато заместим с решението $\alpha = (\alpha_1, \ldots, \alpha_n)$ в i-тото уравнение на система(6) и с решението $\gamma = (\gamma_1, \ldots, \gamma_n)$ в i-тото уравнение на система (7) и съберем двете равенства:

$$\begin{array}{l} a_{i1}\alpha_1 + a_{i2}\alpha_2 + \cdots + a_{1n}\alpha_n = b_i \\ a_{i1}\gamma_1 + a_{i2}\gamma_2 + \cdots + a_{in}\gamma_n = 0 \end{array} \Rightarrow\ a_{i1}(\alpha_1 + \gamma_1) + \cdots + a_{1n}(\alpha_n + \gamma_n) = b_i.$$

в) Нека $\alpha = (\alpha_1, \ldots, \alpha_n)$ е едно фиксирано решение на система (6) и $\beta = (\beta_1, \ldots, \beta_n)$ е друго решение на тази система. От т.а), получихме че $\gamma = \alpha - \beta$ е решение на система (7), следователно $\beta = \alpha + \gamma$. □

Следствие 2.7. *Една линейна система с* n *неизвестни има единствено решение точно когато* $n = r(A) = r(\bar{A})$.

Доказателство За да има решение системата трябва да изпълнява $r(A) = r(\bar{A})$. Но когато $n = r(A)$ решението на съответната хомогенна система има размерност $0 = n - r$ и следователно хомогенната система има единствено нулевото решение, откъдето получаваме, че и решението на изходната система също е единствено. □

Непосредствено от Теорема 2.3 и Твърдение 2.6 получаваме:

Следствие 2.8. *Ако една линейна система с* n *неизвестни е съвместима, тогава решението ѝ може да се изрази с* $n - r(A)$ *параметъра.*