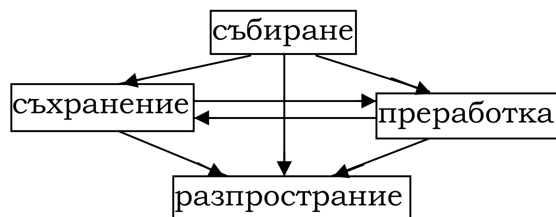
Увод в Програмирането (УП)
за специалности
Приложна Математика и Статистика,
I курс

1. Основни информационни дейности. Средства за автоматизация на смятането.

Основни информационни дейности – събиране, съхранение, преработване и разпространение на информация.



Средства за автоматизация на смятането – нуждата от тях е възникнала много отдавна:

1. Първо помагало на човека за изчисления са пръстите на ръцете. С появата на писмеността, за изчисленията се използват папируси/пергаменти и средства за писане при пресмятанията. След това се появява сметалото (абак), използвано и днес;
2. Сметачни машини – извършват автоматично отделни операции – Блез Паскал (около 1640 г.);

3. Автоматични сметачни машини – Чарлз Бабедж (около 1840 г.). Съставени от няколко блока: управляващо устройство, аритметично устройство, памет (където се съхраняват междинните резултати), вход и изход (външен носител). Моделът се усъвършенства от Джон фон Нойман, който предлага и самата програма да се намира в паметта – автоматизация на целия изчислителен процес.

Два вида автоматични сметачни машини – **аналогови** и **цифрови.** Разликата е в начина на представяне на числата. В аналоговите числата се представят чрез физически величини (напрежение, дължина, налягане), а в цифровите числата се представят в цифров вид. Цифрата се изразява чрез състоянието на физически елемент, който може да се намира в краен брой различни състояния. Цифри – 0 или 1 (двоични числа) ⇔ има/няма напрежение, южен/северен полюс на магнит. Правени са опити с три състояния.

Компютър – цифрова автоматична сметачна машина. Съвременните компютри са фон Нойманови компютри. Развитието на компютрите засяга техническата реализация, а не принципите.
Нулево поколение – електромеханични компютри, основен елемент – реле (Джон Атанасов – признат от Върховния съд на САЩ през 1974 г. за баща на цифровите електронно-изчислителни машини).
Първо поколение – основен елемент – електронна лампа.
Второ поколение – основен елемент – транзистори.
Трето поколение – основен елемент – интегрални схеми.
Четвърто поколение – основен елемент – големи интегрални схеми, по-компактни по размери (минитюаризация).
Пето поколение – търсят се други принципи, различни отноймановите. Основен елемент – свръхголеми интегрални схеми.

2. Алгоритми. Определение и примери. Свойства. Начини на изразяване – словесно описание, блок-схема, алгоритмичен език.

Компютър (друга дефиниция) – техническо средство за автоматично изпълнение на алгоритми.

Алгоритъм – списък от краен брой правила за извършване на дадени действия в определен ред, чрез който се решават задачи от даден тип.

Пример: Дадени са числата $a_1,a_2,...,a_k,…,a_n$. Да се намерят най-малкото (α) и най-голямото (β).

Словесно описание (номер на стъпката, действие и указване на наследника):

1. Въвеждане на n и $a_1,a_2,...,a_n$. Премини към 2.
2. $\alpha = a_1$. Премини към 3.
3. $\beta = a_1$. Премини към 4.
4. k = 2. Премини към 5.

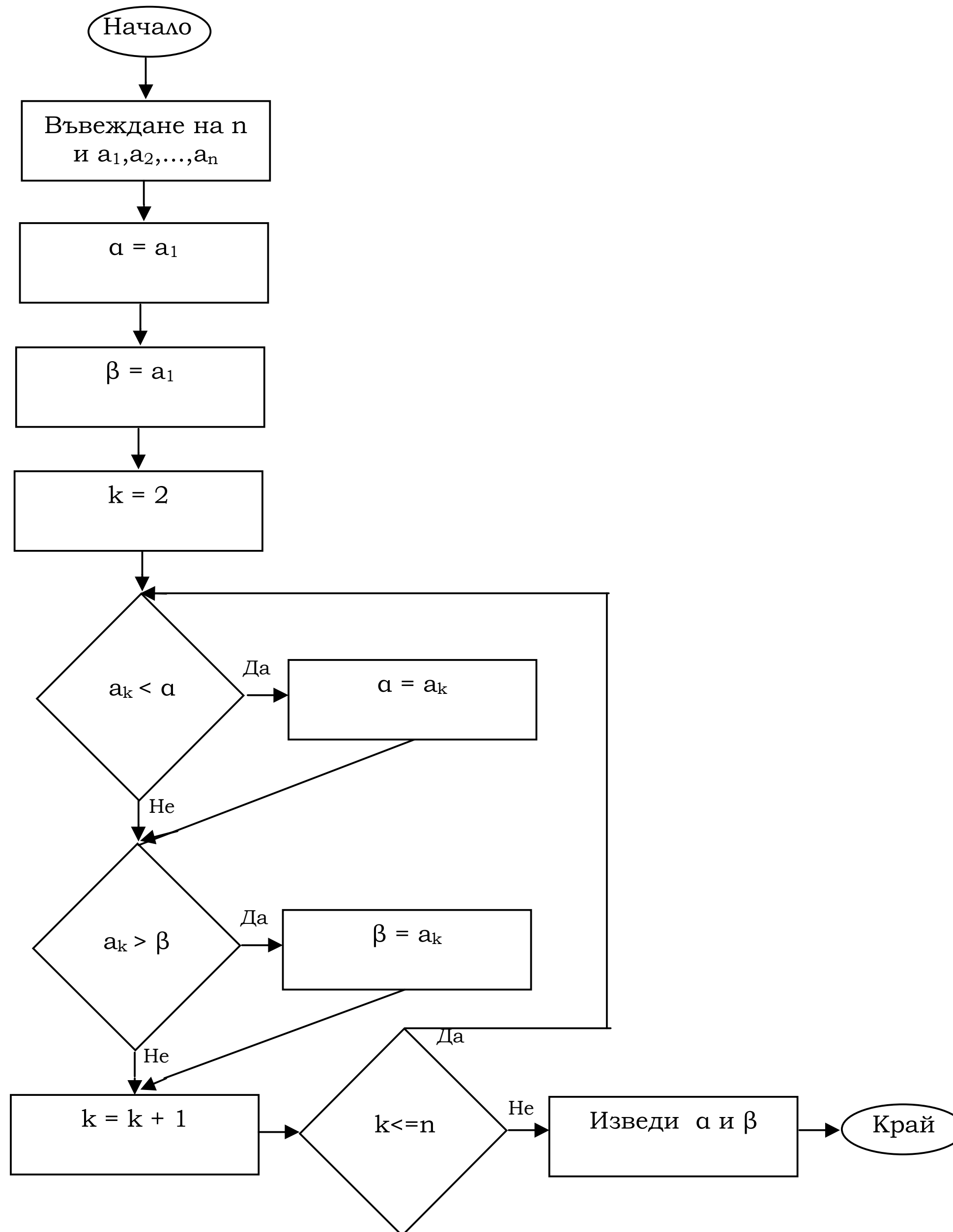
5. Ако $a_k < \alpha$ премини към 6, иначе премини към 7.
6. $\alpha = a_k$. Премини към 7.
7. Ако $a_k > \beta$ премини към 8, иначе премини към 9.
8. $\beta = a_k$. Премини към 9.
9. $k = k + 1$. Премини към 10.
10. Ако $k<=n$ премини към 5, иначе премини към 11.
11. Изведи α и β. Премини към 12.
12. Прекрати изпълнението на алгоритъма.

Безусловни и условни действия – от условните може да се премине към повече от една стъпка. От безусловните само към една стъпка.
Стъпка – изпълнение на едно правило. Стъпките са съществено повече от правилата (стъпките зависят и от входните данни).

Основни свойства на алгоритмите:

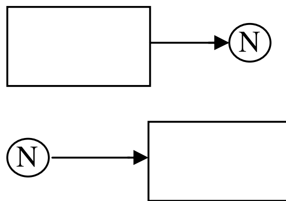
- **определеност** - описанието на алгоритъма е ясно и недвусмислено, определя еднозначно действията, които се извършват;
- **масовост** – прилага се не за една конкретна задача, а за клас от задачи. Конкретната задача се определя с набора от входните данни;
- **резултатност** – всяко изпълнение на алгоритъма завършва за крайно време. Трябва да са налице предпоставки – да има краен брой стъпки и всяка стъпка да се изпълнява за крайно време;
- **цикличност** – има възможност при описване на алгоритмите групи от правила да се изпълняват многократно.

Блок – схеми за описание на алгоритмите



С правоъгълните блокове се означават безусловните действия. С ромбовидните се означават условните (от тях има две разклонения - истина или лъжа). Елипсовидните блокове указват началото и края на програмата. Възможно е да се използват блокове с форма на успоредници за въвеждането и извеждането на данни. Във всеки блок влиза и излиза поне една стъпка с изключение на началото и края.

Възможно е да се прави прекъсване в блок–схемата по следния начин:



N е цяло число.

Компютърът изисква **алгоритмите** да са написани на **машинен език.** Транслаторът е програма на машинен език. Функцията му е да преобразува алгоритъма от словесно описание или блок – схема в машинен език. Естествено транслаторите за словесно описание и за блок – схема са различни.
Алгоритмичен език – начин на изразяване на алгоритми чрез набор от команди с точно определен синтаксис, който е по-близък до машинния език и улеснява значително работата на транслатора.

```
#include <iostream.h>  // включва се за да може да се използва стандартна библиотека за вход/изход
void main ()  //  функцията не връща резултат, затова е void
{ int i, n, k;
   float alfa, beta, a[100];
   cin >> n;
   for (i = 0; i < n; i++)
    cin >> a[i];
   alfa = a[0];
   beta = a[0];
   for (k = 1; k < n; k++)
    if (a[k] < alfa) alfa = a[k];
    else if (a[k] > beta) beta = a[k];
  cout << alfa << " " << beta;
}
```

Процедурно програмиране

Съществуват два основни стила за програмиране**: процедурен** и **декларативен**.
При процедурния стил програмата се реализира по схемата:
Програма = Алгоритъм + Структури от данни
Съществуват стотици процедурни езици. Такива са асемблерните, Фортран, Алгол, Кобол, Снобол, Паскал, Ада, С, С++ и др.
Процедурният стил е тясно свързан с фонНоймановата архитектура на компютрите и наследява всички техни недостатъци.
Линията за връзка между ОП и ЦП е тясното място наНоймановите компютри и се нарича “гърло на бутилката”.
Изпълнението на машинната програма води до съществено изменение на ОП. Това се достига чрез многократно прехвърляне на стойността на отделни клетки в едната и другата посока.

При това обаче съществена част от трафика през това място не са "полезни" данни, а адреси на данни, а също операции, които се използват за пресмятане на такива адреси. Програмата, написана на процедурен език, действа над клетките на паметта, отделени за променливите. Програмистът трябва да опише всички последователни изменения на тази памет, така че като се изпълнят тези изменения на Неймановия компютър, да се получи търсеният резултат. Така програмата трябва да управлява последователните стъпки на работа на компютъра. От необходимостта, обработката на данните в стил "клетка след клетка" да се опише точно и детайлно, следват следните недостатъци: ***относително малка мощност, липса на гъвкавост, ниска производителност***.

Все пак връзката на процедурните езици с архитектурата на фон Нойман има и предимства. Главното предимство е ефективната им реализация.

История на C и C++

Езикът C++ се е развил от C, който на свой ред е създаден на базата на два предшестващи езика - BCPL и B.

Езикът BCPL е създаден през 1967 година от Мартин Ричард като език за писане на компилатори и операционни системи. Кен Томпсон предвижда много възможности в своя язик B – дубликат на BCPL и използва B за създаването на ранните версии на операционната система UNIX в Bell Laboratories в 1970 г. на компютъра DEC PDP-7.

Езикът C е развит от езика B от Денис Ричи в Bell Laboratories и първоначално е реализиран на компютъра DEC PDP-11 в 1972 г. C използва много от концепциите на BCPL и B, а също добави типовете данни и други свойства. Първоначално C придоби широка известност като език за разработка на операционната система UNIX. Сега фактически всички нови операционни системи са написани на C или на C++. В последните две-три десетилетия C стана достъпен за болшинството компютри. C е независим от апаратните средства.

При щателна работа на С може да се напишат ***мобилни*** програми, ***преносими*** на болшинството компютери.
В края на 70-те години С се разви в това, което сега се разбира като ”традиционно С”, “класическо С” или “С на Керниган и Ричи”. Публикуването на книгата на Керниган и Ричи “Език за програмиране С” (известна като “Синята книга”) от издателството Prentice-Hall привлече широко вниманието към този език. Широкото разпространение на С на различни типове компютри (понякога наричани ***апаратни платформи****)* доведе за съжаление към много вариации на езика. Те си приличаха, но бяха несъвместими една с друга. Стана ясно, че е необходима стандартна версия на С. В 1983 г. при Американския Национален Институт за Стандарти беше създаден технически комитет за да “осигури недвусмислено и машинно-независимо определение на езика”. В 1989 г. стандартът е утвърден. Второто издание на книгата на Керниган и Ричи, излязло през 1988 г., отразява тази версия, наричана ANSI C, която се използва сега повсеместно.

C++ е разширение на C и е разработен от Бьорн Страуструп в началото на 80-те години в Bell Laboratories. C++ осигурява ред свойства, които подобряват езика C, но по-важното е, че той осигурява възможността за ***обектно-ориентирано програмиране****.* Това е революционна идея в света на софтуера. ***Обектите*** са ефективни, повторно използваеми ***компоненти*** на софтуера, моделиращи елементите на реалния свят. Използването на обектно-ориентираното програмиране може значително да повиши производителността на труда. Обектно-ориентираните програми се разбират, коригират и модифицират по-лесно.

C++ е хибриден език, който дава възможност да се програмира и в стил C, и в обектно-ориентиран стил, и в двата стила едновременно. C++ е доминиращ език за системно програмиране.

3. Обща структура на компютрите. Процесори. Видове вътрешна памет. Програма на машинен език.

Най-общо компютърът е съставен от две части: **апаратна част** (**хардуер**) и **програмна част** (**софтуер**).

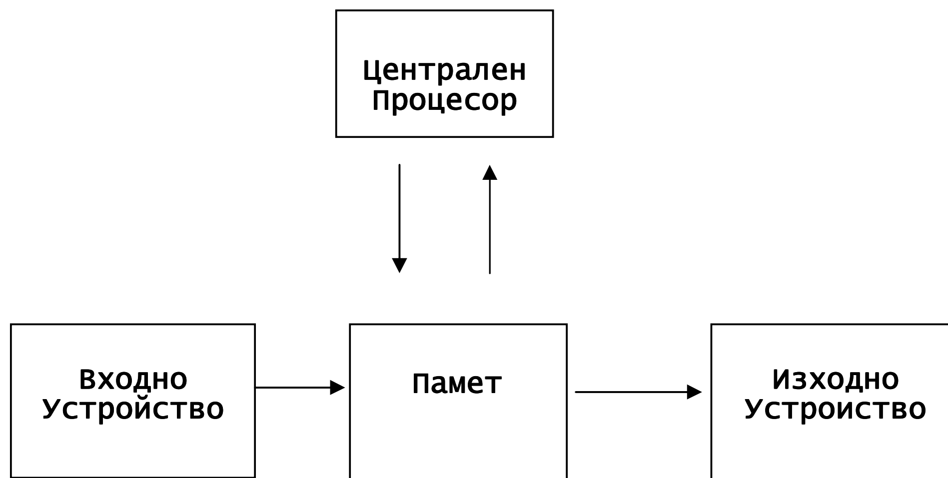
Основни компоненти на софтуера:

- **операционни системи** – реализират връзката с потребителя, поддържат различни режими на работа на централния процесор, управляват работата на компютъра в мрежа - MS DOS, Windows, Unix
- **среди за програмиране** – предоставят средства за съставяне, проверка и изпълнение на програми – Borland C++, Visual C++
- **приложни системи** – изпълними приложения

Основни компоненти на хардуера:

- **централен процесор** – съставен от управляващо устройство и аритметично устройство

- **вътрешна памет (оперативна)** – постоянна памет (ROM) и памет с произволен достъп (RAM)
- **периферни устройства** – за въвеждане на информация, за извеждане на информация и външна памет (за въвеждане и извеждане на информация)
- **шина**



Архитектура на компютър.

Начин на изпълнение на програма:

Програма → периферно устройство за въвеждане. Въвеждат се програмата и данните, които тя трябва да обработва от външен носител → оперативна памет → последователно по една команда се вписва в регистър на процесора → управляващото устройство дешифрира командата и изпраща съответните инструкции към аритметичното устройство → извеждане на външен носител или записване на външна памет (тъй като съдържанието на оперативната памет се губи при спиране на компютъра). Съществуват връзки между отделните компоненти за пренасяне на информация. Самата среда за пренасяне на информация – това е **шината** (многожилест проводник).

Разлика между постоянна памет (ROM) и памет с произволен достъп (RAM):

ROM (Read-only memory) – памет, от която може само да се чете. В нея са записани програми, които се изпълняват, когато се стартира компютъра. Тези програми най-често се използват за тестване на компонентите на хардуера преди управлението им да се предаде

на операционната система. Тези програми често се наричат BIOS (Basic Input Output System).
RAM (Random Access Memory) – памет с произволен достъп (четене/запис). Съхранява информация само когато компютърът е включен.
На най-ниско логическо ниво паметта представлява последователност от двоични разряди (0 или 1), наречени **битове**.
Най-малката единица в паметта с номер (адрес) е **байта** – последователност от 8 бита. Адресацията започва от 0.
Важна характеристика на паметта е нейния **обем**:
КБ = 1024 байта
МБ = 1024 Кбайта
ГБ = 1024 Мбайта
ТБ = 1024 Гбайта

Важни характеристики на централния процесор.
Тактова честота – колко такта прави за 1 sec. Под един **такт** се разбира една елементарна операция (измерва се в херца – Hz).

От колко **бита** се състоят операндите - 8, 16, 32 бита.

Начини за увеличаване скоростта на процесора:

- **Cache памет** – междинна памет между централния процесор и оперативната памет. Когато процесорът избързва спрямо скоростта на приемане и предаване на данни от оперативната памет, той използва Cache паметта като буфер между него и оперативната памет. Cache паметта е близка по скорост до скоростта, с която процесорът оперира с регистрите си.
- **Конвейерни принципи** – конвейери в централния процесор, които едновременно да изпълняват команди.
- **Матричен процесор** – служи за по-специфични данни. Съдържа едно управляващо устройство и много аритметични устройства.
- **паралелизъм при изпълнение на командите.**

Шина – характеристики – **тактова честота**, **битове.** За един такт по шината се пренасят толкова бита, колкото е самата шина. Най-

добре е битовите характеристики на шината и на процесора да съвпадат.
Програма на машинен език – списък от краен брой команди. Една команда е едно правило от алгоритъма. Тя е съставена от **код на операцията** и **адресна част**. Адресната част може да бъде едноадресна, двуадресна, триадресна, четириадресна в зависимост от това колко адреса има записани в нея. Обикновено, компютрите изпълняват команди с един до три адреса. Адресна част в редки случаи може да липсва. Има компютри, които изпълняват само едноадресни команди, други изпълняват само двуадресни.
Пример +A1A2A3; операцията е “+”, адресът на първия операнд е A1, адресът на втория е A2, а резултатът трябва да се запише в A3.

Видове адреси:

- **непосредствен** – операндът е записан направо в командата

- **абсолютен** – съдържа препратка към регистър от централния процесор или поле от оперативната памет
- **косвен** – съдържа адрес на регистър от процесора, в който се съдържа самият адрес. По този начин се използват толкова бита, колкото е процесора, което обикновено е значително повече от битовете за един адрес
- **относителен** – съдържа две полета: адрес на регистър и отместване A = (R) + D. По този начин ако адресът на регистъра е 4 бита (има 16 регистри), а отместването е 12, то при положение, че регистърът е 32 бита, получаме 16 битов адрес, който всъщност може да използва 32 бита;

Предимства на относителния адрес:

- регистърът може да е базов. Той се ползва, когато програмата се модифицира в зависимост от това къде е разположена в оперативната памет. Всички команди, които зависят от това къде е разположена програмата, използват базов регистър, в който е записан адреса на началото на програмата

- индекс-регистър – когато се програмират цикли, при всяко изпълнение се менят адресите, които се използват. Затова адресирането се извършва спрямо този индекс-регистър, който се коригира при всяка итерация на цикъла
- адресирането може да се направи и без базов регистър, ако отместването препраща към адреса на следващата команда

Видове команди:

- **аритметични** – събиране, изваждане, умножение, деление
- **логически** – логическо събиране, умножение, отрицание, “изключващо или”
- **управляващи** – безусловен преход, условен преход, предаване на управлението към подпрограма, предаване на управление с възврат, команди за обмен на данни (между регистри и оперативната памет)
- **за работа с периферни устройства**

4. Периферни устройства и външна памет. Начин на представяне на числата.

Периферните устройства са:

- **за въвеждане** – от външен носител се въвежда в ОП, най-често мишка, клавиатура и др.
- **за извеждане** – от ОП се записва на външен носител, най-често монитор и принтер
- **за въвеждане и извеждане** – външна памет

Съществуват т.н. **компютърни ферми**, които представляват множество компютри свързани в мрежа. Един компютър е управляващ и потребителят управлява фермата от него. В този случай обикновено само той е снабден с периферни устройства.

Монитори – характеристики – дължина на диагонала на екрана (в инчове), брой на пикселите върху екрана. Мониторът се управлява от видео контролер, който има памет. Определен брой пъти в секунда записаното в паметта се изобразява на екрана чрез лъч,

който минава по пикселите ред по ред. За да не трепти мониторът, честотата на опресняване (Refresh Rate) трябва да е поне 72 Hz.
Принтери – използват се за извеждане. Най-разпространените са три вида:

- **матрични** – иглички се забиват върху лента (матрица) и по този начин се отпечатват върху листа. Предимството им е, че са евтини и че могат да се използват с индиго за правене на копия
- **мастилено-струйни** – главата впръсква мастило в листа. Евтини са, но поддръжката им е скъпа
- **лазерни** – работи чрез барабан, в който има светлинно изображение на това, което трябва да се печата. Към него е прикачен тонер, който оцветява листа. Той се управлява посредством лазер, който разчита изображението

HDD – hard disc drive (твърд диск) – характеристики – бързина и обем. Съставен е от няколко плочи една върху друга, заградени от две повърхнини, които се намагнитват. Всяка повърхнина се състои от концентрични окръжности (писти), всяка писта се състои от сектори. Пистите с един и същи радиус върху плочите

образуват цилиндър. Освен това твърдият диск има глави за четене и записване.
FDD – floppy disc drive (флопи-диск, дискета) – преносими дискети. Колкото и да е учудващо, все още се използват в НОИ и НАП.
CD – compact disc – обемът им е от порядъка на 700MB. Вече са основна средство за дистрибуция на софтуер. **CD ROM** – след еднократно записване върху тях, могат само да бъдат прочетени.
DVD – дискове с капацитет до над 25 пъти по-голям от обикновените дискове. Обикновено е 4,7 ГБ.
Магнитна летна – за голям обем важна информация, която е копирана за по-голяма сигурност.
Скенер – периферно устройство за въвеждане на снимки, фотографии, картини и текст.
Видеокамери – за въвеждане на филми и клипове.
За звук – въвеждане чрез микрофон, извеждане чрез тонколони.
Плотер – за извеждане (отпечатване) на чертежи, скици.
И още много нови устройства, които непрекъснато се появяват на пазара.

