Структури от Данни и Програмиране (СДП) за специалности Приложна Математика и Статистика, I курс

1. Вход и изход на потоци. Общи сведения.

Стандартните библиотеки в C++ имат разширен набор от средства за вход и изход.
В C++ са предвидени средства за съхраняване на типа на данните при вход и изход. Всяка операция за вход и изход в C++ се изпълнява по начин, който зависи от типа на данните. Ако за даден тип данни е била дефинирана функция за вход и изход, то именно тя автоматично се извиква при обработване на този тип данни. Ако няма съответствие между типа на данните и функцията за обработка на този тип данни, компилаторът дава съобщение за грешка. По такъв начин несъответстващите данни не могат да проникнат в системата, както това може да се случи в C, което от своя страна води до възникване на трудно откриваеми грешки.
C++ дава възможност за стандартна обработка на входа и изхода както за вградените типове, така и за типове, дефинирани от потребителя.

В C++ се извършва вход и изход на поток от байтове, т.е. на последователност от байтове.
Входът и изходът могат да бъдат форматирани или неформатирани, т.е. с преобразуване или без преобразуване.
При неформатиран вход и изход се постига максимална ефективност, когато се обработват файлове с голям обем.
При форматиран вход и изход байтовете се групират в такива елементи от данни, като цели числа, реални числа, символи, низове, а също данни от типове, определени от потребителя. Това не е ефективно при обработване на файлове с голям обем.
Библиотеката за вход и изход от потоци е iostream. Нейният интерфейс е разбит на няколко заглавни файла:

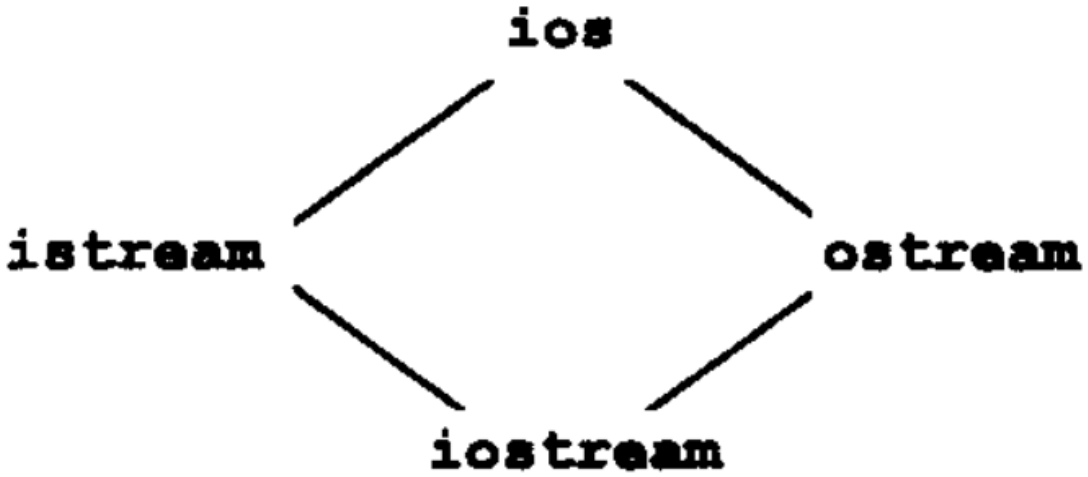
- заглавният файл iostream.h съдържа основната информация, която е необходима за всички операции за вход и изход. Предвидени са средства за форматиран и неформатиран вход и изход. Този заглавен файл включва обектите:
  cin – съответства на стандартния поток за вход;
  cout – съответства на стандартния поток за изход;

cerr – съответства на небуфериран поток за извеждане на съобщения за грешки;
clog – съответства на буфериран поток за извеждане на съобщения за грешки;

- заглавният файл iomanip.h съдържа информация, полезна за обработката на форматиран вход и изход с помощта на така наречените параметризирани манипулатори на потока;
- заглавният файл fstream.h съдържа информация за извършване на операции с файлове;
- заглавният файл strstream.h съдържа информация за използване на форматиран вход и изход в паметта. Това прилича на обработката на файлове, но операциите за вход и изход се извършват със символни масиви, а не с файлове;
- заглавният файл stdiostream.h включва необходимата информация за програми, които използват за изпълнение на операциите за вход и изход средства от C и C++.

Програмите на C++ могат да използват също други библиотеки, свързани с вход и изход, в които са предвидени например такива

специфични за системата възможности, като управляване на специални устройства за вход и изход на аудио и видео данни. Библиотеката iostream съдържа много класове за извършване на широк спектър операции за вход и изход. Класът istream поддържа операциите по въвеждане от потоци. Класът ostream поддържа операциите по извеждане в потоци. Класът iostream поддържа двата типа операции: и вход, и изход на потоци.
Класът istream и класът ostream са производни класове, които пряко наследяват базовия клас ios. Класът iostream е производен клас, който множествено наследява класовете istream и ostream. Структурата на наследяването на класовете за потоците за вход и изход е следната:



Предефинирането на операции позволява удобно записване на операциите за вход и изход.
Операцията за изместване вляво << е предефинирана за означаване на изход в поток и се нарича операция за извеждане в поток.
Операцията за изместване вдясно >> е предефинирана за означаване на вход от поток и се нарича операция за въвеждане от поток.