Начини на представяне на числата

В компютъра числата се представят в двоична бройна система.

Цели числа

	Цяло положително								Цяло отрицателно							
Прав код	0	0	0	0	0	1	0	1	1	0	0	0	0	1	0	1
Обратен код	0	0	0	0	0	1	0	1	1	1	1	1	1	0	1	0
Допълнителен код	0	0	0	0	0	1	0	1	1	1	1	1	1	0	1	1

Таблицата показва представянето на числото 5 (-5) в един байт. Първият бит е знаков - знакът е “-“ ако знаковият бит е 1. Обратният код за отрицателните цели числа се получава от правия код, като се обърнат всички битове без знаковия. Допълнителният код за цели отрицателни числа се получава като към обратният код прибавим 1 (аритметично). В C++ за представянето на числа се използва допълнителният код.

Числа с плаваща запетая се представят като $m \,.\, 2^p$, където m е мантисата (числовият запис на числото), а p (цяло число) е порядък на числото. Допуска се m да е със знак.

Ако имаме 4 байта, едно число с плаваща запетая може да се представи чрез 1 бит (знак на мантисата) + 8 бита (порядъка) + 23 бита (мантисата). За да можем да получим и отрицателен порядък числата се изчисляват като $m \,.\, 2^{p-const}$. Това се прави с цел множеството от допустимите стойности на порядъка да е симетрично относно нулата. Обикновено мантисата е нормализирана (нейната цяла част е съставена от една цифра и е възможно тя да не се пази, ако цифрата е фиксирана, например 1), т.е. тя е от вида y.xxxxxx, където x са някакви битове (0 или 1), а y е цифра (представена също с битове).

5. Константи. Видове. Вътрешно представяне.

Програма, която намира лицето и периметъра на квадрат със страна a.

```
#include <iostream.h>
void main ()
{ int a, p, s;
  cout << " Въведи a = ";
  cin >> a;
  s = a*a;
  p = 4*a;
  cout << " S = " << s << endl;
  cout << " P = " << p << endl;
}
```

Допустимите символи (азбуката на езика), които могат да се използват в една програма се делят на четири групи:

- **букви** – главни и малки латински букви + символ за подчертаване;
- **цифри** – арабските цифри от 0 до 9;
- **празни символи** – символи, които нямат явен графичен вид – интервал, символ за табулация, символ за нов ред и др.;
- **специални символи** – всички останали . , ; + - * / { } ( ) [ ] .

Някои операции са комбинация от символи (например >=, <=, ==, !=).

Адресът на младшия байт е адресът на полето от байтове. Старши/младши байт – зависи от начина на номерирането на байтовете (подреждането отляво надясно или обратно).
Идентификатор – последователност от букви и цифри, която започва с буква. Дължината не е фиксирана, но в компилатора може да се зададе брой на символите (например 32 символа) и компилаторът счита за еднакви идентификаторите, за които позицията на първият символ, по-който се различават, е по-голяма от това число.

Резервирани (ключови) думи – резервирани идентификатори, които изпълняват вградени функции и потребителят не може да ги използва за имена на свои променливи или функции. Служебните идентификатори започват с "_" и затова не се препоръчва потребителят да ползва този символ за начало на свой идентификатор.

В низ от символи може да се използват всички символи, поддържани от конфигурацията на компютъра.

Ключовите думи в С са **задължително** с **малки букви**. Малките и главните букви в идентификаторите играят различна роля.

Синтаксис – правила за изграждане на програмата като текст (низ от символи).
Семантика – правила, по които се изпълнява дадена програма или инструкция.

Числовите константи в С могат да бъдат:

- **цели** – десетични – те не трябва да започват с нула (25, -313 и т.н.). Осмични – започват с 0 (0713, 05264 и т.н.). Шестнадесетични – започват с 0x или 0X (0xAbC, 0xA231bB, и т.н.), няма значение дали шестнадесетичните константи са с малки или големи букви;
- **реални** – представят се с десетична запетая (тя е задължителна дори числото да е цяло). Два начина за представяне – като десетично число (34.5, -33.1, 5. , .3 и т.н.). С мантиса и порядък (24e-5 = $24.\ 10^{-5}$, 32E13 = $32.\ 10^{13}$ и т.н.).

Целите числови константи могат да бъдат от следните типове (Visual C++ 6.0) :

- short int – целочислен тип 2 байта [-32 768; 32 767];
- unsigned short int – целочислен тип 2 байта [0; 65 535];
- signed int (signed може да се пропусне) – целочислен тип 4 байта [-2 147 483 648; 2 147 483 647];
- unsigned int – целочислен тип 4 байта [0; 4 294 967 295];

- signed long int (signed може да се пропусне) – целочислен тип 4 байта [-2 147 483 648; 2 147 483 647];
- unsigned long int - целочислен тип 4 байта [0; 4 294 967 295].

Навсякъде в горните примери запазената дума int се подразбира и може да се пропусне.

Реалните числови константи са от следните типове (Visual C++ 6.0):

- double – реален тип (с плаваща десетична запетая). 8 байта, точност 15 цифри. ~ [10^{-308} .. 10^{308}] по абсолютна стойност;
- long double – реален тип (с плаваща десетична запетая). 10 байта, точност 19 цифри. ~ [10^{-4932} .. 10^{4932}] по абсолютна стойност;
- float – реален тип (с плаваща десетична запетая). 4 байта, точност 7 цифри. ~ [10^{-38} .. 10^{38}] по абсолютна стойност.

Една реална константа по подразбиране е от тип double. Могат да се добавят **суфикси** за да се промени нейният тип:

- F (f) за float (например 3.4f, –3.53F и т.н.)
- L (l) за long double (например 23.5l, -34.3L)

Символни константи – единичен символ, заграден в апострофи (‘’), един символ е един байт. Един символ се съпоставя на точно едно цяло число по определен начин. В езика С (както и в много други езици) стандартът е ASCII (American Standard Code for Information Interchange). Кодът се задава в десетична, осмична или шестнадесетична система.

Низови константи – последователност от символи, заградени в кавички. На всеки символ съответства байт, но накрая има един нулев байт (той е признак за край във вътрешното представяне). Представяне на особени символи в низ:

- \’ – представяне на апостроф
- \” - представяне на кавичка
- \\ - представяне на обратна черта

- \осм. код; \xшестн. код – представяне на символи, които нямат графично изображение
- \0 – нулев байт
- \b – връща курсора един символ назад
- \r – carriage return (позиционира курсора в началото на реда)
- \f – минава на следваща страница
- \n – минава на нов ред
- \t – табулация

6. Променливи и типове данни

Променлива – място за съхранение на данни – характеризира се с тип, име и стойност. Името е идентификатор, стойността е според типа на променливата.
Една променлива трябва да се дефинира задължително преди да се използва:
тип списък_от_променливи;
Типове:

- **char** – символен тип (1 байт). Може да е с модификация signed или unsigned. В C++ може да се интерпретира като числов тип;
- **int** – целочислен тип (при 16 битовите компилатори – 2 байта (int = signed int), при 32 битовите – 4 байта (int = signed long int). Допуска модификации signed, unsigned, long, short;
- **float** – тип с плаваща запетая (4 байта);
- **double** – тип с плаваща запетая (двойна точност) (8 байта); Допуска модификация long.

Когато компилаторът обработи дефиницията на променлива, той заделя памет в зависимост от нейният тип. Типът определя по какъв начин се кодира стойността на променливата в паметта, определя и множеството от допустимите стойности на променливата. Например – целият тип int обхваща всички цели числа от интервала
[-2 147 483 648; 2 147 483 647];
Не е така при double – там се вземат краен брой рационални числа от диапазона на стойностите.
В езика С няма вграден тип низ (подобно на Pascal). Променлива от тип низ е едномерен символен масив (от тип char).

7. Коментари, структура на програмата и етапи на нейната обработка

Програма за пресмятане на периметъра и лицето на кръг с радиус *r*

```
#include <iostream.h>
void main ()
 { float p, r, s;
   cout << "Въведи r = ";
   cin >> r;
   p = 2 * 3.14159 * r;
   s = 3.14159 * r * r;
   cout << "Периметърът P = " << p << endl;
   cout << "Лицето S = " << s << endl;
 }
```

Ред по ред:

1. Директива към препроцесора за включване на заглавен файл (iostream.h);

2. Дефиниране на главната функция main без аргументи и без параметри;
3. Разпределяне на памет за променливите (от тип float – 3x4 байта);
4. Извеждане на подканващо съобщение (низът зададен в кавичките);
5. Спиране изпълнението на програмата и подканване към потребителя да въведе стойност, която се записва в адреса на променливата;
6. Аритметична операция умножение (приложена два пъти) и оператор за присвояване;
7. Като 6.;
8. Извеждане на резултата. Низът се извежда символ по символ, после се извежда стойността на променливата, след това се преминава на нов ред (endl е константата endline);
9. Като 8.

Коментар – последователност от символи, заградени със знаците /* и */. Използват се за да се запишат пояснения в програмата,

обикновено за да се поясни какво точно прави определен израз, блок, функция и т.н. Програмата не губи ефективност ако има големи коментари, те се изключват от препроцесора преди компилацията. С коментари могат да се изолират отделни фрагменти от програмата (които са излишни или подлежат на корекции и т.н.). Влагане на коментари се допуска с опция (по подразбиране не се допуска). Друг начин да се дефинира коментар е със //. С този символ се обозначава коментар в рамките на един ред – в коментара попадат всички символи след този знак до края на реда.

Структура на програмата – програмата на С представлява съвкупност от функции (непразна). Точно една от функциите е главна (носи задължително името main). При изпълнението на една програма операционната система предава управлението точно на тази функция – main е входна точка на програмата. След завършване на изпълнението на главната функция управлението се връща към операционната система. Най-общо структурата е следната:

- описания на заглавни файлове;
- дефиниция на външни (глобални) променливи;
- функции f_1, f_2, …, f_n и главна функция.

Външните променливи са един от начините за обменяне на данни между функциите. В С не се допуска влагане на функции. Допуска се навсякъде в програмата да се използват стандартните функции, към компилатора има библиотека, която съдържа тяхната реализация.

Важно: Информацията от “h” (header) файла – указват се единствено прототипите на функциите (име, тип, аргументи), но не и реализацията.

Етапи за обработка на програмата

Програмата е текст. Въвежда се чрез текстов редактор, който предоставя възможност за лесна корекция на грешки. Препоръчително е програмата периодично да се съхранява във външната памет, тъй като тя се намира в оперативната и може да бъде загубена). Една програма на С обикновено е с разширение .c, а на С++ с разширение .cpp:

1. Преди да се компилира програмата, тя минава през обработка от препроцесор – изчистват се коментарите, изпълняват се директивите, означени с # и др.;
2. След това програмата се обработва от компилатора. Ако има синтактични (и др.) грешки, компилаторът извежда съответното съобщение. След успешно компилиране се създава .obj-файл (обектен код);
3. На следващия етап свързващ редактор преобразува .obj файла в изпълним (.exe) файл. Свързващият редактор обединява в едно обектния код на програмата и обектния код на стандартните функции. Дава съобщение за грешка, ако пътят

към библиотеката със стандартните функции е зададен неправилно. Изпълнимият файл (.exe) е вече на машинен език;

4. По време на изпълнение на програмата може да възникнат семантични грешки поради проблем с входните данни или в самата програма. И накрая изпълнението на програмата може да е успешно, но тя да не връща очаквания резултат – тогава програмистът търси алгоритмична или семантична грешка.

8. Операции и изрази

Операциите са три вида:

- **унарни** (едноаргументни) – ‘+’, ‘-‘
- **бинарни** (двуаргументни) – ‘ + ’,’ - ‘,’ *’, ‘ / ‘, ‘ %’ – остатък от целочислено деление
- **тернарни** (триаргументни) – операция за условен израз

Операндите са променливи, константи или обръщения към функции. Ако и двата операнда са цели, резултатът е цял. Ако един от операндите е реален, то резултатът е реален. ‘ %’ се използва само с цели числа

В С няма вграден булев тип. Всяка стойност различна от 0 се счита за истина (true), а всяка стойност равна на 0 се счита за лъжа (false). Ако една операция връща булева стойност, то тя е 1 ако е истина и 0 ако е лъжа.

Задача: Правоъгълен триъгълник има катети x и y и хипотенуза z. По дадени z и a = x – y да се определят x и y. (Входът е коректен.)

Формулите са:

$x = (a + \sqrt{2z^2 - a^2})/2$

$y = (- a + \sqrt{2z^2 - a^2})/2$

```
#include <iostream.h>
#include <math.h>
void main ()
 { double z, a, x, d, y;
   cout << "Въведете z и a: " << endl;
   cin >> z >> a;
   d = sqrt (2*z*z - a*a);
   x = (a + d) / 2;
   y = (- a + d) / 2;
   cout << "x = " << x << " y = " << y << endl;
}
```

В езиците за програмиране изразите се записват линейно, а не многоетажно.

Израз е валидна комбинация от операции, операнди и малки скоби. Съобразен е със синтактичните правила и със семантичните изисквания. Операндите могат да бъдат променливи, константи, елементи на масиви или обръщения към функции.

Операции за отношение: >, >=, <=, <, ==, !=

Резултатът от тях е истина (true) или лъжа(false).

Логически операции:

&& - логическо “и” – операнд1 && операнд2 && ... && операндN.

Особеност – компилаторите правят оптимизации (къса схема). При израз от горния тип по време на изпълнението изчисляването се преустановява, когато някой операнд има стойност лъжа, тъй като тогава целият израз има стойност лъжа.

|| - логическо “или” - операнд1 || операнд2 || ... || операндN.

Същата особеност, както при логическото “и” – по време на изпълнението изчисляването се преустановява, когато някой

операнд има стойност истина, тъй като тогава целият израз има стойност истина.
! – логическо отрицание.
Важно: Оптимизациите могат да пропуснат някои операнди, заедно с евентуалните промени, които се правят в тях.

Операция за присвояване: В C++ присвояването е операция за разлика от другите езици.
име_на_променлива = израз – пресмята се израза и резултатът, преобразуван по тип се записва в адреса на променливата.
Резултатът от тази операция е стойността на десния операнд.
Пример: a = b = 2; a = b = c = d и т.н. Изпълняват се отдясно наляво.
Оператор за присвояване:
име_на_променлива = израз **;** (не връща резултат)

Често срещани **комбинации от операции**:

- a = a + b, съкратено a += b

- a = a - b, съкратено a -= b
- a = a * b, съкратено a *= b
- a = a / b, съкратено a /= b
- a = a % b, съкратено a %= b
- a = a + 1, съкратено a++ или ++a
- a = a - 1, съкратено a-- или --a

Когато ++ (--) е преди променливата първо се изпълнява операцията, а след това променливата се използва с новата си стойност. Ако са след променливата първо се използва променливата със старата стойност и след това се изпълнява операцията.

Примери:

```
int x, y;
x = 15
y = ++x; |след операцията x = 16; y = 16
```

```
x = 12
y = x--; | след операцията x = 11; y = 12
x++;
++x; (ако са самостоятелни няма значение дали ++ (--
) е преди или след променливата)
```

Операция за последователно изпълнение – операция запетая ‘,’:

операнд1**,** операнд2**,** ... операндN – последователно се пресмятат операндите и резултатът от операцията е стойността на операндN.
Премятане на сумата от първите 100 естествени числа:

```
for (sum=1, i=1; i<100; i++,sum += i)
        ;
```

Операция за преобразуване на тип:
(тип) операнд

При някои компилатори тип (операнд) . Преобразува операнд в даден тип. Пресмята се стойността на операнда и получената стойност се преобразува в указания в скобите тип.

Пример:

```
int uspeh, broi;
float srus;
srus = (float) uspeh/broi;
```

Операция за размер на даден обект:

sizeof (операнд) – дава като резултат размера на операнда в байтове. Например sizeof (srus) връща 4. sizeof (int) връща 2 или 4 (в зависимост от това дали компилатора е 16 или 32 битов). Като операнд на операцията може да се използва променлива от всеки валиден тип, включително структури, обединения и дефинирани от потребителя променливи, изрази или просто ключова дума за

определен тип данни. Особеност на тази операция е, че се изпълнява по време на компилация, а не по време на изпълнение. Тя не увеличава времето за изпълнение на програмата и осигурява машинна независимост и преносимост на програмата.

Ред на изпълнение на операциите в един израз – зависи от приоритета на операциите. Операциите с по-висок приоритет се изпълняват преди операциите с по-нисък приоритет. Ако операциите са с един и същи приоритет те се изпълняват отляво надясно ако са ляво-асоциативни или отдясно наляво ако са дясно-асоциативни. Малките скоби указват операцията в тях да се изпълни преди останалите операции, независимо от приоритета.

Таблица на приоритетите:

Приоритет	Операции	Асоциативност
1	() – извикване на функция и групиране в изрази [] – достъп до елемент на масив	ляво-асоциативни
2	‘!’ – логическо отрицание ‘+’ - унарен плюс ‘-‘ - унарен минус ‘++’ – прибавяне на 1 ‘--‘ – изваждане на 1 & - адресиране (тип) – преобразуване на тип sizeof – размер на обект	дясно-асоциативни
3	‘*’ – умножение ‘/’ – деление ‘%’ – деление по модул	ляво-асоциативни

<table>
<tr><td>4</td><td>‘+’ – събиране
‘-‘ - изваждане</td><td>ляво-асоциативни</td></tr>
<tr><td>6</td><td><, <=, >=, >- операции за отношение “по-малко, по-голямо”</td><td>ляво-асоциативни</td></tr>
<tr><td>7</td><td>==, != - операции за отношение “равно, не равно”</td><td>ляво-асоциативни</td></tr>
<tr><td>11</td><td>&& - логическо “и”</td><td>ляво-асоциативни</td></tr>
<tr><td>12</td><td>|| - логическо “или”</td><td>ляво-асоциативни</td></tr>
<tr><td>13</td><td>? : - операция за условен израз</td><td>дясно-асоциативни</td></tr>
<tr><td>14</td><td>=, +=, -=, *=, /=, %= - операции за присвояване</td><td>дясно-асоциативни</td></tr>
<tr><td>15</td><td>‘,’ – операция за последователно изпълнение</td><td>ляво-асоциативни</td></tr>
</table>

9. Преобразуване на типовете

Автоматично преобразуване на типовете на операндите се извършва при изпълнение на бинарните операции.
Синтаксисът на бинарна операция е: операнд1 операция операнд2
Преди изпълнението на почти всички операции се извършва уеднаквяване на типовете на операндите, които участват в операцията. Уеднаквяването става към **по-старшия** от двата операнда. Един операнд е по-старши, ако се представя с повече байтове или в случай на равенство, който може да приема по-големи стойности. Числовите типове, сортирани по приоритета в уеднаквяването:

1. long double – 10 байта;
2. double – 8 байта;
3. float – 4 байта;
4. unsigned long int – 4 байта;
5. long int – 4 байта;
6. unsigned int – 4 байта;

7. signed int – 4 байта;
8. unsigned short int – 2 байта;
9. short int – 2 байта;
10. unsigned char – 1 байт;
11. signed char – 1 байт;

Правила за преобразуване на типовете при бинарни операции

Особеност – char винаги се преобразува към int. По-нататък преобразуването продължава в следния ред:

Ако някой от операндите е от тип long double, то другият се преобразува към long double.

В противен случай, ако някой от операндите има тип double, то другият се преобразува към double.

В противен случай, ако някой от операндите има тип float, то другият се преобразува към float.

В противен случай за двата операнда се извършва целочислено преобразуване към ”по-голям” тип. След това, ако един от

операндите има тип unsigned long int, то другия се преобразува в unsigned long int.
В противен случай, ако един от операндите е от тип long int, а другият от unsigned int, то резултатът зависи от това, обхваща ли long int всички стойности на unsigned int, и ако това е така, то unsigned int се преобразува към long int. Ако не, то двата операнда се преобразуват в unsigned long int.
В противен случай, ако един от операндите има тип long int, то другият се преобразува към long int.
В противен случай, ако един от операндите е unsigned int, то другият се преобразува към unsigned int.
В противен случай двата операнда имат тип int.

Правила за преобразуване на типовете при оператор за присвояване

Синтаксисът на оператор за присвояване е:

име_на_променлива = израз **;**

Ако име_на_променлива е от тип bool, израз може да бъде от тип bool или от кой да е числов тип.

Ако име_на_променлива е от тип double, всички числови типове, а също типът bool, могат да са типове на израз.
Ако име_на_променлива е от тип float, типовете float, short, unsiged short и bool, могат да са типове на израз. Ако израз е от тип int, unsigned int или double, присвояването може да се извърши със загуба на точност. Компилаторът предупреждава за това.
Ако име_на_променлива е от тип int, типовете int, long int, short int и bool, могат да са типове на израз. В този случай ако израз е от тип double или float, дробната част на стойността на израз ще бъде отрязана и ако полученото цяло е извън множеството от стойности на типа int, ще се получи случаен резултат. Компилаторът издава предупреждение за това.
Ако име_на_променлива е от тип short int, типовете short int и bool, могат да са типове на израз. В противен случай се извършват преобразувания, които водят до загуба на точност или даже до случайни резултати. Много компилатори не предупреждават за това.

10. Условни изрази и условни оператори

Операция за условен израз (тернарна операция): ?:

Синтаксис: операнд1 ? операнд2 : операнд3
Операнди могат да бъдат валидни за езика комбинации от променливи, константи, елементи на масиви и свързващи ги операции. Тази операция се изпълнява по следния начин.
Пресмята се операнд1. Ако стойността му е истина (≠0), тогава се пресмята операнд2 и стойността му е резултатът от изпълнението на операцията. Ако стойността му е лъжа (=0), тогава се пресмята операнд3 и стойността му е резултатът от изпълнението на операцията. Изчислява се или се изпълнява, ако е функция, само единият от операнд2 или операнд3. Ако операнд2 и операнд3 са от различен тип, то резултатът се преобразува в старшия тип.
Пример:
int x, y;

x = 1; y = 2;
y = x < y ? 10 : 100; |y става 10

Операнд1 се изчислява винаги първи и се оценява дали има нулева или ненулева стойност. Обикновено не е нужно той да се поставя в малки скоби, но се препоръчва, тъй като подобрява разбираемостта на програмите.

Използването на тази операция води до много по-компактен и по-ефективен код. Например фрагментът

```
if (a > b)
    z = a;
else
    z = b;
```

може да бъде написан като z = (a > b) ? a : b;

Тъй като това е операция, за разлика от оператора if else тя не позволява използването на други оператори като операнди.

Операцията е дясно-асоциативна. Ако имаме:

операнд1 ? операнд2 : операнд3 ? операнд4 : операнд5

първо се изчислява (операнд3 ? операнд4 : операнд5) и след това с получения резултат се изчислява (операнд1 ? операнд2 : резултат). Ако искаме първо да се изпълни (операнд1 ? операнд2 : операнд3), трябва да поставим малки скоби.
Програма за пресмятане на заплащането за 1 седмица (почасова система за плащане). Нека работникът е работил h часа и за 1 час получава norma лева. Всеки извънреден труд се заплаща с 50% повече (над 40 часа на седмица).

suma = h.norma, h≤40; 40.norma + 1,5.(h-40).norma, h>40

```
#include <iostream.h>
void main ()
 { int cod;
   float h, norma, suma;
   cout << "Въведете cod, h, norma: ";
   cin >> cod >> h >> norma;
```

```
    suma = (h<=40) ? (h*norma) : (40*norma + 1.5*(h-40)*norma);
    cout << "cod = " << cod;
    cout << " suma = " << suma << endl;
}
```

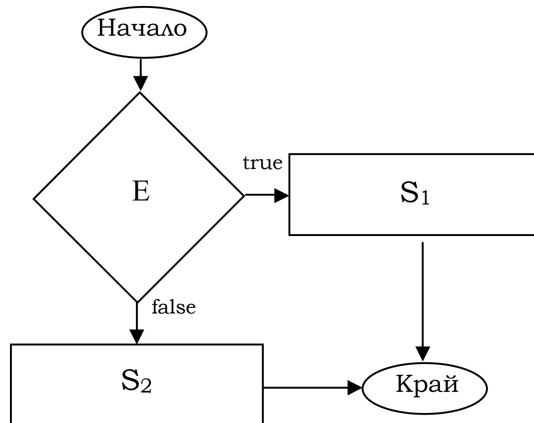
Условни оператори

С тези оператори се реализират разклоняващи се изчислителни процеси. Оператор, който дава възможност да се изпълни (или не) един или друг оператор в зависимост от някакво условие, се нарича **условен**. Ще разгледаме следните условни оператори: if else, if и switch.

Условен оператор if else

Операторът се използва за избор на една от две възможни алтернативи в зависимост от стойността на дадено условие. Чрез

него се реализира разклоняващ се изчислителен процес от вид, илюстриран на блок-схемата:



Ако указаното условие е истина, се изпълнява оператор S_1, а ако е лъжа – оператор S_2. И в двата случая след това се изпълнява операторът, който следва оператора if else.

Условието се задава с булев израз, а оператор S_1 и оператор S_2 – чрез оператор(и).

Синтаксис на условен оператор if else:

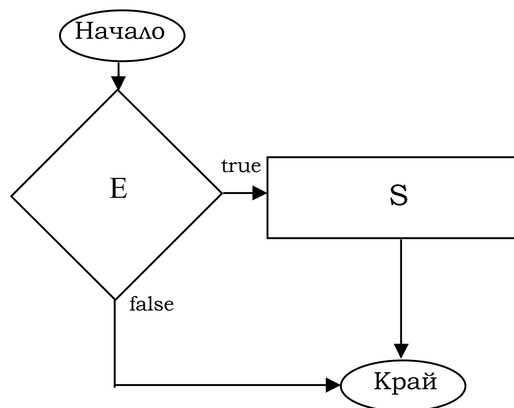
if (E) S_1 else S_2

E – булев израз; S_1 и S_2 – оператори

Булевият израз трябва да бъде напълно определен. Задължително се огражда в малки скоби. Операторът след условието е точно един. Ако е необходимо няколко оператора да се изпълнят, трябва да се обединят в блок (съставен оператор). Операторът след else е точно един. Ако е необходимо няколко оператора да се изпълнят, трябва да се обединят в блок (съставен оператор).

Условен оператор if

С този условен оператор се реализира разклоняващ се изчислителен процес от вид, илюстриран на блок-схемата:



Ако указаното условие е истина, се изпълнява оператор S, ако е лъжа, оператор S се прескача. И в двата случая след това се изпълнява операторът, който следва оператора if.
Условието се задава с булев израз, а оператор S – чрез оператор(и).

Синтаксис на условен оператор if:
if (E) S
E – булев израз; S – оператор
Булевият израз трябва да бъде определен. Огражда се в малки скоби. Операторът след условието е точно един. Ако е необходимо няколко оператора да се изпълнят, трябва да се обединят в блок (съставен оператор).
В последната програма можем да заменим операцията за условен израз с условния оператор: if (h<=40) suma = h*norma;
else suma = 40*norma + 1.5*(h-40)*norma;
Операцията за условен израз е по-ефективна от условния оператор и нейната употреба е за предпочитане в случай, че има възможност за избор.
Важно: Принципът за поставяне на “**;**” в C е различен от този в Pascal. **В C “;” се поставя в края на всеки оператор.**
Празен оператор – това е най-простият оператор.
Синтаксис: **;**

Операторът не съдържа никакви символи. Завършва със знака ; (точка и запетая).
Не извършва никакви действия. Използва се, когато синтаксисът на някакъв оператор изисква присъствието на поне един оператор, а логиката на програмата не изисква такъв.

Съставен оператор (блок) – няколко оператора заградени с големи скоби. Използва се, когато трябва да се изпълнят повече от един оператори, но да се разглеждат като един оператор. Пример:

```
float x, y, max;
if (x>y) { max = x; x--; }
     else { max=y; --y; }
```

Вложени условни оператори

В условните оператори:

if (<условие>) <оператор>

if (<условие>) <оператор1> else <оператор2>

<оператор>, <оператор1> и <оператор2> са произволни оператори, в т. число могат да бъдат условни оператори. В този

случай имаме вложени условни оператори. При влагането е възможно да възникнат двусмислици. Ако в един условен оператор има повече запазени думи if отколкото else, възниква въпросът, за кой от операторите if се отнася съответното else.

Записът със съответни подравнявания, не влияе на компилатора. В езика C++ има правило, което определя начина по който се изпълняват вложени условни оператори.

Правило: Всяко else се съчетава в един условен оператор с най-близкото преди него несъчетано if. Текстът се гледа отляво надясно.

if (E_1) if (E_2) S_1 else S_2

Този израз допуска две различни тълкувания:

- ако съпоставим else на втория if, вторият if е в пълна форма, а първия е в кратка форма. Този начин е възприет в C. Текстът на програмата се обработва от ляво надясно и когато се попадне на else, се търси първият if, който е наляво от този else;

- ако съпоставим този else на първия if, вторият if е в кратка форма. Операторът, написан в горния формат, няма да се изпълни по този начин, освен ако не заградим в големи скоби вторият if - if (E_1) { if (E_2) S_1 } else S_2 - тогава компилатора ще свърже else с първия if, тъй като вторият е част от съставен оператор.

Добре е да се избягват сложните конструкции. Опростяването на програмата повишава нейната ефективност. Препоръчва се условният оператор да се използва в следния вид (ако алгоритъмът го позволява):

if (E_1) S_1
    else if (E_2) S_2
        else if (E_3) S_3
…
            else if (E_n) S_n
                [else S_{n+1}]

при условие, че S_i не са условни оператори за всяко i. Този условен оператор работи по следния начин: ако някое E_i е изпълнено, но E_1, E_2, ..., E_{i-1} не са изпълнени, тогава се изпълнява S_i. Ако никое E_i не е изпълнено, тогава ако последният ред присъства, се изпълнява S_{n+1}, в противен случай нищо не се изпълнява.

Пример за условен оператор с по-сложен вид на условието:

```
if ((r=getchar())!='Y' && r!='y') оператор;
```

getchar е функция, която чете символ. В това условие съществено е, че на r се присвоява стойност в първия операнд на логическото "и". **Трите операции,** които изискват **изпълнението** да се започне **от първи операнд са операция за условен израз, логическо "и"** (&&) **и логическо "или"** (||)**.** При това положение трябва да се внимава с комутативните операции.

11. Оператори за цикъл while и do-while

Операторите за цикъл се използват за реализиране на циклични изчислителни процеси.
Изчислителен процес, при който оператор или група оператори се изпълняват многократно за различни стойности на техни параметри, се нарича **цикличен**.
Тези оператори се наричат **тяло на цикъла**.
Съществуват два вида циклични процеси:
- индуктивни и
- итеративни.

Цикличен изчислителен процес, при който броят на повторенията е известен предварително, се нарича **индуктивен цикличен процес**.
Цикличен изчислителен процес, при който броят на повторенията **не** е известен предварително, се нарича **итеративен цикличен процес**. При тези циклични процеси, броят на повторенията зависи от някакво условие.

В езика C++ има три оператора за цикъл:

- *оператори while и do/while*

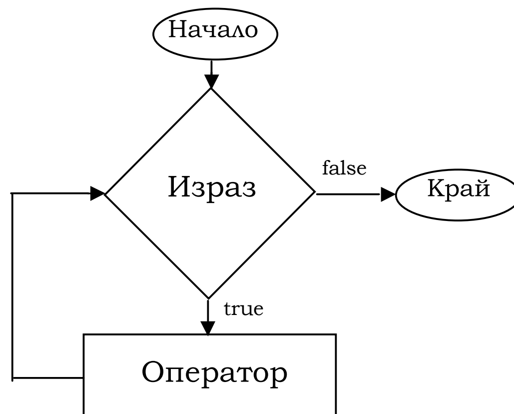
Използват се за реализиране на произволни циклични процеси – индуктивни и итеративни.

- *оператор for*

Чрез него могат да бъдат реализирани произволни циклични процеси, но се използва главно за реализиране на индуктивни циклични процеси.

Оператор за цикъл while

С този оператор за цикъл се реализира цикличен процес от вид, илюстриран на блок-схемата:



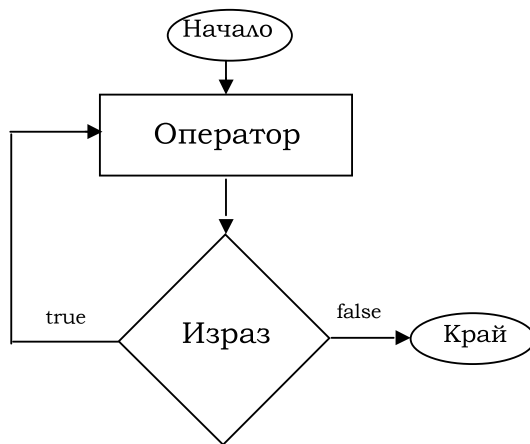
Синтаксисът на оператора while е:
while (израз)
  оператор

където израз е условие, което управлява продължителността на цикъла. Може да е всеки допустим за езика израз. Оператор е произволен прост или съставен оператор.
Операторът while предизвиква следните действия:

1. Изчислява се стойността на израз (условието за изпълнението на цикъла).
2. Ако стойността на израз е истина, изпълнява се оператор (тялото на цикъла) и управлението се връща в т.1 за следваща проверка на израз.
3. Ако стойността на израз е лъжа, без да се изпълнява оператор, се преминава към изпълнение на оператора, който следва непосредствено след него.

Оператор за цикъл do-while

С този оператор за цикъл се реализира цикличен процес от вид, илюстриран на блок-схемата:



Синтаксис на оператора do while:

```
do
  оператор
while (израз);
```

където оператор (тялото на цикъла) и израз се дефинират по същия начин както при оператора while. Действието на оператора do-while обаче е принципно различно от това на оператора while:

1. Изпълнява се оператор (тялото на цикъла).
2. Изчислява се стойността на израз (условието за изпълнението на цикъла).
3. Ако стойността на израз (условието за изпълнението на цикъла) е истина, отново се изпълнява оператор (тялото на цикъла).
4. Ако стойността на израз (условието за изпълнението на цикъла) е лъжа, се преминава към изпълнение на оператора, който следва непосредствено след оператора do while.

Принципната разлика между операторите while и do-while е в това, къде се проверява условието за край на цикъла. При оператора while проверката се извършва в началото на цикъла, а при оператора do-while – в края на цикъла. От това следва, че тялото на цикъла в оператора do-while винаги се изпълнява веднъж, на

първата итерация, независимо от стойността на условието за спиране.
За разлика от оператора while в оператора do-while частта while завършва с разделител ;. Това е добра информация за локализирането на операторите do-while, когато анализираме програми. Ако при програмирането пропуснем разделителя ;, компилаторът ще сигнализира, че в оператора do-while липсва затварящата ключова дума while. А записаният без разделител while(израз) ще се свърже с непосредствено следващия го оператор и ще се оформи условен оператор while. Това, разбира се, изобщо няма да отговаря на нашите намерения. Познаването на тази особеност е важно при коригирането на грешки при компилиране.

Особености: променливите, които се променят в цикъла трябва да се инициализират преди това. Поне една променлива, която участва в израза, трябва да се променя в тялото на цикъла, в противен случай рискуваме да получим зацикляне.

Пример за зацикляне:

```
int x, a;
x = 15;
while (x < 25)
 a = 2*x;
```

Задача: От клавиатурата се въвежда текст, който завършва със ‘ *’. Да се определи колко пъти даден символ се среща в текста.

```
#include <iostream.h>
#include <stdio.h>
void main ()
{   int i, symbol, r;
//     Въвеждане на символа
    symbol = getchar();
    r = getchar(); //Премахване на Enter от буфера на 
getchar
//     Въвеждане на текста
    i = 0;
    while ( ( r = getchar() ) != '*' )
     if (r == symbol) i++;
    cout << " Честотата на символа в текста = " << i;
}
```

Пример:

```
int i;
float x;
i = 1;
x = 0.005;
while (x != 0.995)
 { cout << "i = " << i << ", x = " << x << endl;
   i++; x+ = 0.01;
 }
```

Този фрагмент води до безкрайно зацикляне, въпреки че то не е явно. Строго математически, цикълът би трябвало да завърши, защото след 99 итерации на тялото на цикъла x приема стойност 0.995. Проблемът е в грешките при закръглянето на числата с плаваща запетая. Поради тази причина x никога не приема стойност точно 0.995, а някое друго число съвсем близко до него. Ако сменим израза на цикъла while с x<=0.995, тогава цикълът със сигурност ще приключи.

12. Масиви. Низ от символи

Масив е множество последователно разположени в паметта еднотипни променливи, притежаващи общо име. Името на масива е аналогично на име на променлива, т.е. валиден идентификатор. Всяка променлива в масива е прието да се нарича елемент на масива.

Преди да се използва един масив, той трябва да се дефинира. Описанието на масив има следния синтаксис:

тип име[израз];

За тип се използва ключова дума за тип данни от разгледаните по-горе. Тип на масива е типът на неговите елементи, а те са от един и същ тип. Име е валиден идентификатор, както име на променлива. Средните скоби [] определят, че зададеният идентификатор е име на масив, а не име на променлива. Поставеният в средни скоби израз определя броя на елементите в масива и този брой се нарича граница на масива. Изразът трябва да е константен, т.е. съставен от цели числови константи или от

символни константи. Обикновено за граница на масива се задава цяло десетично число.
Например int vect[30]; char a[3], b[4]; float c[100];
Всеки елемент на масив може да бъде разглеждан и обработван като променлива, т.е. да приема различни стойности и да участва в изчисления. За да можем да използваме даден елемент на масив, към него се обръщаме по името му, както при променливите. Разликата е, че всеки елемент на масива има едно и също име – името на масива, към което се добавя поставен в средни скоби индекс. Индекс на елемента на масива е цяла константа, променлива от цял тип, най-общо казано израз с целочислена стойност. Индексът определя позицията на съответния елемент спрямо началото на масива. Първият елемент на масива винаги има индекс нула. Максималният индекс на всеки масив е с 1 по-малък от границата на масива, зададена в описанието му.
Валидни индекси:
array[3], array[‘A’], array[i+5], array[i*j]

Компилаторите не проверяват дали **индексът е в допустимите граници** за даден масив. Ако индексът излиза от тези граници, тогава обръщението е към оперативната памет, която е непосредствено след или преди елементите на масива. Такава грешка се открива много трудно.

Внимание: обръщането към елемент, който е извън масива (чрез задаване на некоректен индекс) може да изтрие важна информация за операционната система и дори да я блокира.

Разполагане на масив в оперативната памет – отделните елементи се разполагат в последователни полета от оперативната памет, наредени по индекс, всяко поле заема толкова байтове, колкото е типа на масива.

В езика С водещ принцип е получаването на ефективни програми, затова елементите на масива се номерират от 0, което е неудобно за потребителя.

Горните масиви се наричат **едномерни**, защото имат само една размерност (индекс). Едномерните масиви са аналог на векторите в математиката. Ако си представим, че елементите на едномерния

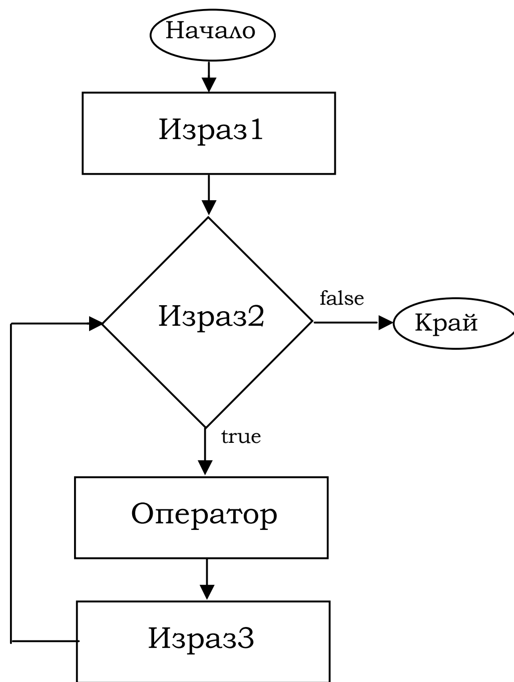
масив са едномерни масиви (вектори), получаваме **двумерен масив и т.н.** Елементите на двумерните масиви се означават с два индекса. Матриците са аналог на двумерните масиви.
Дефиниране на двумерен масив – float mat[20][30];
Достъп до елемент на двумерен масив – mat[i][j] , първият индекс е за редовете, вторият за стълбовете и за всяка размерност отделна двойка скоби. Необходимата памет за един масив е произведението на границите на отделните размерности и паметта, необходима за един елемент.
Разполагането на масиви от всякакви размерности в паметта става последователно (линейно), по реда на нарастване на десните индекси, т.е. десните индекси нарастват преди левите. За двумерните масиви това означава, че те се записват в паметта по редове.
Например:
mat[0][0],…,mat[0][29],mat[1][0],…,mat[1][29],…,mat[19][0],…,mat[19][29]

За тримерен масив – double table[4][3][2], елементите се разполагат по редове както следва:
table[0][0][0], table[0][0][1], table[0][1][0], table[0][1][1], table[0][2][0], table[0][2][1], table[1][0][0], …., table[3][2][1],
достъп до елемент - table[i][j][k];

Съгласно общата дефиниция за масиви те могат да бъдат съставени и от променливи от тип char, например char name[12]; Масив от тип char представлява последователност от символи. В него може да се запише произволен текст. В C последователност от символи се нарича низова променлива. В C **няма вграден тип за низ.** Аналогично на низовите константи и низовите променливи трябва да завършват с нулев байт. Това означава, че последният елемент на такава последователност от символи трябва да е символът ‘\0’. Грижа на програмиста е да осигури място за нулевия байт и да го зададе. Например char a[20]; - разглежда се като променлива от тип низ от символи; броят на символите, обаче е с едно по-малък заради нулевия байт, който стои на последно място. За константата от тип низ от символи “abc” масивът от тип char изглежда така: a[0]= ‘a’; a[1]=’b’; a[2]=’c’; a[3]=’\0’;.

13. Оператор за цикъл for

Принципната разлика между операторите **while** и **do-while** е в това, където се проверява условието за край на цикъла. При оператора **while** проверката се извършва в началото на цикъла, а при оператора **do-while** – в края на цикъла.
Тъй като първото действие, свързано с изпълнението на оператора **while**, е проверката на условието му за край на цикъла, операторът се нарича още оператор за цикъл с пред-условие.
Проверката на условието за край на цикъла **do-while** е в края му и затова той се нарича цикъл с пост-условие.
Операторът за цикъл **for** е най-сложният оператор за цикъл в C.
Той също е от тип цикъл с пред-условие.
С този оператор за цикъл се реализира цикличен процес от вид, илюстриран на блок-схемата:



Синтаксис:

for (израз1; израз2; израз3)

оператор

където израз1 е израз за инициализация. С този израз се присвояват началните стойности на параметрите на цикъла.

израз2 – условие за проверка, управляващо продължителността на цикъла.
израз3 – израз за изменение на текущите стойности на параметрите на цикъла.
оператор – произволен прост или съставен оператор, образуващ тялото на цикъла.
Конструкцията на оператора **for** позволява израз1, израз2 и израз3 да бъдат произволни, допустими за езика изрази. Обикновено тези изрази са свързани логически помежду си по следния начин: с израз1 се установяват началните стойности на променливите, които участват в условието за проверка, а с израз3 се определя някакво правило, по което тези променливи се изменят на всяка итерация на цикъла. Най-често израз1 и израз3 са изрази за присвояване, а израз2 е израз за сравнение. С изпълнението на оператора **for** се поражда следната последователност от действия:

1. Изчислява се стойността на израз1, която се използва като начална стойност на управляващата променлива на цикъла.

2. Изчислява се стойността на израз2, която зависи от управляващата променлива.

3. Ако стойността на израз2 е различна от нула, изпълнява се операторът в тялото на цикъла. След това се изчислява стойността на израз3 и се присвоява нова стойност на управляващата променлива. Управлението се връща в т.2, за да се осъществи следващата итерация на цикъла.

4. Ако стойността на израз2 е равна на нула, изпълнява се операторът, записан непосредствено след тялото на цикъла.

Най-често **for** се използва за цикъл с предварително известен брой повторения (итерации).

Всеки оператор **for** е еквивалентен на следната последователност:

```
израз1;
while(израз2)
{
оператори
израз3;
}
```

Израз1 винаги се изпълнява само веднъж, в началото на цикъла. Ако стойността на израз2 е нула още на първата итерация, израз3 изобщо не се изпълнява. Препоръчва се всички променливи, които се използват в израз1, израз2, израз3 предварително да се дефинират.
Препоръка: Ако горната граница на цикъла е зададена чрез променлива, тя може да се променя в тялото на цикъла. Управляващата променлива на цикъла запазва своята последна стойност след завършване на цикъла и може да бъде използвана в следващите изчисления. Няма значение дали цикълът е завършил нормално или е прекъснат принудително. Стойностите на променливите, участващи в израз2 и израз3, могат да се променят в тялото на цикъла. Управляващата променлива не е задължително понятие за цикъл **for**. Цикъл **for** може да бъде организиран без явно да се използва управляваща променлива, чиято стойност е броят на извършените итерации.
В синтаксиса на оператора **for** израз1, израз2 и израз3 не са задължителни. Всеки от тях поотделно и в комбинация с другите

може да бъде пропуснат. В тези случаи разделителите **”;”** , отнасящи се към пропуснатите изрази, трябва задължително да присъстват. Допустими са оператори от вида:

for(;израз2;израз3)
    оператор
for(;;израз3)
    оператор
или дори
for(;;)
    оператор

В случаите, когато е изпуснат израз2, по подразбиране се приема, че той винаги е истина. Излизането от такива цикли (безкрайни) винаги трябва да става принудително.

Ако израз1 и израз3 са пропуснати, но израз2 го има, тогава за предпочитане е вместо цикъл **for** да се използва цикъл **while**.

Примерни фрагменти:масивът а е предварително нулиран, а след това в него са записани някакъв брой числа, различни от нула, започвайки от първия му елемент, поне един нулев елемент

съществува. За да определим колко числа са записани в него използваме следния фрагмент:

```
int a[100];
int count;
// запис на числа в началните елементи на масива
for (count = 0; a[count] != 0; count++)
        ;
```

В масив а е записан низ. Към този низ ще добавим друг, за което има място. Най-напред трябва да открием края на първия низ. Низът завършва с нулев байт ‘\0’. Откриването на края на низа става със следния фрагмент:

```
char a[100];
// въвеждане на елементите на низа
for (count = 0; a[count++] != ‘\0’ ; )
        ;
```

Изпълнението на цикъла ще завърши, когато в масива достигнем нулевия байт. След завършване на изпълнението нулевият байт ще има индекс count-1.

```
for (count = -1; a[++count] != '\0' ; )
        ;
```

След завършване на изпълнението нулевият байт ще има индекс count.

```
for (suma = 0, i = 0; i < 100; suma += a[i++]);
```

След завършване на цикъла suma ще съдържа сумата на всички елементи на масива a.

Възможно е да се влагат цикли един в друг.

Програма, която извежда набор от данни по нагледен начин в псевдографичен режим с последователност от символи *. Въвежда се и мащабен множител p, който е цяло число, на което се разделя всяко число от данните, преди да се покаже на екрана. Данните са положителни числа.

```
#include <iostream.h>
void main ()
 { int i, j, k, m, p;
   float array[25];
   k = 0;
   while (k==0)
     { cout << " Въведете броя на данните: ";
       cin >> m;
       if ((m > 0) && (m <=25)) k = 1;
       else cout << " Грешен брой!\n ";
     }
   cout << " Въведете мащабен множител: ";
   cin >> p;
   for ( i = 0; i < m; i++)
    { cout << " Въведете данна номер " << i+1 << " : ";
      cin >> array[i];
     }
   for ( j = 0; j < m; j++)
```

```
  { cout << array[j] << " |";
    for (i = 0; i < array[j]/p; i++)
     cout << "*";
    cout << "\n";
  }
}
```

Фрагмент от програма, който отпечатва стойностите на масив и не трябва да се променя, ако сменим размерността на масива:

```
int a[15], i;
for ( i = 0; i < sizeof(a)/sizeof(int); i++)
 cout <<”a[“ << i << “]=” << a[i] << endl;
```

Примерни програми с цикли

Да се напише програма, която по дадени реални числа x и ε (ε > 0), приближено пресмята сумата

$$s = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots$$

Сумирането да продължи докато абсолютната стойност на последното добавено събираемо стане по-малка от ε.

В тази задача броят на повторенията предварително не е известен, а зависи от условието $|x1| < \varepsilon$, където с x1 е означено произволно събираемо. Ще използваме оператора за цикъл while.

```
#include <iostream.h>
#include <math.h>
int main()
{cout << "x= ";
 double x;
 cin >> x;
 if (!cin)
 {cout << "Error, Bad input! \n";
  return 1;
 }
 cout << "eps= ";
 double eps;
 cin >> eps;
 if (!cin)
 {cout << "Error, Bad input! \n";
  return 1;
 }
```

```
if (eps <= 0)
 {cout << "Incorrect input! \n";
  return 1;
 }
 double x1 = 1;
 double s = 1;
 int i = 1;
 while (fabs(x1) >= eps)
 {x1 = x1 * x / i;
  s = s + x1;
  i++;
 }
 cout << "s=" << s << "\n";
 return 0;
}
```

Решение с цикъл for: задава се естествено число n

```
#include <iostream.h>
int main()
double x;
int n;
.......................................

double x1 = 1;
double s = 1;
for (int i = 1; i <= n; i++)
{x1 = x1 * x/i;
 s = s + x1;
}
cout << "s= " << s << "\n";
return 0;
}
```

Да се напише програма, която пресмята приближено следната безкрайна сума:

$$S = \frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

Сумирането да продължи докато абсолютната стойност на разликата на последните две добавени събираеми стане по-малка от ε. Ще използваме оператора do/while.

```
#include <iostream.h>
#include <math.h>
int main()
{cout << "x= ";
 double x;
 cin >> x;
 if (!cin)
 {cout << "Error, Bad input! \n";
  return 1;
 }
 cout << "eps= ";
 double eps;
 cin >> eps;
 if (!cin)
 {cout << "Error, Bad input! \n";
  return 1;
 }
```

```
 if (eps <= 0)
 {cout << "Incorrect input! \n";
  return 1;
 }
  double x1;
  double x2 = x;
  double s = x;
  int i = 2;
  do
  {x1 = x2;
   x2 = -x1 * x * x / (i*(i+1));
   s = s + x2;
   i = i + 2;
  } while (fabs(x1-x2) >= eps);
  cout << "s=" << s << "\n";
  return 0;
}
```

14. Оператор за избор на вариант switch. Оператори break, continue и goto

Оператор break – оператор за прекъсване на изпълнението на цикли и оператор **switch**. Може да бъде използван на произволно място в тялото на един цикъл. Синтаксисът на оператора е следният:

break;

Изпълнението на **break** предизвиква излизане от цикъла и предаване на управлението на оператора, записан непосредствено след цикъла. След изпълнението на **break**, стойностите на променливите, които са участвали в цикъла, не се променят.

Пример: От клавиатурата се въвежда низ, в който се проверява за наличие на определен символ и се определя позицията му в низа. Текстът завършва с ‘*’.Ако поредният въведен символ е търсеният, цикълът се прекъсва с оператор **break**.

```
#include <iostream.h>
#include <stdio.h>
void main ()
 { int i, j, r;
   char symbol;
   //   Въвеждане на символа
   symbol = getchar();
   r = getchar(); //Премахване на Enter от буфера на getchar
//   Въвеждане на текста
   j = 0; i = 1;
  while ( ( r = getchar () ) != '*' )
    if (r == symbol)
     {  j = 1; break; }
    else i++;
  if (j == 1) cout << " Символът " << symbol << " е на позиция " << i <<
endl;
      else cout << " Символът " << symbol << " не е открит!";
 }
```

Оператор continue – позволява да се завърши изпълнението на текущата итерация на цикъл **while**, **do-while** или **for**. Синтаксисът на оператора е следният:

continue;

Може да се разположи на произволно място в тялото на един цикъл. Операторът **continue** прекъсва изпълнението на текущата итерация на цикъла, а моментните стойности на променливите се запазват. За цикли **while** и **do-while** се преминава към проверка на условието за спиране, а за цикъл **for** се преминава към изчисление на израз3 за присвояване на нова стойност на управляващата променлива на цикъла. Изпълнението на цикъла продължава по стандартния начин, определен от дефиницията на цикъла.

Пример: От клавиатурата в цикъл **for** се въвеждат 10 цели числа. С друг цикъл **for** се определят четните числа и се извеждат индексите им. Ако числото е нечетно, с **continue** се прекъсва текущата итерация на цикъла и не се изпълнява операторът **cout**.

```
#include <iostream.h>
void main ()
 { int a[10], i;
   for (i = 0; i < 10; i++)
    { cout << “Въведете a[“ << i+1 << “] = “;
      cin >> a[i];
    }
  cout << “Индекси на четните числа в масива:\n”;
  for ( i = 0; i < 10; i++)
   { if (a[i] % 2 != 0) continue;
     cout << i << “ “;
   }
 }
```

Оператор goto – дава възможност да се промени реда на изпълнение на операторите в програмата. Неговият синтаксис е:
goto E;

където E е етикет, представляващ име, съставено по правилата за идентификатори и завършващо с двоеточие. Етикет може да бъде поставен пред всеки оператор, при това отделен с произволен брой интервали или разположен на нов ред.
етикет: оператор
При изпълнението на оператора **goto** E; управлението се предава на оператора с етикет E. Необходимо е операторът **goto** и етикетът E да са разположени в една и съща функция.
Препоръка: Влизането в тялото на оператори **if, while, do-while** и **for** с помощта на оператора **goto** е забранено. Обратната ситуация – излизане от тялото на оператор за управление с оператор **goto** е напълно определена. Променливите запазват стойностите, които са имали до момента на прехода.
Внимание: трябва да се внимава, ако с **goto** се навлиза в тялото на цикъл без инициализация на променливите му.
Не се препоръчва използването му – води до неясни програми, трудни за разбиране и модифициране.
Пример: реализация на цикъл **do-while** чрез **goto**;

```
do
  оператор
while (израз);

ЕТ: оператор
if (израз) goto ЕТ;
```

Оператор switch – оператор за условен преход, който позволява разклонение към повече от два клона в програмата. С него се избира една от няколко взаимно изключващи се алтернативни възможности. Операторът има следния най-често използван синтаксис:

```
switch (израз-селектор)
 { case израз1:
          оператори
          [ break; ]
   case израз2:
          оператори
```

```
        [ break; ]
   …
   [ default:
        оператори
        [ break; ]
   ]
  }
```

където израз-селектор е условие за преход. Може да бъде допустим израз със стойност **int** или **char** (целочислен).
case изразi: (i=1,2,...) – етикет, приписан към първия оператор на следващата в описанието група оператори.
case – ключова дума за означаване, че етикетът се използва в оператор **switch.**
изразi: (i=1,2,...) – цяла или символна константа или константен израз от тип **int** или **char.**
оператори – група от оператори на езика, съставящи съответния клон на програмата.

break; - оператор за прекъсване на изпълнението (не е задължителен).
default: - етикет, образуван от ключовата дума **default** и ”**:**”.
Действието на оператора **switch** е следното:

1. Изчислява се стойността на израз-селектор.
2. Стойността на израз-селектор се сравнява със стойностите на константните изрази израз1, израз2 и т.н.
3. Ако стойността на израз-селектор съвпада със стойността на някоя константа изразi (i=1,2,...), изпълнява се групата оператори, приписана към съответния етикет. С изпълнението на оператора **break;** управлението се предава на оператора, записан след затварящата скоба } на **switch.**
4. Ако няма съвпадение с нито една от константите изразi (i=1,2,...), изпълнява се групата оператори, означена с етикета **default:.** Операторът **break;** и в този случай предава управлението след оператора **switch.**

Операторът **switch** изпълнява ролята на превключвател, избиращ един от клоновете на програмата, означени в програмата със

специален етикет **case**. Връщането от разклонението се извършва с оператора **break.** Действието на оператора **break** в комбинация със **switch** е аналогично на прекъсването с негова помощ на цикли **while** и **for**.

При съставянето на оператора **switch** трябва да се спазват следните изисквания:

- не може да има константни изрази израз1, израз2, …с еднакви стойности;
- последователността, в която са подредени етикетите **case** и **default**, е напълно произволна. В какъв ред се изчисляват и сравняват стойностите на изразі зависи от компилатора. Това означава, че първия **case** може да не се проверява първи;
- броят на **case** не е ограничен, но той не може да бъде по-голям от броя на целите стойности, които приема израз-селектор.

Основната конструкция на **switch** може да се модифицира, като се има предвид следното:

1. Няколко етикета могат да бъдат свързани с една и съща група оператори. В този случай етикетите се разполагат последователно:

   ...
   case израз1:
   case израз2:
   ...
   case изразк:
   оператори;
   ...

2. Операторите **break** не са задължителни. Ако не се използва **break**, след изпълнението на избрания клон веднага се преминава към изпълнението на непосредствено следващия клон (следващия **case)** и т.н. Ако няма нито един **break**, изпълнението на **switch** завършва с достигането на затварящата скоба } .

3. Етикетът **default** не е задължителен. Ако такъв клон не е предвиден и няма съвпадение на стойността на израз-селектор с някое изразi, нищо от оператора **switch** няма да се изпълни.
4. Изход от разклонение **case** на оператора **switch** може да се осъществи освен с **break** и с оператора за безусловен преход **goto** или с оператора **return** за връщане от функция.

Да разгледаме следната конструкция:

if (израз-селектор == израз1)
 { оператори }
else if (израз-селектор == израз2)
 { оператори }
else if (израз-селектор == израз3)
…

Тази конструкция на практика функционира като оператор **switch**, но тук ние знаем наредбата на сравненията. Предимството на **switch** е по-ясната структура. Конструкцията **if else** е много по-

обща и в общия случай не може да се сведе до оператор **switch**. Условията за проверка в оператора **if else** могат да бъдат много по-сложни, отколкото в **case**.

Програма: Величината r може да е радиус на окръжност, страна на квадрат или страна на равностранен триъгълник. При зададена стойност на r се изчислява периметърът на тези фигури. Всяка фигура се кодира, както следва:

‘c’ – окръжност, ‘s’ – квадрат, ‘t’ – триъгълник.

Този код се въвежда от клавиатурата и се записва в променливата sys.

```
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h> //прототип на функцията exit
void main ()
 { char sys;
   float r, x;
   cout << “Въведете кода на фигурата: “ << endl;
   sys = getchar ();
```

```
  cout << “Въведете величината r” << endl;
  cin >> r;
  switch (sys)
  { case ‘c’:
      x = 2 * 3.14 * r;
      break;
    case ‘s’:
      x = 4 * r;
      break;
    case ‘t’:
      x = 3 * r;
      break;
    default:
      cout << “Грешен код!”;
      exit (1);
   }
 cout << “Периметърът = “ <<  x;
}
```

Функцията **exit** прекратява изпълнението на програмата в произволна функция и предава управлението на операционната система. Ако резултатът, който връща функцията **exit**, е 0, това означава липса на грешки, а ако е число, различно от нула, означава грешка. Вместо 0 и 1 могат да се използват константите **EXIT_SUCCESS** и **EXIT_FAILURE** от заглавния файл **stdlib.h**.

15. Форматиран изход. Функция printf.

Форматиран изход – преобразуване на поле от оперативната памет и извеждане на полученото върху външен носител.

Функция printf

Синтаксис:
printf (“форматиращи параметри”, списък от аргументи);

форматиращи параметри – константа от тип символи. Те са два вида:

- последователност от символи, които се извеждат на екрана. Графичните символи се изобразяват, а управляващите символи се изпълняват;
- последователност от форматиращи символи, които не се извеждат, но управляват показването на съответния

аргумент от списъка. В общия случай аргументите са произволни изрази.

Форматиращите параметри от втори тип имат следния синтаксис:

%[модификатор]форматна спецификация

В общия случай броят на форматиращите параметри е равен на броя на аргументите.

printf е прекалено усложнена функция и води до удобства за потребителя, но не и до по-бързи и ефективни програми.

Съществуват и други по-прости функции, които са свързани с точно определен входен поток от данни. За предпочитане е потребителят да използва тях, ако може да ги приложи в съответната програма.

16. Форматиран вход. Функция scanf.

Форматиран вход – от външен носител постъпва входен поток от символи. Функцията, която ги обработва задава вида на данните като елементи от този входен поток и ги преобразува в променливи, представени в оперативната памет.

Функция scanf

Синтаксис:
scanf (“форматиращи параметри”, списък от аргументи);
scanf връща резултат от тип int. Този резултат е броят на успешно въведените символи. Връща EOF, ако има грешка при въвеждането или ако е въведен символ за край на файла. Форматиращите параметри при scanf съвпадат с параметрите от втори тип на printf.

Въвежданата информация чрез scanf се присвоява на определени променливи. Затова аргументите на scanf са адресите на съответните променливи.
Адресът на една променлива се получава като пред името на променливата се постави символ &. Изключение са имената на низовите променливи, тъй като те се разглеждат като символни масиви. Името на един символен масив е адресът на първия елемент на този масив.
Информацията, която се въвежда от клавиатурата се нарича **входен поток от данни.** Този поток от данни е разбит на елементи, като разделители между елементите се използват празните символи (интервал, символ за нов ред, символ за табулация).

За scanf се отнасят същите забележки като за printf. Тя е неефективна и не се препоръчва нейната употреба, когато е възможно използването на по-прости функции.

17. Функции - общ вид, оператор return

Една програма на С представлява съвкупност от функции. Функцията е основната програмна единица в С++. Във функционално отношение тя е част от програма, с изпълнението на която се получават определени резултати. Като програмна единица тя се оформя по строго фиксирани правила и представлява самостоятелен фрагмент. Този фрагмент може да се изпълнява многократно в различни части на програмата, като към него се прави обръщение по име.

Чрез използването на функции една програма може да се раздели на части (модули) и при нейното разработване да се приложат принципите на структурното програмиране. Предимствата на този подход са следните: отделните модули могат да се програмират независимо един от друг; те много по-лесно се тестват за грешки и модифицират;

могат да се използват вече готови програми, оформени като функции; логиката на цялата програма става по-разбираема.

Синтаксисът на една функция е следният:

```
тип_на_резултата  име_на_функцията (списък_на_параметрите)
{
  тяло на функцията
}
```

Типът на една функция е типът на резултата, който тя връща. Функцията може да не връща резултат, ако на мястото на типа е записано void. Ако типът е пропуснат, той по подразбиране се смята за int.
Името на функцията е идентификатор. Главната функция има име main, това е запазена дума и не може да се използва за име на потребителска функция.
Тялото на функцията се състои от дефиниции на променливи и оператори.

Резултатът от една функция се формира с оператор **return.**
Синтаксис:
return (израз);
Изпълнение: пресмята се израза, типа на израза се преобразува към типа на функцията и се прекратява изпълнението на функцията. Ако типът на функцията е void можем да използваме return без израз за да прекратяваме изпълнението на функцията.
Ако една функция е void, но в нея няма оператор return, тогава достигането до затварящата голяма скоба на тялото на функцията е еквивалентно на изпълнението на оператор return. Скобите, които заграждат изразът не са задължителни.
Функцията може да няма формални параметри, но скобите са задължителни.
Концепция за скобите в различните версии на C:

- в Кърнингън и Ричи C ако е зададено (), това означава, че не се уточнява броя на параметрите;
- в ANSI C за да зададем функция без формални параметри между малките скоби трябва да има void;

- в C++ () задава функция без формални параметри

В тялото на една функция може да има няколко оператора return, но ще се изпълни само един или нито един тях.

Пример: функция за преобразуване на малки латински букви в големи

```
char toupper( c : char)
 { return ( (c>=’a’ && c<=’z’ )?c – ‘a’ + ‘A’:c ); }
```

Функция за намиране на n!

```
long int fact (int n)
 { long nf; nf = 1; int i;
   for (i = 2; i<=n; i++)
    nf = nf * i;
   return nf;
 }
```

18. Обръщение към функция

За да се изпълни една функция, трябва да й се предаде управлението. Това става чрез обръщение към тази функция. Функцията, която предава управлението, е **извикваща**, а тази която приема управлението е **извикана.** Една извикана функция може да извика друга функция.
Точно една функция в програмата е с име main (тя се нарича главна функция). При стартиране на изпълнението на програмата на C++ операционната система предава управлението на главната функция, която от своя страна може да се обръща към други функции. Изпълнението на програмата завършва с изпълнението на главната функция, която връща управлението на операционната система. Предвидена е възможност за излизане от програмата от всяка функция чрез обръщението към стандартната функция exit.

Резултатът от изпълнението на дадена функция може да се върне като се използва оператор return, външни променливи или параметри на функцията. Типът на връщания резултат се записва преди името на функцията.

Синтаксис на обръщението:

име_на_функцията (списък_от_фактически_параметри);

Обръщението към функция може да се използва по два различни начина:

- като операнд в израз. Резултатът, който се връща от return замества операнда. Ако обръщението към функцията е в операнд, тя обезателно трябва да връща резултат, в противен случай операндът ще има неопределена стойност;
- обръщението към функцията завършва с точка със запетая (;) – това обръщение се разглежда като оператор. Естествено е в този случай функцията да е от тип void. Ако тя не е от тип void резултатът от функцията не се използва (той се губи).

В общия случай фактическите параметри са изрази. Въпросът за съответствие между тях и формалните параметри търпи развитие. Вграденият механизъм в С е заместване на формалните с фактически параметри. Параметрите се заместват по стойност. Първо се пресмятат фактическите параметри, ако типът им не съвпада с този на формалните параметри, фактическите параметри се преобразуват към типа на формалните и стойността им се присвоява на формалните параметри. В общия случай броят на фактическите параметри трябва да е равен на броя на формалните параметри. Това не се спазва в случая, когато функцията е с променлив брой параметри. Въпросът е дали типовете на фактическите параметри съвпадат с тези на формалните.
В Кърнингън и Ричи С се изисква съвпадение.

В ANSI C, C++ няма такова изискване, т.е. типа на фактическия параметър може да се различава от типа на формалния параметър. В такъв случай стойността на фактическия параметър се преобразува към типа на формалния параметър.
Реализация на обръщението – използва се програмен стек, разположен в частта от оперативната памет, разпределена за програмата:

- пресмятат се фактическите параметри. Проверява се дали типа на съответните формални параметри съвпада с типа на фактическите параметри. Ако някъде няма съотвествие се извършва преобразуване на стойността на фактическия параметър. В С фактическите параметри се пресмятат от дясно на ляво (за разлика от Pascal, където е от ляво на дясно) и стойностите им една по една се записват в стека. Повечето компилатори имат опция с която се задава реда на пресмятане на фактическите параметри;
- след това се предава управлението на функцията и се разпределя памет за формалните параметри в стека. При

това за всеки формален параметър се разпределя памет точно там, където е записана стойността на съответния фактически параметър, която трябва да му се присвои.

Ако променяме стойностите на формалните параметри, тогава стойностите на съответните фактически параметри не се променят. Ако искаме да променяме един фактически параметър, трябва в обръщението към функцията да зададем неговия адрес.

Примери:

```
#include <iostream.h>
#include <stdio.h>
char toupper(char c)
 { return ( (c>='a' && c<='z' )?c – 'a' + 'A':c ); }
void main ()
 { char a, b;   int br = 0;
   cout << "Въведи символи: " << endl;
   do {
     a = getchar();     b = toupper (a);
     if (br==0)
       cout << "Преобразуваната последователност от символи: " <<
endl;
     cout << b;
     br++; }
  while (b != '\n');
  br--;
  cout << "Брой обработени символи: " << br << endl;
 }
```

```
#include <iostream.h>
long int fact (int n)
 { long nf; nf = 1; int i;
   for (i = 2; i<=n; i++)
    nf = nf * i;
   return nf;
 }

void main ()
 { int x;
   cout << “Въведи x = “;
   cin >> x;
   cout << “x! = “ << fact(x) << endl;
 }
```

19. Прототипи на функция

Ако извикваната функция е разположена в текста на програмата след извикващата, тя трябва да бъде подходящо декларирана предварително. Това предварително обявяване се нарича прототип на функция, който информира компилатора за типа на връщаната стойност, броя и типа на формалните параметри. Използвайки прототипа, компилаторът извършва контрол за съответствието по брой и тип между формалните и фактическите параметри и при необходимост извършва необходимите преобразувания.
Прототипите на една функция са нужни за да може правилно да стане заместването на формалните с фактическите параметри. Когато транслаторът реализира обръщение към функция, той трябва предварително да има информация за броя и типовете на формалните параметри за да знае дали стойностите на съответните фактически параметри трябва да бъдат преобразувани и към какъв тип.

Освен това, той трябва да знае какъв е типа на връщания резултат, за да може да прецени дали този тип трябва да се преобразува.
В програмите по-горе, компилаторът научава това за функцията преди да има обръщение към нея. Ако обаче разменим местата на двете функции, компилаторът ще даде съобщение за грешка, тъй като той попада на обръщението, но няма необходимата информация.
Именно затова в този случай трябва да запишем **прототипа** на функцията преди обръщението към нея (прието е, прототипите на функциите да се поставят непосредствено след директивите към препроцесора).
Синтаксис:
тип_на_функция име_на_функция (списък от формални параметри)**;**
; показва, че е прототип.
В прототипа на една функция можем да пропуснем имената на формалните параметри и да оставим само имената на типовете.

Примери:
```
char toupper (char c);
long fact (int);
```

Един заглавен файл съдържа (не само) прототипи на функции. По същата причина, за да се обръщаме към стандартна функция, след като не сме я описали, трябва да зададем нейния прототип, който е записан в заглавния файл. Затова го включваме в началото на програмата.

20. Области на действие на променливите

В тялото на една функция могат да бъдат дефинирани **вътрешни** променливи. Те са определени само в тялото на конкретната функция от мястото на дефинирането им до края на тялото на тази функция и това е тяхната **област на действие** (или област на видимост).
В С задължително дефинирането трябва да е в началото на тялото на функцията.
В С++ това може да става на произволно място, където може да има израз. Това изисква и по-голямо внимание при определяне на тяхната област на видимост.
Областта на видимост започва от мястото, където е дефинирана променливата и продължава до края на функцията.
Една програма на С е съвкупност от функции. Между тях и преди тях може да се дефинират променливи.

Те се наричат **външни** и са определени само за тези функции, които са записани след тяхното дефиниране в първичния файл. С помощта на външните променливи може да се обменят данни между функциите. Един друг начин за това е механизмът на формалните и фактическите параметри. Той е основен и е за предпочитане, тъй като с него се постигат по-ясни и разбираеми програми, въпреки че използването на външни променливи е по-ефективно, тъй като не изисква заместване. Имената на външните променливи трябва да са уникални в програмата. Ако в дадена функция се среща вътрешна променлива с име, което съвпада с това на външна променлива, то във функцията може да се използва само вътрешната променлива.

Пример:
float x, y;
void main ()
{ int k; … може да се използват външните променливи x, y и вътрешната променлива к}
void fun1 ()
{ int x; … от мястото на дефинирането до края на функцията може да се използва вътрешната променлива x (скрива външната променлива x) и използва външната променлива y }
float z;
void fun2 ()
{ int y; … тук y скрива външната променлива y, може да се използва вътрешната y и външните променливи x, z }

Външните променливи x и y имат област на действие цялата програма (въпреки че не могат да се ползват навсякъде). Външната променлива z има област на действие само последната функция.

Във функцията main могат да се използват външните променливи x, y и вътрешната променлива k. Областта на действие на k свършва в края на функцията.
Във функцията fun1 могат да се използват външната променлива y и вътрешната променлива x, която скрива външната променлива x.
Във функцията fun2 могат да се използват външните променливи x и z и вътрешната променлива y, която скрива външната променлива y.

Конструкцията **блок** има специално отношение към областта на действие на променливите. Блок е последователност от описания и оператори, заключени в големи скоби ({ }). Тялото на една функция е блок, но в тялото на функцията може да има други блокове. Те се наричат **вложени.** Самият блок може да се запише навсякъде, където може да се запише оператор. В блоковете могат да се дефинират променливи.
В C обезателно това трябва да става в началото на блока.
В C++ това може да става на произволно място в блока.

Променливите, които са дефинирани в даден блок са **локални** за този блок и **глобални** по отношение на всички вложени блокове в блока. Възможно е съвпадане на имена между локални и глобални променливи. В такъв случай името на локалната променлива скрива името на глобалната променлива.
Вътрешните променливи се делят на глобални и локални по отношение на вложените блокове във функцията. Външните променливи са един тип - те са глобални по отношение на функциите в програмата.
Пример:

```
void main ()
 { float x, y;
   { int k; … }
   { int x; … }
   { float z;
     { int y; …}
    }
  }
```

Променливите x и y са локални за функцията main, въпреки че не могат да се използват навсякъде.
Променливите x и y са глобални за първи блок, а променливата k е локална за него.
Променливата y е глобална за втори блок, а променливата x е локална за него и тя скрива глобалната променлива x.
В третия блок z е локална променлива, а x и y са глобални.
В четвъртия блок, който е вложен на третия, могат да се използват глобалните променливи x и z и локалната променлива y, която скрива глобалната променлива y.
Ако имаме малък блок, може да се дефинират променливи преди него. При големи блокове дефинирането на променливите е по-добре да е по-близо до мястото на тяхното използване.
Два блока могат да бъдат или вложени или паралелни, т.е. или изцяло са вложени или нямат сечение.

21. Класове памет

Всяка променлива се характеризира с три атрибута – тип, област на действие и период на активност.
Типът на променливата определя колко байта памет заема тя и по какъв начин тя се кодира в тези байтове. Вследствие на това получаваме множеството от допустими стойности на променливата.
Областта на действие се определя от точното място, където е дефинирана променливата.
Под **период на активност** се разбира частта от времето при изпълнението на програмата, през която се съхранява последната текуща стойност на променливата. Периодът на активност е равен или на времето за изпълнение на цялата програма или на времето, през което управлението се намира в областта на нейното действие.

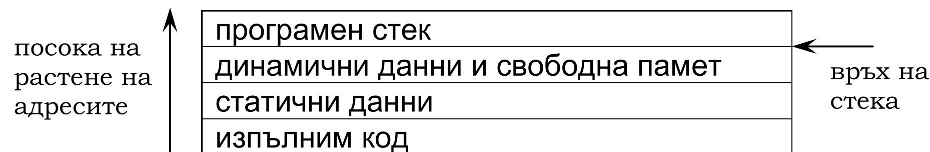
В зависимост от периода на активност за променливите се разпределя памет на различни места или още се казва, че променливите имат различен **клас памет**.
В езика С се различават четири класа памет:

- extern
- auto
- static
- register

Класът памет се задава в дефиницията на променливата.
Синтаксис:
клас_памет тип име

Тук клас_памет може да бъде auto, static, register.

Структура на частта от паметта, която е разпределена за една потребителска програма на машинен език.



Програмният стек заема и освобождава памет – ако влизат или излизат данни в него върхът се измества.
Областта за статични данни се разпределя в началото на изпълнението на програмата и се освобождава в края на изпълнението на програмата.

В зависимост от това какъв клас памет е една променлива за нея се разпределя памет в различна област.

Клас памет extern не се посочва явно при дефинирането на променливите. По подразбиране клас памет extern имат външните променливи. За тях памет се разпределя в областта за статични данни. Стойността се пази през цялото време на програмата, паметта се освобождава когато завърши програмата.

С ключовата дума extern се обявява, че една външна променлива вече е дефинирана в друг модул и заради това външна променлива не се дефинира, а декларира.

Клас памет auto е предназначен за вътрешните променливи и за формалните параметри на функциите. Променливите от този клас се наричат **автоматични.** За тях се разпределя памет в програмния стек и това става по време на изпълнение на програмата след влизане във функцията или блока, в който те са дефинирани и когато управлението за първи път достигне точното място на дефинирането. Поради това в началото автоматичните променливи имат неопределена стойност. Техният период на активност съвпада с областта им на действие, т.е. стойностите им се пазят само докато се изпълнява функцията или блока, в който са дефинирани. При повторно влизане в блока или функцията, стойностите им отново са неопределени. По подразбиране при дефинирането на една вътрешна променлива тя е от клас auto (дори когато не е указано явно).

Клас памет static трябва явно да се задава при дефиницията на една променлива. За променливи от този клас се разпределя памет в областта за статичните данни. Ако една вътрешна променлива е дефинирана от клас static, нейният период на активност става времето за изпълнение на цялата програма. Повечето компилатори нулират областта за статичните данни в началото на програмата. Това не се отнася за програмния стек – областта на автоматичните променливи и затова тяхната стойност е неопределена в момента, когато за тях се разпредели памет. Ако една външна променлива е дефинирана със static смисълът е друг – променливата в случай на разделно компилиране е достъпна само във файла, в който е дефинирана и това означава, че тя не може да се използва в други модули.

```
void f()
{static int i; // i е в областта за статични данни
 float z; // клас auto, памет в програмния стек
…………
 i=i+2;
 z=2;
 …………
void main()  // със завършването на изпълнението на f паметта за z се освобождава
.......
f()
.........
```

При първото изпълнение на функцията f променливата i ще получи стойност 0 след разпределянето на памет в областта за статичните данни и след това ще получи стойност 2. След излизане от функцията паметта за i няма да се освободи (тя е static), макар че ние не можем да я използваме извън функцията, тъй като тя е вътрешна променлива.

След изпълнението на f и връщането в извикващата функция тази стойност на i се запазва. При повторно обръщение към f променливата i ще получи нова стойност 4.
За променливата z се разпределя памет в програмния стек и при изпълнението на функцията тя получава неопределена стойност. Непосредствено преди края на изпълнението тя има стойност 2, но когато изпълнението приключи, паметта за z се освобождава (тя е в стека). При повторно обръщение за z отново ще се разпредели памет в стека (в общия случай на друго място) и стойността и отново ще е неопределена.

Клас памет register е частен случай на клас памет auto. Ако в програмата има променлива, която се използва много често (например в даден цикъл), възможно е за нея да се разпредели по-бърза памет (регистър на централния процесор) и по този начин се засилва ефективността на програмата. Ако няма свободен регистър тя се разглежда като променлива от клас auto.

Оптимизиращите компилатори правят това автоматично – те сами определят кои променливи се използват често и разпределят памет в регистрите на процесора.

Когато в дефинирането на една променлива сме записали само класа памет, но сме пропуснали типа, тогава той се подразбира като int.
Например: static i, auto i и т.н.
Въвеждането на класове памет в езика позволява да се управлява паметта, което води до икономично използване на машинните ресурси и в редица случаи до намаляване на времето за изпълнение на програмата.

22. Разделно компилиране

Разделното компилиране е възможност за самостоятелно компилиране на отделни първични файлове, които образуват една програма. Те се наричат **модули** и могат да се редактират и компилират по общоприетите правила. На етапа свързване, обектните файлове на всички модули се обединяват в един общ изпълним файл на програмата.

За всички модули главната функция е точно една. Функциите в различните модули могат да обменят информация по два начина:

- чрез дефиниране на външни променливи, които са общи за модулите;
- чрез обръщение към функции, които се съдържат в други модули.

За да се направи една външна променлива, дефинирана в един модул, обща и за други модули тя трябва да бъде повторно описана като се **декларира** в тях по следния начин:

Синтаксис:

extern име_на_тип име_на_променлива;

Ключовата дума extern трябва задължително да присъства. Това описание информира компилатора, че променливата е дефинирана в друг модул. Това повторно описание се нарича декларирane, и то е различно от дефинирането. Докато при дефинирането се разпределя памет от компилатора за променливата, при декларирането това не се прави, но свързващият редактор разпределя за нея същата памет, която е разпределена за съответната дефинирана променлива.

Инструкциите към компилатора за разделно компилиране зависят от средата: това става в проектен файл, където са записани имената на първичните файлове.
Това че една променлива е обща за няколко модула, означава, че всеки модул има достъп до нея и тя може да бъде променяна във всеки модул.

Ако декларирането на една променлива като extern е извън функциите в модула, то важи за всички функции след декларацията. Една променлива може да бъде декларирана като extern във функция или блок и тогава нейната област на действие започва от декларацията и свършва в края на функцията или блока.

Когато един масив се посочва като extern, не е задължително да се задава първата размерност. Другите размерности не могат да се пропускат, за да може по индексите на елементите еднозначно да се попадне в паметта, разпределена за съответния елемент от масива.
Пример:

```
extern float a[ ];
extern int b [ ][10];
```

Когато срещне ключовата дума extern, свързващият редактор преглежда най-напред дефинициите в текущия модул. Ако не намери съответната външна променлива в този модул, свързващият редактор търси съответствие в другите модули на програмата. Ако не намери подходяща дефиниция и там, тогава търсенето продължава в указаните или приетите по подразбиране библиотеки.

Пример с два модула:

file1.cpp

```
#include <stdio.h>
unsigned array[20];
void main ()
  { extern double x;
    int a;
    a = 5;
    array[0] = 25;
    x = 1.5;
   …
  }
char ch;
void fun1 ()
 { int a;
   a = 8;
   ch = ‘K’;
```

file2.cpp

```
#include <stdio.h>
extern unsigned array[];
void fun3()
  { extern char ch;
    extern double x;
    array[3] = 100;
    ch = ‘F’;
    x = 3.14;
   …
  }
 void fun4 ()
  {  int a;
     extern char ch;
     ch = ‘I’;
     array[4] = 125;
```

```
   array[1] = 50;
   …
 }
double x;
void fun2 ()
 { int a;
   a = 10;
   ch = 'A';
   array[2] = 75;
   x = 2.5;
   …
  }
```

```
…
}
```

Първи модул:
областта на действие на външния масив array е всички функции в модула.
Функция main:
деклариране на променливата x – свързващият редактор търси външна променлива със същото име и я открива – тя има област на действие функцията fun2 и за нея памет е разпределил компилатора. Свързващият редактор разпределя за нея същата памет като тази за намерената външна променлива x. Областта на действие на декларираната x е функцията main.
Променливите a са различни и са локални за функциите, в които са дефинирани. Те не могат да се използват в други функции.
Променливата ch е външна и има област на действие функциите fun1 и fun2.

Втори модул:
деклариране на масив array – свързващият редактор търси дефиницията на външен масив със същото име – най-напред в същия модул, след това в другите модули и го намира. След това разпределя за него същата памет, която компилатора е разпределил в първия модул. Декларираният array има област на действие целия втори модул.
Функция fun3:
деклариране на ch, която е вътрешна за fun3 - свързващият редактор я намира в първи модул. Аналогично за декларирането на променливата x.
Функция fun4:
деклариране на ch, която е вътрешна за fun4 – свързващият редактор я намира в първи модул. Дефиниране на локална променлива a, която може да се използва само във fun4.

По подразбиране имената на функциите са външни, т.е. в един модул можем да се обръщаме към функция, която е дефинирана в друг модул.

В този случай, обаче, трябва да се опише прототипа на извикваната функция, като в началото на това описание се поставя ключовата дума extern. В C++ тя може да се пропусне. Въпросът стои по същия начин и за стандартните функции, които се разглеждат като дефинирани в друг модул.
За удобство, прототипите на потребителските функции могат да се оформят в потребителски заглавен файл, който да се включва в модула.
Ако една функция (или променлива) е дефинирана като static, то към нея може да има обръщение само в модула, в който тя е дефинирана. Това се използва в случаите, когато желаем да гарантираме използването на променливата или функцията само в един файл.
Променлива, която в някой модул е декларирана като extern, може да се инициализира само във файла, където е дефинирана като външна променлива.

Пример:

modul1.cpp

```
#include <iostream.h>
int mas[10];
extern int max (int);
void main ()
 { int br, i;
   cout << “Въведете броя на числата: “;
   cin >> br;
   for (i = 0; i < br; i++)
    { cout << “Въведете mas[“ << i << “]: “;
      cin >> mas[i];
    }
   cout << “Максималният елемент е “ << max(br);
 }
```

modul2.cpp

```
extern int mas[];
int max (int size)
 { int comp; int i;
   comp = mas[0];
   for (i=0; i < br; i++)
    if (mas[i]>comp)
      comp = mas[i];
   return comp;
  }
```

23. Инициализиране на променливите

Под инициализиране на променливите се разбира присвояване на начални стойности на променливите.
Някои от променливите получават автоматично нулеви стойности в началото на изпълнението на програмата – такива са например тези, за които памет се разпределя в областта за статичните данни (клас памет static или extern). Другите променливи (например клас памет auto, register) не получават нулева начална стойност.
Всяка променлива може да бъде инициализирана при нейното дефиниране.
Синтаксис:
име_на_тип име_на_променлива = израз;

Пример:
float pi = 3.14;

Този запис е еквивалентен на следния:
```
float pi;
pi = 3.14;
```

Ако една променлива има клас памет static или extern, т.е. за нея да е разпределена памет в областта на статичните данни, тогава изразът задължително трябва да е константен. Той не може да съдържа променливи и трябва да осигурява числена стойност от подходящия за променливата тип. Това се налага, тъй като по време на компилация за тях трябва да е известна началната стойност.

Пример:
```
static int x = 230*22;
static float  pi = 22.0/7.0;
static int i = 1, j = 10;
```
На последния ред едновременно са дефинирани и инициализирани няколко променливи.

Променливите с клас памет static или extern запазват своите стойности, докато не се променят явно с израз за присвояване. Например, локалните (вътрешните) статични променливи се инициализират само един път преди първото изпълнение на съответната функция, където са дефинирани, и запазват последната текуща стойност при всяко следващо влизане във функцията. Ако е необходимо такава променлива да се инициализира отново при всяко влизане във функцията, това трябва да се извърши чрез израз за присвояване.

За променливи с клас памет auto или register (за променливите се разпределя памет в програмния стек) инициализиращият израз е произволен – той може да съдържа константи, променливи, формални параметри или обръщения към функции, като също трябва да осигурява числена стойност от подходящия за променливата тип.

Пример: дефиницията на функция

```
void f1(int x, int y)
{int i = 1;
 int j = x*y;
………….
}
```

е еквивалентна на дефиницията

```
void f1(int x, int y)
{int i;
int j;
i=1;
j=x*y;
………….
}
```

Автоматичните променливи могат да се инициализират както при дефинирането им, така и чрез изрази за присвояване. За предпочитане е инициализиране с израз за присвояване, тъй като текста на програмата е по-разбираем. Инициализацията при дефинирането е по-незабележима и е далеч от мястото на използването на променливата.
Променливите от клас auto или register не запазват своите стойности след излизането от функцията. Те съдържат произволни стойности в началото на изпълнението на функцията и е грешно да се използват преди да са инициализирани. Програмистът трябва да се погрижи за това. Зададената инициализираща стойност се изчислява и присвоява на автоматичните и регистровите променливи всеки път, когато се извика функцията.

```
int z;
void fun1 (int x, int y)
 { int i = x*y*z;
   …
 }

void fun ()
 { static int i = 8;
   int j = 4;
   …
   i = i + 2;
   j = j + 2;
 }
```

Инициализиране на локални статични променливи – извършва се точно веднъж в началото от компилатора (преди да се влезе във функцията).

Например в горната програма след първото извикване променливата i става 10, след второто извикване - 12 (тя не се инициализира отново) и т.н.
Инициализиране на локални променливи – когато влезем във функцията се разпределя памет в програмния стек и там се записва началната стойност (например 4 за j). В края на първото извикване j си променя стойността на 6, а след приключване на изпълнението на функцията паметта за j се освобождава – ето защо при повторно изпълнение, за разлика от локалните статичните променливи, j отново ще стане 4, след това 6, накрая паметта за j ще се освободи и т.н.

Въпросът за несъответствие между типа на израза:

- в K&R C не може да има такова несъответствие.
- в ANSI C и C++ типа на израза се преобразува към типа на променливата.

Инициализиране на масиви – при дефинирането това става чрез задаване на списък от началните стойности за елементите на масива.
Списъкът се поставя в големи скоби.
Пример:
int list[5] = { 1, 2, 3, 4, 5 };

Подреждането на стойностите съответства на реда на разпределение на елементите на масива в оперативната памет.
int mat[3][3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9};

Ако началните стойности са по-малко от броя на елементите на масива, останалите се запълват с нули. Явното нулиране се препоръчва като добър стил на програмиране. Ако началните стойности са повече, компилаторът дава съобщение за грешка.
Ако масивите са многомерни може да се използва влагане на големи скоби.
Пример:

int mat[3][3] = { { 1, 2, 3}, { 4, 5 } };

Инициализиране на низова променлива (символен масив) – може да стане не само по общото правило чрез списък от символи, но и с низова константа. Използва се еквивалентността по отношение на записа на низовите константи като символни масиви в паметта.

Примери:
Инициализациите при дефиниране
char name[6] = { ‘P’, ‘e’, ‘t’, ‘a’, ‘r’, ‘\0’ };
char name[6] = “Petar”;
са еквивалентни. Нулевият байт в първия вариант е задължителен.

Във втория случай автоматично се добавя нулевия байт в края на масива. Ако броят на символите е по-малък от размерността на масива, останалите елементи се запълват с нулеви байтове.
Ако е по-голям – излишните символи се игнорират, включително и нулевия байт.

При инициализирането на масиви по време на дефинирането може да не се задава броя на елементите на масива. Задават се само празни средни скоби [ ], а броят на елементите му се определя от компилатора като равен на броя на инициализиращите елементи.
Пример:

```
int list[] = { 1, 2, 3, 4, 5 };
char name[] = "Petar";
```

При първия случай броя се определя от компилатора като равен на броя на инициализиращите елементи – тук 5. При втория случай (масив от тип char) размерността е броя на символите + 1, тъй като е добавен нулев байт.

Ако при дефиниране на един едномерен масив, той е инициализиран, тогава може да се пропусне първата размерност. Ако масивът е от char размерността е броя на символите + 1, тъй като е добавен нулев байт. Ако масивът е многомерен може да се пропусне само първата размерност.

В ANSI C и C++ може да се използва ключовата дума const. Една променлива от тип const задължително трябва да бъде инициализирана при нейното дефиниране и тя не може да променя стойността си по време на изпълнение.
Примери:

```
const int initial = 100;
const double epsilon = 0.001;
const int masiv [ ] = { 1, 2, 3 };
void fun (const int a)
 { … … }
```

Тук формалният параметър е const и не се допуска той да бъде променян в тялото на функцията fun.

В C, C++ се следва принципът за най-малки привилегии – за да може да се подпомогне по-лесната проверка на програмата. Принципът гласи, че ако знаем, че нещо в програмата със сигурност не трябва да се случи, това трябва да се каже на компилатора.

Една външна променлива може да се инициализира само при дефиниране, но не и при деклариране.

Ако имаме външна променлива от тип const, която искаме да се ползва във всички модули, трябва да се постави extern пред const в дефиницията.
Пример:
дефиниция – extern const int initial = 5;
декларация – extern const int initial;

24. Буфериран вход и изход. Функции getchar и putchar

Функцията getchar се използва за въвеждане на един символ от стандартното входно устройство (клавиатурата). Функцията putchar се използва за показване на един символ на стандартното изходно устройство (екрана).
При много от функциите за вход и изход в С се използва буфер (един за вход и един за изход).
Специалният файл, който е свързан с въвеждането от клавиатурата е stdin – стандартен вход. Този файл е свързан със стандартния входен поток от данни и има собствен буфер.
Специалният файл, който е свързан с извеждането на екрана е stdout – стандартен изход. Този файл е свързан със стандартния изходен поток от данни, асоцииран с монитора и има собствен буфер.

Функция getchar (от англ. GET CHARacter – приемане на символ). Форматът й е:

```
int c;
c = getchar();
```

където в променливата c се записва ASCII-кодът на въвеждания символ, прочетен от буфера на клавиатурата, или при грешка c е равно на -1.

getchar е от тип int, защото при грешка тя трябва да върне стойност, която е различима от символите (в случая -1).

Всеки символ, въведен от клавиатурата, се записва последователно в буфер. Функцията извлича поредния символ от буфера, като спазва принципа „пръв въведен – пръв прочетен”.

При достигане на getchar изпълнението на програмата спира. Това може да се използва за паузи в изпълнението на програмата.

Изпълнението на getchar започва след натискането на <CR> (клавиш Return или Enter). Преди натискането на <CR> се въвежда символът, който ще се прочете от getchar.

Това може да е всеки от въвежданите чрез клавиатурата символи (вкл. интервал). Ако преди <CR> няма такъв символ, getchar приема самия символ <CR>. При всяко следващо обръщение към функцията се приема следващия в буфера символ. Ако при първото обръщение въведем един символ и <CR>, с което завършваме въвеждането и активираме getchar, функцията приема първия въведен символ и изпълнението на програмата продължава. Не трябва да забравяме, че символът <CR> е останал в буфера и при следващо обръщение към getchar ще се приеме от нея и изпълнението отново ще продължи. В повечето случаи буферът трябва да се изчисти от излишния <CR>.

Функцията putchar (от англ. PUT CHARacter – постави (изведи) символ). Тази функция е обратна по действие на getchar и показва на стандартното изходно устройство (екрана) символа, чиито ASCII-код е зададен като неин аргумент.

Форматът на putchar е:

```
char c;
.............
putchar (c);
```

където в c е ASCII-кодът на извеждания на екрана символ. putchar връща резултат от тип int. Функцията се изпълнява веднага след обръщението към нея.
putchar изпраща въведените символи на stdout и те се извеждат на екрана.
Аргументът се записва в буфера на stdout и се връща от функцията. Ако има грешка при извикването putchar връща –1.

getchar и putchar са функции, но това не е вярно за всички компилатори – много пъти те са макроси.

25. Указатели. Основни операции

Всеки обект на езика (променлива, константа, елемент на масив и т.н.) се съхранява в определени клетки от паметта на компютъра или, както е прието да се казва, на определен адрес. Променлива, чиято стойност е адрес от паметта на компютъра, се нарича указател. Стойността на указателя посочва (указва) местоположението на дадена променлива и позволява косвен достъп до стойността на тази променлива, за разлика от директния достъп с помощта на нейното име, който използвахме досега.

Указателят е променлива, чиято стойност е адрес от паметта. Както всяка променлива в езика С и указателят трябва да се дефинира, преди да бъде използван.
Синтаксис:
име_на_тип *име_на_указател;
като име_на_тип и име_на_указател могат да бъдат всеки валиден тип данни и идентификатор.

Например, за да дефинираме указател към променлива от тип int, трябва да запишем int *p; където p е името на указателя. Указателите са елементи от езика, които са характерни за асемблерните езици. Те позволяват да се работи с адресите на променливите, като не е нужно да се слиза на най-ниското машинно ниво, както е при асемблерните езици. Дефинираният тип за указател трябва да съответства на типа на обекта, към който сочи указателя. Ако това изискване се наруши, компилаторът дава предупреждение. Възможно е чрез явно преобразуване на типовете да се укаже на компилатора, че това се налага и в такъв случай той няма да даде предупреждение. Указателите могат да бъдат от всички валидни за езика типове данни.

Примери:

```
int *p;
 float *p1;
 double z, *p2;
```

На последния ред е обединена дефиниция на указател с дефиниция на променлива.

След като един указател е дефиниран, той съдържа произволна стойност и може да бъде използват след като бъде инициализиран. Единствените правилни стойности, които указателя може да приеме, са валидни адреси от паметта или 0. Примери:

```
p = 0; p = NULL;
p = (int *) 1507;
```

Нулата (или константа NULL от stdio.h) означава **нулев адрес**. p = NULL; е специален случай на присвояване на цяло число на указател, което е единственото смислено присвояване на цяло число. Паметта с нулев адрес се ползва от операционната система. Инициализирането на указател с NULL се използва за указване, че указателят е свободен и не посочва нищо. Нулевият адрес има и други приложения – може да се използва като знак за край или признак за грешка и др.

Изразът (int *) 1507 представлява абсолютен адрес в паметта с номер 1507.

Има две основни едноместни операции за указатели.

- &операнд – определя се адресът на операнда. Уточнение – всеки байт в паметта си има адрес. Ако операндът съдържа повече от един байт, тогава адресът му е адресът на най-младшия байт (този който има най-малък адрес). Операндът може да е променлива или елемент от масив. Неправилно е да се използва операцията & за определяне на адрес на променлива от клас register.
- *операнд – в общият случай операндът е израз, чиято стойност е адрес. Тази операция определя стойността, която е записана на адреса, който е резултат от пресмятането на операнда. В случай, че операндът на * е адрес на променлива, тогава резултатът е променливата, записана в този адрес.

И двете операции имат приоритет 2 и са дясно-асоциативни.

Примери:

```
float x, y, *px, *py;
x = 3.14; // на x се присвоява 3.14
px = &x; // на px се присвоява адреса на x.
y = *px + 10; // на y се присвоява x + 10.
py = px; // на py се присвоява адреса на x.
*py = 1; // на x се присвоява 1.
```

*px – осигурява косвен достъп към x.
x – осигурява пряк достъп към x.

```
float x ,y, *px, *px1;
px = &x; // адресът на променливата x
y = *px;  // y = x
px1 = &*px;  // px1 = px
y = *&x;  // y = x
```

Последните два оператора илюстрират, че операциите & и * са взаимно неутрализиращи се.

```
float x, *px;
unsigned z;
x = 3.14; px = &x;
z = px; //компилаторът дава съобщение за грешка при
преобразуване.
z = (unsigned) px; //компилаторът не дава предупреждение.
```

Във втория случай на z ще се присвои стойността на адреса на променливата x.

```
float y, x, *px;
x = 3.14; px = &x;
y = ++*px; //и двете операции имат приоритет 2 и са дясно-
             // асоциативни.
```

След изпълнението: x = 4.14; y = 4.14;

```
float y, x, *px;
x = 3.14; px = &x;
y = (*px)--; //и двете операции имат приоритет 2 и са дясно-
                // асоциативни, затова поставяме скоби.
```

След изпълнението: x = 2.14; y = 3.14;

```
float x, y;
int *px, *py;
px = &x; //съобщение за грешка от компилатора, тъй като се
разминават
        // типовете на указателя и на променливата.
y = *px; //предупреждение за грешка от преобразуване и възможна
загуба на данни.
```

Указател от тип void.

На указател от тип void може да бъде присвоена стойността на указатели от произволен тип. В ANSI C и обратното е валидно. В C++ в този случай е необходимо явно преобразуване.

```
void *gp;
int *ip;
gp = ip; // това е възможно и в C, и в C++
ip = gp; // възможно само в ANSI C. Съобщение за грешка от
преобразуване във
          // Visual Studio
ip = (int *)gp; //явно преобразуване
```

Често допускани грешки:

```
float x, y, *py;
*py = x; // преди да се използва един указател, той трябва да сочи
       // към някакъв адрес.
py = y; // на указателя се присвоява променливата от тип float,
вместо адреса на
     //  променливата.
```

const int initial = 100;
int *ptr;
ptr = &initial; - това не се позволява, тъй като ако е разрешено, по косвен начин ще можем да променяме именувана константа.

Можем да дефинираме указател към константа по следния начин:
const int *ptr;
ptr = &initial;
Това е вярно, тъй като не се позволява да се промени *ptr.
Адресът на променлива от тип const може да се присвои само на указател, който сочи към константа. Указател към константа може да получи други стойности, но променливите, извлечени чрез него, не може да бъдат променяни.
int k = 1;
ptr = &k; - възможно е, но чрез *ptr не може да се промени стойността на k.
*ptr = 3; грешка, *ptr сочи константен обект;.

Можем да дефинираме указател от тип const.
int *const ptr1 = &k;
В този случай можем да променяме *ptr1, което е променливата k, но не можем да променяме адресът, към който сочи ptr1.
*ptr1 = 3; е възможно, променливата, към която сочи ptr1, не е константа и може да се променя, но
int i;
ptr1 = &i; не е възможно, защото ptr1 сочи константен обект, указател от тип const не може да се променя.

Можем да дефинираме указател, който е константа и сочи към константа.
const int *const ptr3 = &initial;
Тогава ptr3 не може да променя адреса, към който сочи, което е константата initial (например ptr3 = &i; е невъзможно), нито може да се променя *ptr3 (*ptr3 = 3; също е невъзможно).

26. Адресна аритметика

Силна страна на езика С са предвидените за ефективно използване на указателите аритметични действия с тях. Същността на указателите като особен вид променливи определя и кои от аритметичните действия са позволени с тях. При адресната аритметика са валидни единствено операциите събиране и изваждане. При това има изисквания и към операндите на тези операции.

Унарните операции ++ и -- са напълно валидни за указателите. Най-същественото е, че те притежават необходимата „интелигентност" и увеличението (намалението) с 1, в действителност е увеличение (намаление) с толкова байта, колкото заема в паметта обектът, към който сочи указателят. С други думи, автоматично се изчислява необходимият брой байтове, които се добавят (изваждат) в зависимост съответно от типа на данните и размера им в байтове.

Пример:

```
int x, *px;
px = &x;
cout << px; //0x0012FF7C
```

cout << ++px;//0x0012FF80 - адресът, към който сочи px се увеличава с четири байта и px сочи към следващите четири байта. За компютри, при които типовете на данните са с други размери, съответният компилатор автоматично изчислява необходимия брой единици за добавяне или изваждане, за да определи правилния съседен адрес. Това става автоматично и програмистът не трябва да се грижи и да изчислява правилните адреси. Същото е в сила и при използването на операциите + и -, както и на специалните съкратени операции += и -=. Най-важното е, че само събиране и изваждане на указател с цяло число са валидни аритметични адресни операции.

Пример:

```
int x, *px, *py;
px = &x;
```

px = px + 1; //също както и при ++px - адресът, към който сочи px се увеличава с четири байта и px сочи към следващите четири байта
px = px + 4; //адресът, към който сочи px, се увеличава с 16 байта
px += 4; //също както px = px + 4;
px -= 9; //също както px = px - 9;
py = --px; //py = px – 1; на py се присвоява стойността на px, намалена с четири байта
При добавянето (изваждането) на цяло число към (от) указател, всяка единица е равна на броя на байтовете, отделени в паметта за дадения тип на елемента от данните, към който сочи указателят. Например след изпълнението на px -= 9; указателят px ще съдържа стойността на деветия адрес (на обекти от тип int) преди текущата му стойност, т.е. в действителност се извършва действието нов адрес = стар адрес – 36 (9 x 4 байта за всеки елемент данни от тип int).
Операциите ++ и – са едноместни операции и могат да се комбинират с други операции. Например, ако p е валиден указател, можем да запишем

*p++ //първо се извлича стойността, сочена от p, а после се увеличава указателят p да сочи към следващия адрес
*p-- // първо се извлича стойността сочена от p, а после се намалява указателят p да сочи към предишния адрес
*++p // първо се увеличава указателят p да сочи към следващия адрес и след това се извлича стойността, сочена от p
*--p //първо се намалява указателят p да сочи към предишния адрес и след това се извлича стойността, сочена от p
Тук се използва фактът, че унарните операции *, ++ и -- са с еднакъв приоритет и се изпълняват отдясно наляво.
Присвояването на указател на указател, ако те сочат към еднотипни обекти, е допустимо.
Аритметичните операции – събиране на указатели едни с други, събиране или изваждане на указатели с числа от тип float или double, умножение или деление на указатели, са невалидни аритметични адресни операции.

В случаите, когато два или повече указателя сочат към променливи, между които съществува връзка (например различните елементи на масив), такива указатели могат да бъдат изваждани един от друг. Например, ако p1 и p2 са указатели към елементи от един и същ масив, разликата им p1 - p2 ще определи броя на елементите между двата елемента, които сочат p1 и p2. За подобни указатели към взаимносвързани елементи могат да бъдат използвани операциите за отношение и логическите операции. Могат да се сравняват указатели. Отношението px < py е TRUE, ако px сочи към елемент, който в паметта е разположен преди елемента към който сочи py. Ако това не е така, отношението ще има стойност FALSE.
Единственото смислено присвояване на цяло число на указател е присвояването на нулевата константа NULL. Смисълът на такова присвояване е необходимостта от означаване на особени случаи при използване на указатели (край на масив, грешка и т.н.).

Указател може да се сравнява с нулевия указател. Например if (p == NULL) … или if (p != NULL) … , са валидни фрагменти от програмни оператори.

Пример:
Реализация на стек. Използваме две функции push(int) и pop(), в които използваме адресна аритметика. Потребителският стек създава и използва 10 цели числа.
push – функция, която добавя цяло число към стека, pop – функция, която извлича число от стека.

Ще използваме функции за разпределяне и освобождаване на памет.

void *malloc (size_t size);
Прототипът на тази функция се намира във malloc.h, size_t е целочислен тип, дефиниран в malloc.h. Функцията malloc разпределя блок динамична памет, който има size байта.

Функцията връща указател към първия (най-малкия по адрес) елемент от блока.

void free (void *block);
Прототипът се намира в malloc.h. Функцията free приема като фактически параметър указател към блок динамична памет и освобождава заетата от него памет. За блока преди това трябва да е разпределена памет с функцията malloc (има и други функции).

```
#include <iostream.h>
#include <malloc.h>
void push (int);
int pop();
int *pmin, *p;
void main ()
 { pmin = (int *) malloc (40);
   if (pmin==NULL)
    { cout << “Наличната памет е недостатъчна!”;
      return;
    }
  p = pmin;
  push (10);
  push (20);
  cout << “2-рото число е: “ <<  pop()  << endl;
  cout << “1-рото число е: “ <<  pop()  << endl;
  free (pmin);
 }
```

```
void push (int i)
 { if (p==pmin+10)
   { cout <<  “Препълване на стека!” << endl;
     return;
   }
  *p = i;
  p++;
 }
int pop()
 { if (p==pmin)
    { cout <<  “Стекът е празен!” << endl;
      return 0;
    }
  p--;
  return *p;
}
```

Указателят p сочи към елемента, който е точно над върха на стека (свободният елемент), указателят pmin към дъното (началото) на стека.

В C++ има и други функции, които могат да разпределят памет, например със средствата на обектното програмиране функциите new за разпределяне и delete за освобождаване.

Важно: за всяка функция, която разпределя памет има точно определена функция с която се освобождава тази памет - например памет разпределена с malloc не може да бъде освободена с delete.

27. Едномерни масиви и указатели

Между указателите и масивите съществува взаимозаменяемост, произтичаща от техните определения в езика С. Указател към масив се дефинира по общоприетите правила, като се спазва изискването за еднаквост на типа на указателя и типа на масива, към който сочи указателя. Например, ако масив е дефиниран като

```
float array[20];
```

указателят към него трябва да се дефинира от същия тип

```
float *pa;
```

От друга страна, използването на името на масива е равносилно на обръщение към адреса на първия елемент на този масив. Името на масива се разглежда като константа, чиято стойност е адресът на първия елемент на масива. Следователно указател към масив може да се инициализира като

pa = array; или pa = &array[0];

За да достигнем например до третия елемент на масива, трябва да използваме означението array[2].
От дефиницията на указателите и от адресната аритметика знаем, че увеличението на указател към масив с единица означава достъп до следващия елемент на масива, независимо от типа на този масив. Увеличението на указател към масив с цяло число е равносилно на обръщение към елемент на масива с индекс това число, т.е. вместо чрез array[2] същият този трети елемент можем да достигнем и с помощта на *(pa + 2)
Например, ако pa е указател към масива array, т.е. към елемента array[0],
z = *pa
присвоява на z стойността на array[0],
z = *(pa + 1)
присвоява на z стойността на array[1],
z = *(pa +2)
присвоява на z стойността на array[2] и

z = *(pa +i)
присвоява на z стойността на array[i].
Можем да използваме и конструкции от вида
z = *(array + i)
когато искаме да присвоим на z стойността на array[i]. Това е пряко следствие от факта, че името на масива е всъщност указател към първия елемент на масива. В процеса на компилация всички конструкции от вида array[i] се превръщат в *(array+i). Оттук следва, че &array[i] и (array + i) са еквивалентни - (array + i) всъщност е адресът на i-тия елемент на масива. Трябва да обърнем внимание, че името на масив е константа, а съответно указател е променлива, т.е. опитите да се използват конструкции като
array++, pa = &array, array = pa
са грешни.

Езикът C дава два начина за обръщение към елементите на масив:

- чрез индекси;
- чрез указатели.

Използването на указатели е по-бързият начин за достъп до елементите на масив.

Когато елементите на даден масив се използват строго последователно (т.е. индексите им растат или намаляват с единица), използването на указатели е по-бързият и по-удобният начин. Ако елементите на масива се използват в случаен порядък (т.е. индексите им се изменят произволно), се препоръчва използването на индекси като по-лесен и разбираем начин за програмиране, независимо че това е по-бавният начин.

Повечето функции в C, които обработват низове, използват указатели и адресна аритметика. Да разгледаме функцията cmpstr(), която се използва за сравняване на низове. Тази функция сравнява два низа символ по символ и връща ASCII-кодът на първия символ от низа s1, различен от съответния в низа s2. Ако

низът s2 е по-голям, се връща ASCII-кодът на първия символ в повече, който няма съответен в s1. Ако низовете са равни, се връща нулева стойност. Функцията сравнява символите в двата низа с помощта на цикъл по символите на низа s1. Тя може да бъде реализирана принципно по два начина:

- чрез използване на масиви и индекси:

```
int cmpstr(char s1[], char s2[])
{ int t;
for (t=0; s1[t]; ++t) // цикъл по символите на s1
    if (s1[t] != s2[t])
        return s1[t]; // различен символ от s1
    if (s2[t]) return (s2[t]); // символ в повече от s2
    return(0); // равни низове
}
```

- чрез указатели

```
int cmpstr (char *s1, char *s2)
 { while (*s1)  // цикъл по символите на s1
     if (*s1++ != *s2++) return *--s1; // различен символ от s1
     if (*s2) return (*s2);  // символ в повече от s2
     return (0);  // равни низове
 }
```

Тъй като всеки низ завършва с '\0', т.е. с нулева стойност, която е и стойността FALSE, изразът *s1 в оператора

```
while (*s1)
```

ще бъде истина и тялото на този оператор ще се изпълнява, докато се достигне краят на низа.

В езика C няма оператори, които да обработват низове като цяло. За сметка на това в C има много стандартни функции, които позволяват да се извършват различни операции с низове. Тези функции осъществяват достъпа до всеки низ в програмата с помощта на указател към този низ. Можем да запишем

```
char *ps;
ps = “Това е низ!”;
```

но трябва да разглеждаме този програмен фрагмент като инициализация на указател към низ, т.е. низът не се копира, а се инициализира указателя ps с адреса на първия елемент на символния масив, чрез който този низ е представен в компютъра. Тъй като указателите са променливи, е възможно те да бъдат съхранявани в масиви от указатели. Тези масиви се дефинират съгласно общите правила, като се добавя символът за указател * при описанието на типа им. Например, за да дефинираме масив от указатели към цели числа, трябва да напишем

```
int *px[4];,
```

а за да инициализираме указателите – елементи на този масив с адресите на целите променливи

int x1, x2, x3, x4;

можем да използваме

px[0] = &x1; px[1] = &x2; px[2] = &x3; px[3] = &x4;

Винаги трябва да помним типа на масива от указатели, тъй като единствените стойности, които този масив може да съдържа, са адреси на променливи от същия тип (от тип int в примера).

Можем да дефинираме масив от указатели и да го инициализираме по следния начин:

int *px[] = { &x1, &x2, &x3, &x4 };

28. Многомерни масиви и указатели

Можем да дефинираме масив от указатели и като:
char *py[];
където квадратните скоби определят, че py е указател към масив, елементите на който са указатели към променливи от тип char. Без квадратните скоби py ще се превърне в указател към променлива от тип char. Както знаем, дължината на масиви, дефинирани без горната си граница в квадратните скоби, се определя при инициализирането им. Например

char *error[] = { “Грешка 1”, “Предупреждение 1”, “Фатална грешка”, “Няма грешка” };

Всеки елемент в масива error[] е указател към съответния низ, а отделните низове могат да бъдат съхранявани на различни места в паметта.

Тук трябва да обърнем внимание на подобието и разликата между масив от указатели към едномерни масиви и двумерен масив. В действителност всеки двумерен масив може да се представи като едномерен масив от указатели към едномерни масиви с еднаква дължина, т.е. с еднаква горна граница. Съществената разлика е в начина на дефиниране. Последният пример може да бъде записан и с помощта на двумерен масив

```
char error[4][17] =
    {
    {“Грешка 1”},
    {“Предупреждение 1”},
    {“Фатална грешка”},
    {“Няма грешка”}
    };
```

и този масив заема 68 (4*17) байта в паметта. Дефиниран като масив от указатели, той изисква 4 клетки памет за всеки указател, 9 байта за първия низ, 17 – втория и т.н. Общо по-малко памет от

двумерния масив. Следователно, използването на масиви от указатели има следните предимства пред двумерните масиви:

- не се блокира цялата памет както при двумерните масиви. Всеки указател може да сочи към едномерен масив с произволна дължина и се ангажира само необходимата за този масив памет. При двумерните масиви винаги се отделя памет, определена като произведение на горните граници на масива;
- достъпа до елементите на масива се осъществява чрез бързата адресна аритметика;
- опростява се управлението на паметта. Ако е необходимо да разменим местата на два реда от даден масив, в случая на масив от указатели трябва да разменим само стойностите на указателите. Масивите, към които те сочат, остават на местата си. При двумерния масив трябва да разменяме съобщенията символ по символ.

Най-често масиви от указатели се използват за съхраняване на низове с различна дължина, например за показване на съобщения за грешки, съответни на номера на грешката.
Концепцията за масив от указатели всъщност е израз на възможността за дефиниране на указатели към указатели в езика С. Тази възможност разширява косвения достъп и го превръща в многократен косвен достъп.

```
int x, *px, **ppx;
// px е указател към цялата променлива x, а  ppx е указател към указателя px
x = 20;
px = &x; // px = адреса на променливата x
ppx = &px; // ppx = адреса на указателя px
```

пряк достъп до стойността на x – x;
косвен достъп до стойността на x - *px, **ppx

```
float matrix[2][2];
matrix[0] ⇔ &matrix[0][0];
```

matrix[1] ⇔ &matrix[1][0];
matrix[i] ⇔ &matrix[i][0];
Това е поради интерпретацията на двумерния масив като масив от указатели към едномерни масиви.
При достъп до елементите на масива може да се използва както указател към началото на масива и после той да се увеличава, така и указатели към редовете на масива. Грешка е

```
float matrix[2][2];
float *pm;
pm = matrix;
```

Това е само при едномерен масив.
pm = (float *)matrix; - вярно при всички масиви, даже и тримерни например.
В сила е:
matrix[i][j] ⇔ *(matrix[i] + j) ⇔ * (*(matrix+i) + j);

float a[DIM1][DIM2];
&a[i][j] ⇔ (float *)a + i*DIM2 + j;
Това е така, защото името на един двумерен масив се разглежда като указател към указател към първия елемент на масива. Виждаме, че не се използва първата размерност, за да се достигне до a[i][j].

Ако имаме един указател, можем да го използваме с индекси:
int data = 5;
int *ptr = &data;
ptr[0] ⇔ *ptr; ptr[i] ⇔ *(ptr + i);

29. Операции с отделни битове

Езикът С предлага пълен набор от операции за обработване на отделни битове на целочислени стойности. Тези операции се наричат **поразрядни логически операции**. Те позволяват променяне, проверяване, установяване или преместване на отделни битове от стойността на променлива от тип int или от тип char. Операциите не могат да се използват с операнди от тип float, double. Поразрядните логически операции често се използват за създаване на драйверни програми за управление на външни устройства.

Всеки обект се разполага като двоично число в паметта на компютъра в един или повече байта. За 32-разредните компютри машинната дума е съставена от 4 байта (32 бита). На фигурата са показани поредните номера на битовете в думата, разряд номер 31 е знаковия разряд:

31							24	23							16	15							8	7	6					1	0
Старши байт																								Младши байт							

Поразрядните операции се означават по следния начин:
& - поразрядно логическо “и” (AND),
| - поразрядно логическо “или” (OR),
^ - поразрядно логическо изключващо “или” (XOR),
~ - поразрядно логическо допълване до 1 (отрицание),
>> - поразрядно изместване надясно,
<< - поразрядно изместване наляво.

Първите три операции са бинарни - действат върху два операнда, които могат да бъдат произволни валидни изрази, имащи за резултат цели числени стойности. Операциите са комутативни и операндите могат да се разместват, без това да влияе на крайния резултат. Тяхното действие се основава на известните логически операции, приложени за всеки отделен бит на операндите, както е показано на таблицата.

Поразрядна операция	Битове в		
	операнд1	операнд2	операнд3
И (AND)	1 0 1 0	1 1 0 0	1 0 0 0
ИЛИ (OR)	1 0 1 0	1 1 0 0	1 1 1 0
изключващо ИЛИ (XOR)	1 0 1 0	1 1 0 0	0 1 1 0

Поразрядно логическо И

операнд1 & операнд2

Операндите са изрази от цял или символен тип. Поразрядно се извършва операцията “и”.

Операцията се използва обикновено за нулиране на отделни битове в числената стойност на резултата. Ако някой бит в единия или в другия операнд има нулева стойност, съответният по позиция бит в резултата също има нулева стойност. Конфигурация от нули и единици, за постигане на цели като горната и др., се нарича маска.

Ако искаме да нулираме битове в едно число от тип int използваме маска, в която единицата запазва бита, а нулата го нулира.
11011011
&
10111101

10011001 – използваме маската 10111101 за да нулираме втори (1) и седми (6) бит в двоичното числото 11011011.

Поразрядно логическо ИЛИ
операнд1 | операнд2
Операндите са изрази от цял или символен тип. Поразрядно се извършва операцията “или”.
Операцията може да се разглежда като обратна на предишната операция, т.е. тя се използва за задаване на стойност 1 на отделни битове в резултата. Всеки бит, чиято стойност е равна на 1 в операндите, определя стойност 1 на съответния бит в резултата.

Например, ако искаме да направим определени битове единици в едно число от тип int, използваме маска, в която единицата променя бита на единица, а нулата го запазва.

10110010
|
01000010

11110010 – използваме маската 01000010 за да променим втори (1) и седми (6) бит на единици в двоичното числото 10110010.

Поразрядно логическо ИЗКЛЮЧВАЩО ИЛИ

операнд1 ^ операнд2

Операндите са изрази от цял или символен тип. При тази операция определени битове в резултата се установяват равни на 1 само ако съответните им битове в операндите са различни (или събиране по mod 2).

11010101
^
01111111

10101010

Поразрядните операции не се използват така, както се използват логическите операции. Например, ако имаме x=5 и x&&10, резултатът от тази логическа операция е 1, а за поразрядната операция x&10 резултатът е равен на 0.

Поразрядно логическо ДОПЪЛВАНЕ ДО 1
~операнд
Това е унарна операция с асоциативно правило на изпълнение отдясно наляво. Операндът е израз от цял или символен тип.

При изпълнението на операцията всеки бит на операнда се променя в обратната си стойност, т.е. ако е бил 1, става 0, а ако е бил 0, става 1. От това следва, че двукратното изпълнение на операцията възстановява началната стойност на операнда.
~11010101 -> 00101010, ~00101010 -> 11010101

int x, y;
y = ~x + 1; ⇔ y = -x, ако x > 0
Това е така, тъй като тип int се представя в допълнителен код в паметта.

Изместване надясно
операнд1 >> брой битове
Изместване наляво
операнд1 << брой битове

Последните две поразрядни операции са бинарни. И за двете операции операнд1 е валиден резултат от цял или символен тип.

При изпълнението на операциите всички битове на числената стойност на операнд1 се изместват съответно вдясно или вляво с толкова позиции, колкото е стойността на брой битове. Брой битове е валиден израз с резултат цяла стойност. Типът на резултата от операцията е типът на операнд1. В повечето случаи оставащите празни позиции след изместването се запълват с 0.
Има компилатори обаче, които в някои случаи (цели числа със знак например) добавят 1 в оставащите вляво празни позиции при изместване надясно. Изместваните битове, които излизат вдясно (при >>) или вляво (при <<), се губят. Резултатът от операциите е неопределен, ако брой битове има отрицателна стойност или е по-голям или равен на дължината на операнд1 в брой битове.

Едно от полезните приложения на тези две операции е осъществяването на много бързо умножение или деление. Всяко изместване с 1 позиция вляво ще даде като резултат стойността на операнд1, умножена по 2, а изместването с една позиция вдясно – стойността на операнд1, разделена с 2. Например, ако x има стойност 15 (десетично), т.е. 00001111 (двоично). x << 3 ->

00001111

<<

3

01111000 = 120.

Ако върху получената стойност приложим операцията x >> 3 получаваме

01111000

>>

3

00001111 = 15.

Операциите изместване наляво и изместване надясно могат да се използват за бързо умножение или деление на степени на двойката.

Операцията ~ има приоритет 2 и е дясно-асоциативна. Останалите са с различни приоритети и са ляво-асоциативни.

Бинарните поразрядни операции могат да участват в съкратената форма на операцията за присвояване, например:
операнд1 = операнд1 << операнд2 ⇔ операнд1 <<= операнд2
Асоциативното правило за това присвояване е отдясно наляво.

Пример:
Функция, която преобразува десетично число в допълнителен код.

```
#include <iostream.h>
void detob (int d, int a[]) // d – число за преобразуване, а – масив за
резултата
 { int c = 0, p = 1; // c – брояч на разредите, p – текущата позиция
(не число)
   while (p!=0)
    { a[c++] = ( (d&p) && 1 ); // запис 1 или 0 според стойността на
текущия бит
       p <<= 1; // изместване вляво за следващия бит
    }
 }
```

```
void main()
{int b[32]; // масив, който съдържа 32 двоични цифри
 int n, i; // n – преобразуваното число
 for (n=0;n<=16;n++)
 {cout << n << " = "; // текущо число
  detob(n,b);
  for(i=31;i>=0;i--) //показване на двоичните цифри в ред, обратен на
определянето им
     cout << b[i];
  cout << endl;
 }
}
```

Извличаме кодировката чрез масива, а преобразуването се прави като се използва, че едно цяло десетично число се записва в паметта с допълнителен код.Функцията получава цялото десетично число с параметъра d, което функцията ще преобразува. Допълнителният код се получава в масива, където отделните битове ще бъдат елементи на масива.

30. Предаване на резултатите чрез формални параметри

При разглеждането на съвместната работа на функции и указатели възникват два въпроса: как указател се предава като параметър на функция и може ли върната стойност от функция да бъде указател. Отговорите не изискват нови знания за С и се съдържат в усвоените вече езикови конструкции.
В съответствие с общите правила, валидни за параметри на функция, в С няма ограничения при предаването на указатели като параметри. Указателят е променлива и използвана като формален параметър, тя се описва така, както се дефинира;

```
void fun(int x, int *p)
{
int i=5;
…
*p=10;
```

```
…
}
```

В примера формалният параметър p е указател към променлива от тип int. Както всяка променлива, указателят се предава в тялото на функцията чрез стойност, т.е. в нея се използва копие на фактическия параметър – указател. Така че промените в тялото на функцията се отнасят за копието, а не за фактическия параметър. Обектът обаче, към който сочи фактическия параметър - указател, може да бъде променян във функцията. В примера на него е присвоена нова стойност чрез оператора *p=10;. По този начин в C се реализира механизма за предаване на параметри във функция по адрес. Под предаване по адрес се разбира, че всяка промяна на формален параметър на функцията в тялото й е промяна и на стойността на променливата, използвана като съответен фактически параметър. В C всички обекти, с изключение на масиви, се предават във функция по стойност.

Предавайки обаче адресите на променливите като стойности на указатели, програмистът може сам да реализира механизъм за предаване по адрес.
Адресите на променливите могат да се получат с адресната операция & и директно да се запишат в обръщението на функцията.

```
main()
{
int a=15, b=5;
fun(a,&b);
cout << a << " " << b << " " << endl;
}
```

В извикващата функция не е нужно да се дефинира указател, да му се присвоява стойност и след това да се използва като фактически параметър във функцията fun. Изразът &b в обръщението към fun има конкретна стойност – адресът на b, и тази стойност се приписва на указателя p във функцията fun.

Ако е необходимо поради други съображения, в извикващата функция можем да дефинираме указател и да работим с него.

```
main()
{
int a=15, b=5;
int *q=&b;
fun(a,q);
cout << a << " " << b << " " << endl;
}
```

Инициализирането на указателя q в описанието му може да стане едва след като е дефинирана променливата b.

Съществено предимство на използването на указатели като параметри на функция е, че извиканата функция може да върне повече от една стойност в извикващата функция. По-точно, извиканата функция може да промени повече от една променливи, дефинирани в извикващата функция. За сравнение ще припомним, че с оператора return може да се върне една единствена стойност.

Ако е необходимо тази стойност да се запази, тя трябва да се присвои на променлива в извикващата функция. Възможността за промяна в извиканата функция на няколко променливи, дефинирани в извикващата функция, като се използват параметри – указатели, наподобява тази при глобалните променливи. Особено важно в случая е, че тези промени се локализират в тялото на функцията и много по-ефикасно се контролира поведението на променливите.

Пример: Програма, която пресмята дължина на окръжност и лице на кръг

```
#include <iostream.h>
void circle(float r, float *s, float *p)
{
float pi = 3.141592;
*s = pi*r*r;
*p = 2*pi*r;
} // чрез r внасяме стойност, а чрез p и s - изнасяме
void main()
{
float a,b,c;
cout << "Въведете радиус.";
cin >> a;
circle(a, &b, &c);
cout << "P = " << c << endl;
cout << "S = " << b << endl;
}
```

Псевдоними

Чрез псевдонимите се задават алтернативни имена на обекти в общия смисъл на думата (променливи, константи и др.). Тук ще ги разгледаме ограничено (псевдоними само за променливи).

Дефиниране.

Нека T е име на тип. Типът T& е тип псевдоним на T. T се нарича базов тип на типа псевдоним.

Множество от стойности

Състои се от всички имена на дефинирани вече променливи от тип T.

Пример: Нека програмата съдържа следните дефиниции:

```
int a, b = 5;
...
int x, y = 9, z = 8;
...
```

Множеството от стойности на типа int& съдържа имената a, b, x, y, z.

Променлива величина, множеството от допустимите стойности на която съвпада с множеството от стойности на даден тип псевдоним, се нарича променлива от този тип псевдоним.
Дефиницията

```
T& a = b, c = d;
```

е еквивалентна на

```
T& a = b;
T c = d;
```

т.е. знакът & след типа T се отнася само за първата променлива след него.

Пример: Дефинициите

```
int a = 5;
int& syna = a;
double r = 1.85;
double &syn1 = r, &syn2 = r;
int& syn3 = a, syn4 = a;
```

определят syna и syn3 за псевдоними на цялата променлива a, syn1 и syn2 за псевдоними на реалната променлива r и syn4 за променлива от тип int.

Дефиницията на променлива от тип псевдоним задължително е с инициализация – дефинирана променлива от базовия тип на типа псевдоним. След това не е възможно променливата-псевдоним да стане псевдоним на нова променлива. Затова тя е "най-константната" променлива, която може да съществува.

Пример:

```
...
int a = 5;
int &syn = a;  // syn е псевдоним на a
int b = 10;
int& syn = b;          // error, повторна дефиниция
```

```
int n;
int &nn = n;
```

По този начин декларираме nn като псевдоним на n. nn и n означават една и съща променлива (една и съща памет).

```
double a[10];
double &last = a[9];
```

Псевдоним на елемент от масив.

```
const char &new_line ='\n';
```

Това е псевдоним на вградена константа и за нея не се разпределя памет.

```
int i = 5;
int *p = &i;
int *&s = p;
```
s – псевдоним на указателя p.

Операции и вградени функции

Дефиницията на променлива от тип псевдоним свързва променливата-псевдоним с инициализатора и всички операции и вградени функции, които могат да се прилагат над инициализатора, могат да се прилагат и над псевдонима му и обратно.

Примери:

1. int ii = 0;

int& rr = ii;

rr++;

int* pp = &rr;

Резултатът от изпълнението на първите два оператора е следния:

ii, rr = 0

Операторът rr++; променя стойността на ii и тя от 0 става 1. В случая rr++ е еквивалентен на ii++.

Адресът на rr е адресът на ii. Намира се чрез &rr. Чрез дефиницията

int* pp = &rr;

pp е определена като указател към int, инициализиран с адреса на rr.

2. int a = 5;
int &syn = a; // syn е псевдоним на a
cout << syn << " " << a << '\n';
int b = 10;
syn = b;
cout << b << " " << a << “ “ << syn << '\n';
извежда
5 5
10 10 10
Операторът syn = b; е еквивалентен на a = b;.
3. int i = 1;
int& r = i; // r е псевдоним на i
cout << r; // извежда 1
int x = r; // x има стойност 1
r = 2; // еквивалентно е на i = 2;

Допълнение: Възможно е типът на инициализатора да е различен от този на псевдонима. В този случай се създава нова, наречена временна, променлива от типа на псевдонима, която се инициализира със зададената от инициализатора стойност, преобразувана до типа на псевдонима. Например, след дефиницията

```
double x = 12.56;
int& synx = x;
```

имаме: x е double с 8 байта и стойност 12.56, а synx е int с 4 байта и стойност 12.

Сега x и псевдонимът ѝ synx са различни променливи и промяната на x няма да влияе на synx и обратно.

Константни псевдоними

В C++ е възможно да се дефинират псевдоними, които са константи.

За целта се използва запазената дума const (const T & <променлива> = <инициализатор>; или T const & <променлива> = <инициализатор>), която се поставя пред дефиницията на променливата от тип псевдоним. По такъв начин псевдонимът не може да променя стойността си, но ако е псевдоним на променлива, промяната на стойността му може да стане чрез промяна на променливата.

Пример: Фрагментът

```
int i = 125;
const int& syni = i;
cout << i << " " << syni << '\n'; // i и syni имат стойност 125
syni = 25; // грешка
cout << i << " " << syni << '\n';
```

ще съобщи за грешка (syni е константа и не може да е лява страна на оператор за присвояване),

но фрагментът

```
int i = 125;
const int& syni = i;
 cout << i << " " << syni << '\n';
 i = i + 25;
 cout << i << " " << syni << '\n';
```

ще изведе

```
125 125
150 150
```

Последното показва, че константен псевдоним на променлива защитава промяната на стойността на променливата чрез псевдонима.

Пример: Програма за размяна на стойностите на две реални променливи, предаването на параметрите е по адрес.

```
void swap (float *a, float *b)
 { float temp;
   temp = *a;
   *a = *b;
   *b = temp;
 }
void main ()
 { float x, y;
   …
   if (x > y) swap (&x, &y);
   …
 }
```

Пример: Трети начин за предаване на параметри в C++ с предаване на параметри по псевдоним.

```
void swap (float &a, float &b)
{ float temp;
   temp = a;
   a = b;
   b = temp;
 }
void main ()
 { float x, y;
   …
   if (x > y) swap (x, y);
   …
 }
```

В този случай формалният параметър a става псевдоним на x, а формалният параметър b – псевдоним на y. За тях не се отделя памет в стека и се работи с фактическите параметри от main.

31. Функции от тип указател

Резултатът от една функция може да е указател – тогава самата функция е от тип указател. Изразът в оператор return във функция може да е адресен израз, чиято стойност е валиден адрес, например

return (&b); // b e променлива

return (p+=i); // p е указател, i e int

Отговорност на програмиста е да състави и използва в оператора return израз, който дава валиден за програмата адрес. Точно тук се допускат много трудни за откриване грешки.

Съгласно общите правила типът на връщания резултат трябва да бъде описан преди името на функцията. Адресът е стойност на указател, който сочи към обект от определен тип. Затова в случая типът е указател към обект от тип..., което в заглавието на функцията се записва по следния начин

тип_на_обекта *име_на_функцията(списък_на_параметри)

Например, валидни са следните описания:
int *fun(…) // връщаната стойност е указател към обект от тип int
char *fun(…) // връщаната стойност е указател към обект от тип char

Когато върнатата стойност от една функция е адрес, обикновено тя се използва или в адресни изрази, или за извличане на стойността на обекта, към който указва.

```
int *f1 (списък_от_формални_параметри)
 {
   …
   return (израз);
   …
 }
```

Пример:
Да се определи броя на символите, различни от ‘ ‘ и ‘\t’ в един низ.

```
int length (char *p)
 { int i; char c;
   for (i=0; (c=*p)!=’\0’; p++)
    if (c!=’ ‘ && c!= ‘\t’) i++;
   return i;
}
```

Функция, която сравнява два низа сочени от p и q по брой на непразни символи и връща указател към по-дългия.

```
char *compare (char *p, char *q)
 { return ( ( length(p)>length(q) ) ? p : q ); }
```

32. Формални параметри и масиви

Функция, която намира максималния елемент на редица от n числа от тип float. Редицата е записана в масив, който се предава като параметър на функцията. Намерената максимална стойност m се връща като резултат.

```
float maxi (float c[], int n)
 { int i; float m = c[0];
   for (i=1;i<n;i++)
    if (m<c[i]) m = c[i];
   return m;
 }
void main ()
 { float mas[100], x;
   int broi;
   …
   x = maxi (mas, broi);
   …
```

}

Указвайки името на масива в обръщението ние всъщност задаваме адреса към първия елемент на масива. За масива c във функцията maxi ще се разпредели точно тази памет, където се намира адреса на първия елемент на mas. Когато ползваме c във функцията, то все едно работим с mas. Ако вътре във функцията променяме c, то ние променяме и mas, т.е. имаме предаване по адрес.

Вграденият начин за заместване на формални с фактически параметри масиви не предполага, че целият масив ще се запише в стека, а само адресът на първия елемент.

Можем да запишем по друг начин функцията

```
float maxi (float *p, int n)
{ int i; float m = *p;
  for (i=1;i<n;i++)
   if (m<*++p) m = *p;
  return m;
}
```

Това е така, защото името на масива е указател към първия елемент и елементите на масива са разположени последователно в паметта.
И в двете реализации можем да запишем в главната функция
int i = 4;
x = maxi (&mas[i], broi – i); // или x = maxi (mas+i, broi – i);
Тогава на c[0] се съпоставя mas[i], т.е. петият елемент mas[4].

Предаване на двумерни и тримерни масиви като параметри:
функция за транспониране на матрици

```
const int DIM = 10 // дефиниране на константа
void trans (float [][DIM], float [][DIM]);
void main ()
 { float u[DIM][DIM], v[DIM][DIM];
   …
   trans (u, v);
   …
  }
void trans (float a[][DIM], float b[][DIM])
 { int m, n;
   for (m=0;m<DIM;m++)
    for (n=0;n<DIM;n++)
     b[n][m] = a[m][n];
 }
```

Формалните параметри са двумерни масиви, а фактическият параметър е името на съответния двумерен масив или указател към указател към началото на масива. Вместо двумерен масив, можем да използваме указател към указател.

Например:

```
void fun (int i, int **p)
 { …
 }
void main ()
 { int *a[10], k;
   …
   fun (k, a);
   …
 }
```

Фактическият параметър е масив от указатели, който се предава и описва във функцията като указател. Ако във функцията fun използваме **p – това е стойността, записана в адреса към който сочи първия указател в масива a (указателят a[0]).

В двумерен масив от символен тип се записва текст. Всеки ред е съобщение. Да се напише програма за преобразуване към малки букви. Ще използваме функция tolower - int tolower (int ch); Нейният прототип се намира в заглавния файл ctype.h. Функцията преобразува големи букви към малки, ако ch не е голяма буква, нищо не се променя.

```
const int ROW 100
const int COL 200
void trans (char *p);
void main ()
 { char str[ROW][COL];
   … //вход
   for (int i = 0; i < ROW; i++)
    trans (str[i]);
   … //изход
  }
```

```
void trans (char *p)
 { for (; *p != ‘\0’; p++)
    *p = tolower (*p);
 }
```

33. Указатели към функции

Функцията е сложен обект и по известните досега правила не можем да определим указател към нея. В езика C е разрешено дефинирането на указатели към функции, като се използва следния синтаксис:
тип_на_връщания_резултат (*идентификатор)();
където идентификатор е допустим за езика идентификатор;
тип_на_връщания_резултат – описание на типа на връщаната стойност от функцията.
Валидни описания на указатели към функция са например:
int (*pfun)(); по този начин дефинираме променливата pfun като указател към функция, която няма параметри и връща резултат от тип int.
char (*qfun)(int); по този начин дефинираме променливата qfun като указател към функция, която има един параметър от тип int и връща резултат от тип char.

double (*sfun)(int,char); по този начин дефинираме променливата sfun като указател към функция, която има първи параметър от тип int, втори параметът от тип char и връща резултат от тип double. Аналогията между тези описания и дефиницията на указател можем да потърсим по следния начин. Обектът pfun е

- указател (това следва от символа *),
- към функция (това се показва от отварящата и затварящата скоба () на списъка на параметрите),
- която връща резултат от тип int (това е типът на обекта, който може да се използва чрез указателя).

Скобите, в които се затваря *идентификатор, са задължителни. Ако те бъдат пропуснати, за първия пример бихме получили
int *pfun();
Това обаче е декларация на функция, която връща резултат от тип int*. Така че изпускането на скобите няма да се сигнализира веднага от компилатора, но малко по-късно той ще ни напомни, че неправилно използваме недефиниран указател към функция.

Поставяйки скобите, ние повишаваме приоритета на операцията *. В противен случай малките скоби () имат по-висок приоритет от * и компилаторът започва разшифроването на описанието от тях.

Какво всъщност представлява указател към функция? Той е променлива, която съдържа адреса на началото на кода на функцията. Такива и само такива стойности могат да се присвояват на указатели към функции. Името на функция се третира от компилатора като стойност на указател към функция. Затова имената на функциите могат да се присвояват директно на указатели към функции (без да се използва адресната операция &), например:

```
extern int fun();
int (*pfun)();
…..
pfun = fun; // на указателя pfun е присвоен адресът на функцията fun
```

Очевидна е аналогията между указатели и имена на масиви. Операцията присвояване е една от малкото допустими операции с указатели към функции. При указателите към функции адресната аритметика няма смисъл.
Дефинирайки указател към функция, ние косвено загатваме, че съществува и начин за извличане на стойността на обекта, към който сочи този указател, така както е при обикновените указатели. При указателите към функции това може да означава само едно нещо, а именно, генериране на обръщение към функция, т.е. предаване на управлението на адреса на функцията. Тъй като функцията има и списък с параметри, обръщението към нея става по следния начин:
(*идентификатор) (параметър1, параметър2, ...);
където идентификатор е указател към функция, имащ конкретна стойност.
Този начин за обръщение към функция е непряк.

Общото правило е, че компилаторът разглежда всяко име на функция без скоби като указател към тази функция. Да разгледаме някои непреки обръщения към функции:

```
int fun (char i, char j)
{
тяло на функцията
}
main()
{
int (*pfun)(char, char);
int a,b,x;
…..
pfun = fun; // присвояване на стойност на указателя pfun
x = (*pfun)(a,b); // непряко обръщение към  fun
……
}
```

Не е задължително винаги да дефинираме указател, на който да присвояваме името на функция, за да осъществим след това непряко обръщение. Можем направо да използваме името на функцията като указател. В последния пример бихме могли да запишем обръщение

```
x = (*fun)(a,b);
```

В този случай идентификаторът fun вече е известен на компилатора като име на функция.

Когато прилагаме операция * към указател към функция, това значи да се предаде управлението към функцията (да се направи обръщение към нея).

Пример:

```
int k, x;
int func ();
int (*pf)();
pf = func;
// името на функцията се разглежда като константен указател към
// тази функция (то съдържа адреса на началото на функцията)
```

k = (*pf)(); //косвено обръщение към функцията
x = func(); //пряко обръщение към функцията
// чрез указател към функция може да се извършва и директно обръщение
k = pf();
// чрез името на функцията може да се извършва и косвено обръщение
x = (*func)();
Малките скоби отново са задължителни. Иначе например
k = *pf ();
ще се интерпретира по следния начин: резултатът от pf () ще се разгледа като адрес и на k ще се присвои стойността, записана в този адрес.

Ако функцията не е още описана или е описана в друг файл, е необходима нейна декларация. Да разгледаме следния пример, илюстриращ употребата на указатели към функции в изрази:
extern int fun1(int, int), fun2(int, int);
(*((a > 5) ? fun1 : fun2))(x, y); // предполагаме, че fun1 и fun2 имат еднакви параметри
Предполагаме, че функциите fun1 и fun2 са описани в друг файл. В примера условният израз получава стойност, която е адресът на fun1 или fun2. След това по тази стойност се извършва непряко обръщение към функция. Това е възможно при предположение, че fun1 и fun2 имат еднакви по брой и тип параметри.
За да се убедим колко е важна употребата на скобите при непрякото обръщение към функция, нека да разгледаме една функция fun, която връща като резултат указател към обект от тип int. Как изглежда описанието на указател към такава функция? От изложените правила следва да запишем:
int *(*pfun)();

Непрякото обръщение към fun изглежда така:
```
extern int *fun(int, int, …);
int *x;
……….
x = (*fun)(параметър1, параметър2, ...);
```
Ако пропуснем първите скоби в последния оператор, получаваме валидна за езика конструкция
```
x = *fun(параметър1, параметър2, ...);
```
С нея на променливата x се присвоява стойността на обекта, към който сочи указателят, върнат от функцията fun. А всъщност ние искаме да присвоим на x върнатата стойност от fun, т.е. върнатия адрес. Компилаторът няма да сигнализира за грешки, а да разбере грешното поведение на програмата в този случай ще бъде трудно за всеки програмист.

Пример:

```
#include <iostream.h>
int suma(int,int);
int razlika(int,int);
void main()                    // Същият резултат можем да получи и така
{int (*func_ptr)(int,int);// void main()
int var1=5;                   // {int var1 = 5, var2 = 20, rez;
int var2=20,rez;              //rez=((var1<var2)?suma:razlika)(var1*var2,var2);
func_ptr=(var1<var2) ? suma:razlika;// cout << endl << " rez= " << rez;
rez = (*func_ptr)(var1 * var2,var2);    //}
cout << endl << " rez= " << rez;
}
int suma (int a, int b)
{return a+b;}
```

int razlika (int a, int b)
{return a-b;}

Указателите към функции могат да бъдат елементи на съставни обекти, каквито са масивите. Описанието на масив от указатели към функции има следния вид:

int (*array[])();

Като следваме приоритета на операциите в описанието, можем да го разшифроваме по следния начин. Обектът array е:

- масив (операцията [] е с по-висок приоритет от операцията *),
- от указатели (следва от символа *),
- към функции (следва от списъка на параметрите ()),
- които връщат стойност int.

За да се обърнем непряко към функцията, указател към която е елемент на масива, трябва да запишем:

(*array[i])(); където индексът i има конкретна стойност, по-малка от границата на масива.

Стойностите на елементите на масива array са имена на функции, описани в програмата. Масивът се инициализира по следния начин:

```
extern int fun1(), fun2(), fun3();
static int (*array[3])() = {fun1, fun2, fun3};
int (*array[])(int, int) = { suma, razlika } ;
```

Можем да използваме този масив.

```
ret = (*array[ (var1 < var2)? 0 : 1 ])(var1*var2, var2);
```

Това е косвено обръщение.

```
ret = array[ (var1 < var2)? 0 : 1 ](var1*var2, var2);
```

Това е пряко обръщение.

Указателите към функции позволяват да се реализира нова операция – предаване на функция като параметър. Функция не може да се предава директно като параметър. Указател, в това число и указател към функция, може да бъде параметър на друга функция. Това позволява да се управлява избора на функцията, към която има обръщение в друга функция.

```
#include <iostream.h>
int suma (int, int);
int razlika (int, int);
void f(int a, int b, int (*sf)(int, int))
{int rez;
rez = (*sf)(a*b,b);
cout << endl << " rez = " << rez;
}
void main ()
 { int var1 = 5, var2 = 20;
 f(var1 , var2, (var1 <var2)? suma: razlika);
}
int suma (int a, int b)
 { return a+b; }
int razlika (int a, int b)
 { return a-b; }
```

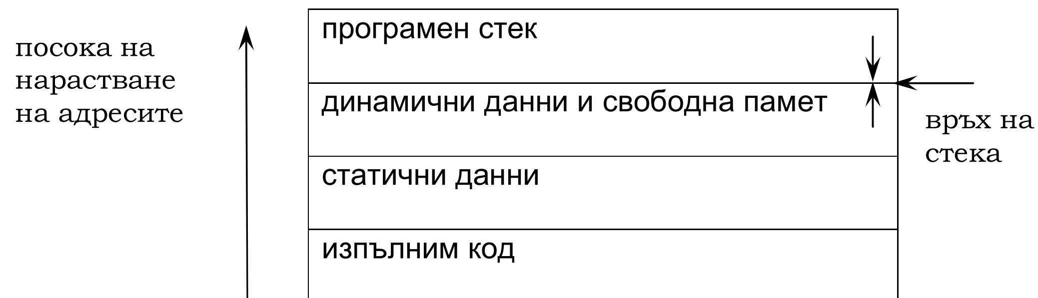
Макар и по-рядко употребявани, указателите към функции имат някои интересни приложения: за построяване на таблици на преходи, при реализация на менюта, при обработване на прекъсвания, при приложения за избиране на различни алгоритми и др. Могат да се избират различни алгоритми чрез предаване на указател към функция. Някои библиотечни функции получават като фактически параметър указател към параметър. Функцията за сортиране qsort получава като четвърти параметър указател към съставена от потребителя функция, която извършва сравнение на два елемента.

34. Функции с променлив брой параметри

Езика С допуска да се използват променлив брой фактически параметри при обръщение към функция. Признак за това, че една функция е с променлив брой параметри, е многоточие в списъка на формалните параметри или в прототипа.
Когато срещне многоточие, компилаторът прекратява контрола за съответствие на типовете на параметрите за тази функция.
Естествено е, че при функция с променлив брой параметри трябва да има начин за точното им определяне при всяко обръщение.
Един от начините е първият фактически параметър да задава броя на фактическите параметри.
Пример: Функция, която извежда броя и стойностите на фактическите параметри.

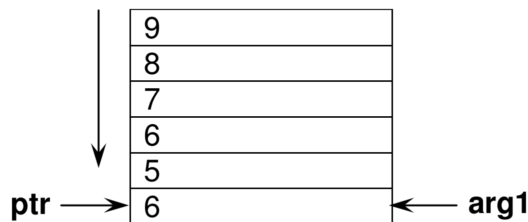
```
#include <iostream.h>
void example (int, ...);
void main ()
 { int var1 = 5, var2 = 6, var3 = 7,
        var4 = 8, var5 = 9;
   example(2, var1);
   example(3, var1, var2);
   example(6, var1, var2, var3, var4, var5);
 }
void example (int arg1, ...)
 { int *ptr = &arg1;
   cout << “Брой на параметрите: “<< arg1 << " " << &arg1 << endl;
   for (; arg1; arg1--)
    {cout <<  *ptr << " " << ptr << endl;ptr++;}
 }
```

Структура на частта от паметта, която е разпределена за една потребителска програма на машинен език:



Резултат:
Брой на параметрите: 2 0x0012FF18
2 0x0012FF18
5 0x0012FF1C
Брой на параметрите: 3 0x0012FF14
3 0x0012FF14
5 0x0012FF18
6 0x0012FF1C

Брой на параметрите: 6 0x0012FF08
6 0x0012FF08
5 0x0012FF0C
6 0x0012FF10
7 0x0012FF14
8 0x0012FF18
9 0x0012FF1C



Както се вижда от реализацията, за да ползваме функция с променлив брой параметри, налага се директно да работим с програмния стек.
Друга реализация – ако фактическият параметър не може да приема дадена стойност, можем да я използваме като признак за край на списъка от параметрите – нула за нашия случай.

```
#include <iostream.h>
void example (int, …);
void main ()
 { int var1 = 5, var2 = 6, var3 = 7,
        var4 = 8, var5 = 9;
   example (var1, 0);
   example (var1, var2, 0);
   example (var1, var2, var3, var4, var5, 0);
 }
void example (int arg1, …)
 { int number = 1;
   int *ptr = &arg1;
   while (*ptr!=0)
    { cout <<  *ptr++ << endl;
      number++;
    }
  cout << endl << “Броят е: “ << number << endl;
}
```

Резултат:
5
Броят е: 2
5
6
Броят е: 3
5
6
7
8
9
Броят е: 6

За да се улесни работата с функции с променлив брой параметри и за да не се интересуваме от особеностите на реализацията, например по какъв начин се разпределят елементите в стека, в езика C има средства, които позволяват да се пропуснат тези особености. За да се използват, трябва да се включи заглавният файл stdarg.h.

Тип va_list – тип на указател към стека (дефиниран в stdarg.h).

Макрос va_start (va_list ap, lastfix) – първият аргумент ap е от тип va_list, а вторият аргумент lastfix е името на последния фиксиран формален параметър. va_start дава начална стойност на указателя ap като адреса на първия формален параметър след последния фиксиран (lastfix).

Макрос va_arg (va_list ap, type) – първият аргумент ap е указател към стека, а вторият аргумент type е типа на формалния параметър, към който сочи ap. Този макрос връща като резултат стойността на съответния фактически параметър от тип type и увеличава ap да сочи към следващия формален параметър.

Макрос va_end (va_list ap) – за извършване на завършителни действия по използването на указателя ap.
Важно: За да можем да работим с функции с променлив брой параметри, задължително трябва да има най-малко един фиксиран формален параметър.

```
#include <iostream.h>
#include <stdarg.h>
void example (int, …);
#define EOL 0
void main ()
 { int var1 = 5, var2 = 6, var3 = 7,
        var4 = 8, var5 = 9;
   example (var1, EOL);
   example (var1, var2, EOL);
   example (var1, var2, var3, var4, var5, EOL);
 }
```

```
void example (int arg1, …)
 { int number = 2, value;
   cout  << arg1 <<endl;
   va_list ptr;
   va_start (ptr, arg1);
   while ( (value = va_arg (ptr, int)) != EOL )
    { cout << value <<endl;
     number++;
    }
 cout << endl << “Броят е: “ << number << endl;
va_end(ptr);
}
```

Резултат:
5
Броят е: 2
5
6
Броят е: 3
5
6
7
8
9
Броят е: 6

Фактическите параметри на функция може да се инициализират с подразбиращи се стойности. Най-добре е това да се направи в декларацията, а не в дефиницията на функцията:

```
int fun(int a1 = 10, char c = 'X');
```

Инициализираните параметри трябва да са подредени последователно отдясно наляво преди затварящата скоба) в заглавието на функцията. Функцията може да има инициализирани и неинициализирани параметри. При обръщението към такава функция може да се пропуснат параметрите, които имат стойности по подразбиране:
fun (); // обръщение към fun() с фактически параметри a1 = 10 и c = ‘X’
Параметрите на функция може да се инициализират с подразбиращи се стойности само веднъж. Инициализиран параметър може да се пропусне в обръщението към функцията, ако всички следващи след него параметри са инициализирани и пропуснати:
fun (‘A’); // Неправилно обръщение. За да се пропусне int a1 = 10, трябва да е пропуснат char c = ‘X’

Ако една функция се използва често, вместо непрекъснато използване на механизма на извикването й, по-ефективно е в мястото на обръщенията към функцията да се копира директно нейното тяло. Ще припомним, че точно така се заместват макросите с аргументи и това е тяхното основно предимство. Например, ако има следната дефиниция на макроса max

```
#define max(a,b) ((a) > (b) ? (a) : (b))
```

който се използва в програмата като

```
int m, x, y;
…
m = max(x,y);
```

в първата фаза на компилацията на програмата обръщението към макроса max(x,y) ще се замести със заместващия низ от дефиницията му ((x) > (y) ? (x) : (y)). При обръщението формалните параметри a и b в заместващия низ се заместват с актуалните x и y. Външно използването на max(x,y) много наподобява на обръщението към функция.

Ако искаме обръщението към една функция да се замести по време на компилация с операторите на тялото ѝ, тя трябва да се обяви чрез ключовата дума inline, че ще бъде вградена. Ключовата дума inline се използва в дефиницията на функцията:

```
inline void fun(int a, char c) {
тяло на fun()
}
```

Не е сигурно, че функция, обявена за inline, действително ще се реализира като вградена. Това зависи от конкретния компилатор. Механизмът на вграждане на функции трябва да се балансира с нежеланото увеличение на паметта, която се заема от програмата. Той е подходящ за използване с функции с малък код и чести обръщения към тях.

35. Рекурсивни функции

Функция, която се обръща пряко или косвено към себе си се нарича **рекурсивна.** Под косвено обръщение разбираме следното: една функция F_1 се обръща към функция F_2, която се обръща към F_1.

При всяко рекурсивно обръщение в програмния стек се разпределя памет за параметрите и локалните променливи на функцията.

По-общо един обект е рекурсивен, ако частично се съдържа в себе си или е дефиниран с помощта на себе си. Езикът С поддържа две езикови конструкции, които позволяват да се реализират рекурсивни алгоритми – това са рекурсивни функции и структури с рекурсия.

Пример:

n! = 1. 2. … .n, n > 0,

n! = 1, n = 0; - итеративна дефиниция (не съдържа рекурсия)

n! = n*(n-1)!, n > 0
n! = 1, n = 0; - рекурсивна дефиниция

```
#include <iostream.h>
int fact (int n)
 { if (n==0) return 1;
   else return n*fact(n-1);
 }
void main ()
{int k;
 cout << “Въведете к ”  << endl;
cin >> k;
cout << k  << “! = “ << fact(k) << endl;
}
```

В общия случай рекурсията позволява да се пишат по-елегантни алгоритми и позволява лесно решаване на по-сложни задачи. Рекурсията повишава ефективността на програмисткия труд.

Обикновено рекурсивните програми са по-неефективни, изискват повече ресурси (памет, време). От итерация към рекурсия се преминава лесно, обратното е доста сложно (но винаги е възможно).
Реализиране на обръщението fact (3) - при това обръщение рекурсивно се извършват обръщенията fact(2), fact(1), fact (0). При изпълнението на fact (0) няма да последва рекурсивно обръщение, тъй като n става равно на 0. Функцията ще върне стойност 1, след което ще приключи изпълнението на fact (0) и паметта разпределена в стека за n при това изпълнение ще се освободи.
След това се връщаме на обръщението fact (1), което връща стойност 1, паметта за n за това обръщение се изтрива от стека и отново се връщаме на по-горното обръщение и т.н. fact (3) връща стойност 6.

Общо правило: когато реализираме рекурсия трябва да има гранични условия (условия за излизане от рекурсията), които <u>задължително</u> се достигат. В противен случай се получава зацикляне и програмата обикновено завършва с препълване на програмния стек. След края се освобождава паметта в програмния стек. Когато имаме рекурсия е добре да се минимизира броя на формалните параметри и локалните променливи.
Разгледаното обръщение към функция се нарича пряко рекурсивно обръщение (пряка рекурсия), тъй като се извършва в самата функция. В езика С е възможно да осъществим и непряко рекурсивно обръщение (непряка, косвена рекурсия) чрез друга функция. Това изглежда примерно така:

```
void post (int a, int b, int c);
int fun (char a, char b)
{
……..
post (x, y, z);
……..
}
void post (int a, int b, int c)
{
………
d = fun (a, b);
………
}
```

Рекурсивна функция за x^n, $n > 1$:

```
double sqr_or_pow (double x, int k = 2)
 { if (k==2) return x*x;
   else return x*sqr_or_pow(x, k-1);
 }
```

sqr_or_pow(y) – y^2, k – параметър по подразбиране.
sqr_or_pow(y, 3) - y^3 – само последните формални параметри могат да бъдат по подразбиране.
Във функцията сме използвали инициализиране на формален параметър с константа. Това позволява да не се указва фактически параметър (стойността на параметъра е по подразбиране), ако стойността му съвпада със съответната инициализирана стойност за параметъра. Инициализираните формални параметри трябва да стоят на последно място в списъка. По този начин получаваме още един начин за използване на функция с променлив брой параметри.

Проблемът при реализацията на рекурсията в горната програма е, че се губи част от ефективността от това, че променливата x не се променя, но влиза в стека при всяко рекурсивно обръщение. Обикновено всички вътрешни променливи, които се използват само преди рекурсивното обръщение могат да се направят външни с цел да се повиши ефективността на програмата.
Друг пример за рекурсия: рекурсивна програма, която чете низ от символи (до натискане на Enter) и го извежда в обратен ред.

```
#include <iostream.h>
#include <stdio.h>
void niz()
 { int ch;
   if ( (ch = getchar ()) != '\n' )
    { niz();
      putchar (ch);
    }
 }
void main()
{
niz();
}
```

При първото обръщение към niz в програмния стек ще се разпредели памет за локалната променлива ch. Ако въведем низ “abc“ и натиснем Enter, на ch ще се присвои ‘a’, ‘a’ ≠ ‘\n’ и след това рекурсивно ще се извърши второ обръщение. При второто обръщение отново ще се разпредели памет за ch и ще му се присвои стойност ‘b’ ≠ ‘\n’. След това ще последва трето рекурсивно обръщение, при което на новата ch ще се присвои стойност ‘c’ ≠ ‘\n’ и ще последва рекурсивно четвърто обръщение. Тук вече на новата ch ще се присвои стойност ‘\n’ и обръщението ще приключи, като се освободи паметта за ch. След това се връщаме на третото обръщение, където ch е ‘c’ и я отпечатваме, след което освобождаваме паметта за ch и приключваме третото обръщение. Връщаме се на второто обръщение, отпечатваме ‘b’, след това на първото обръщение, отпечатваме ‘a’ и изпълнението на функцията приключва.

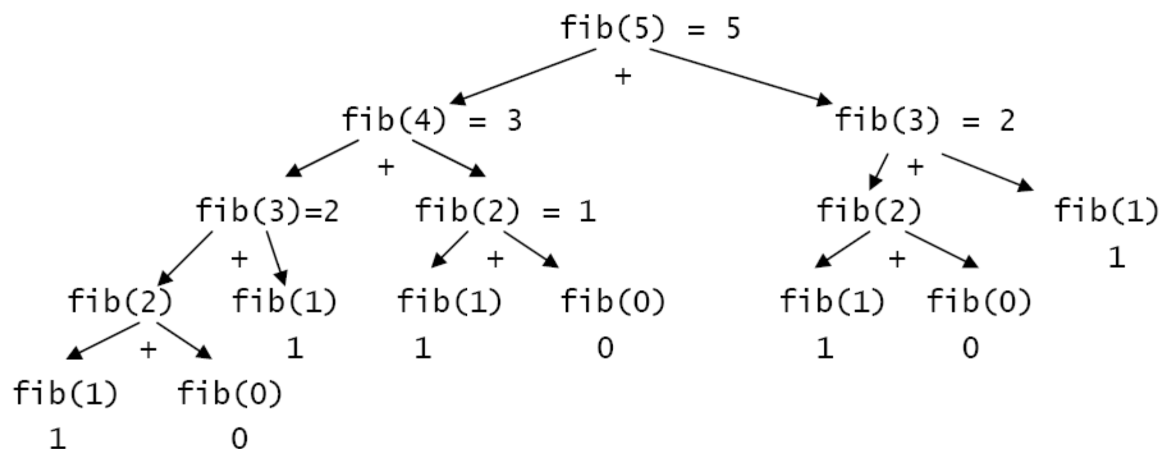
Редицата от числа на Фибоначи (Fibonacci)

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

започва с 0 и 1 и има свойството, че всеки следващ елемент от редицата е сумата на двата предишни елемента.
Редицата от числа на Фибоначи може да се определи рекурсивно по следния начин:

fib(0) = 0
fib(1) = 1
fib(n) = fib(n - 1) + fib(n - 2)

Намирането на fib(5) поражда последователност от обръщения, която изглежда така:



Такава рекурсия се нарича разклонена или дървовидна рекурсия.

В този случай намирането на fib(5) генерира дървовиден процес. Операцията + се отлага. Забелязваме много повтарящи се изчисления, например fib(0) се пресмята 3 пъти, fib(1) – 5 пъти, fib(2) - 3 пъти, fib(3) – 2 пъти, което илюстрира неефективността на този начин за пресмятане.

36. Стандартни функции за обработка на низове

В езика C няма голям набор от средства за обработване на низове и обикновено обработката става със стандартните функции. Техните прототипи са описани в заглавния файл string.h. Някои от тези функции имат параметри с тип данни size_t. Този тип се определя в заглавния файл stddef.h (от стандартната библиотека на C) като беззнаков цял тип unsigned int.
При работа с тези функции задължение на програмиста е да задели достатъчно памет за обработваните низове.

По-често използвани функции:
char *strcat (char *str1, const char *str2); конкатенация на низове
Функцията strcat добавя копие на низа, сочен от str2, в края на низа, сочен от str1 и връща указател към конкатенирания низ (str1). Първият символ на str2 се записва върху нулевия байт на str1. Тя реализира конкатенация на низове. Функцията не променя str2, тъй като той е описан като константен формален параметър.

Тази функция се използва и в следната модификация:
char *strncat(char *str1, const char *str2, size_t n);
В този случай функцията копира не повече от n символа от низа, сочен от str2 в края на низа, сочен от str1 и добавя нулев байт. Първият символ на str2 се записва върху нулевия байт на str1. Тя отново връща указател към получения низ (str1).

int strcmp (const char *str1, const char *str2); сравняване на низове
Функцията strcmp сравнява два низа посимволно и връща цяло число: 0 – ако съвпадат, < 0 – ако низът, сочен от str1, е по-малък от низа, сочен от str2, > 0 – ако низът, сочен от str1, е по-голям от низа, сочен от str2. Сравняването на низовете започва с първия символ от двата низа и продължава със следващите съответни символи, докато те станат различни или докато не се стигне до края и на двата низа.
Тази функция също има модификация:
int strncmp (const char *str1, const char *str2, size_t n);
Тя сравнява не повече от n символа от двата низа.

Пример: Имаме масив от указатели към редове с текст. Нека броят на редовете е n. Трябва редовете да се сортират по азбучен ред. Самите низове не си разменят стойностите, само указателите към тях си разменят стойностите.

```
void sort ( char *s[], int n)
 { int i, change;
   char *temp;
   do {
     change = 0;
     for (i = 0; i < n-1; i++)
      if ( strcmp (s[i], s[i+1]) > 0 )
        { temp = s[i];
          s[i] = s[i+1];
          s[i+1] = temp;
          change = 1;
        }
   } while (change);
 }
```

char *strcpy (char *str1, const char *str2); копиране на низове
Функцията strcpy копира низа, сочен от str2, в str1 включително и нулевия байт. Функцията връща резултат указателя str1.
Тази функция може има модификация:
char *strncpy (char *str1, const char *str2, size_t n);
Тя копира не повече от n символа от низа, сочен от str2, в str1 и връща указателят str1. Низа str1 може да не завършва с нулевия байт, ако дължината на str2 е по-голяма или равна на n.

Пример:
char str1[10];
char *str2 = “abcdefghi”;
strncpy (str1, str2, 3); //str1 няма да завършва с нулев байт
// и той трябва да се добави str1[3] = ‘\0’;
char *str2 = “ab”;
strncpy (str1, str2, 3); //str1 ще завършва с нулев байт

char *strchr (const char *str, int ch);
Тази функция преглежда низа от ляво надясно и връща указател към първия срещнат символ ch. Ако не открие такъв символ връща нулев указател. Може да се търси и нулев байт.
size_t strlen (const char *str);
Тази функция връща цяло число без знак, което е броят на символите в низа, сочен от str без да се брои нулевия байт в края на низа.
char *strbrk(const char *str1, const char *str2)
Функцията преглежда отляво надясно str1 за първо срещане на който и да е символ от str2 и връща указател към него, ако такъв символ не е открит, връща нулев указател.
char *strstr (const char *str1, const char *str2);
Тази функция връща указател към първото срещане на подниза, сочен от str2, в низа сочен от str1. Ако низът сочен от str2 не е подниз на низа, сочен от str1, функцията връща нулев указател.