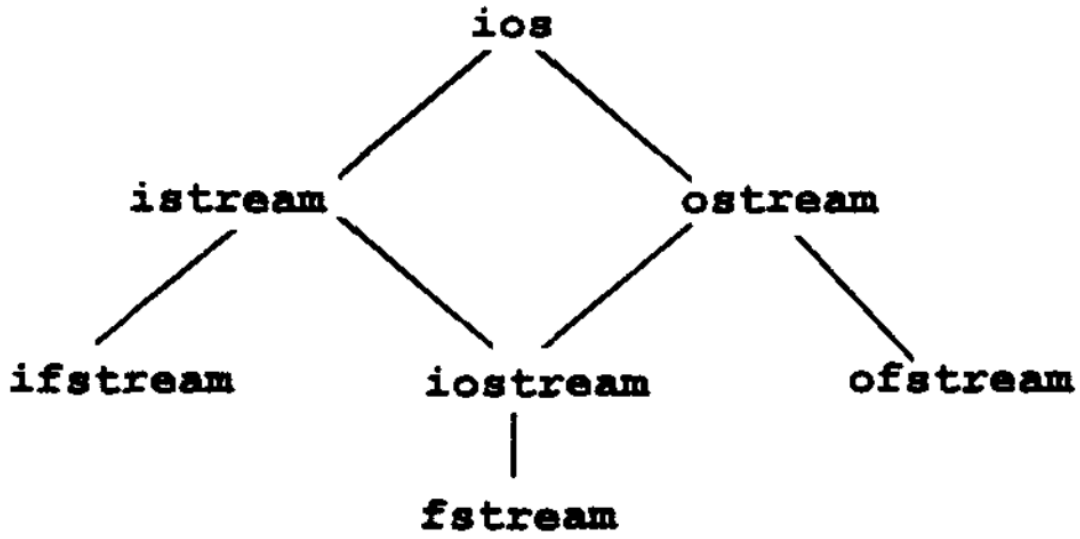
Тези операции се прилагат към обектите на стандартните потоци cin, cout, cerr, clog и също се използват с обекти на потоци, чийто тип е дефиниран от потребителя.
Обектът cin е от клас istream и е свързан със стандартното устройство за вход (обикновено клавиатурата).
Обектът cout е от клас ostream и е свързан със стандартното устройство за изход (обикновено екрана на монитора).
Операциите за въвеждане от поток и извеждане в поток определят типа на използваните данни.
Ако типът не е предвиден, т.е. операциите не са предефинирани за този тип, се включват различни флагове за грешки, с помощта на които потребителя може да определи дали операциите за въвеждане и извеждане са били успешни.
Обектът cerr е от клас ostream и е свързан със стандартното устройство за извеждане на съобщения за грешки. Изведеният поток от данни за обекта cerr е небуфериран, т.е. всяка операция за извеждане в cerr води до незабавно появяване на изведеното съобщение.

Обектът clog е от клас ostream и също е свързан със стандартното устройство за извеждане на съобщения за грешки. Изведеният поток от данни за обекта clog е буфериран, т.е. всяка операция за извеждане в clog води до съхраняване на изведеното в буфер, докато той не бъде запълнен изцяло или докато съдържанието на буфера не бъде изведено принудително.

При обработка на файлове в C++ се използват следните класове:

- клас ifstream, който изпълнява операции за вход от файлове;
- клас ofstream, който изпълнява операции за изход във файлове;
- клас fstream, който е предназначен за операции при вход и изход от файлове.

Класът ifstream наследява istream, класът ofstream наследява ostream и класът fstream наследява iostream. Структурата на наследяването на класовете за потоците за вход и изход е следната:



В повечето системи в пълната йерархия на класовете се съдържат още много други класове, които се използват при вход и изход на потоци, но разгледаните класове предоставят широки възможности, достатъчни за повечето програмисти.

2. Форматиран изход на потоци. Неформатиран вход и изход.

Класът ostream в C++ дава възможност за реализиране на форматирано и неформатирано извеждане в поток на следните данни:

- извеждане на данни от стандартни типове с помощта на операцията за извеждане в поток (<<);
- извеждане на символи с помощта на функцията-елемент put;
- неформатирано извеждане с помощта на функцията-елемент write;
- извеждане на цели числа в десетичен, осмичен и шестнадесетичен формат;
- извеждане на стойности с плаваща запетая с различна точност, с указание за извеждането на десетичната точка, в експоненциален формат или във формат с фиксирана точка;
- извеждане на данни с изравняване относно някаква граница на полето с указана ширина;

- извеждане на данни с полета, запълнени със зададени символи;
- извеждане на букви в горен регистър в експоненциалния формат и при извеждане на шестнадесетични числа.

Операцията за изместване наляво стандартно е предефинирана за извеждане на данни от вградени типове, за извеждане на низове и за извеждане стойността на указатели. Тази операция може да бъде предефинирана в програмата за извеждане на данни от потребителски типове. Възможно е слепване на операцията за извеждане в поток, тъй като тя връща псевдоним на левия си операнд и е ляво-асоциативна.

Преминаването на нов ред става с помощта на управляващия символ ‘\n’ или с помощта на манипулатора endl. Изпълнението на endl освен това води до незабавно извеждане на съдържанието на буфера. Това може да стане и с манипулатора flush:

cout << flush;

Свързване на изходния поток с входния

Интерактивните приложения обикновено включват класът istream за входа и класът ostream за изхода. Когато на екрана се появи покана за вход, потребителят въвежда съответните данни. Очевидно поканата за вход трябва да се появи преди осъществяване на операцията за вход. При буферизация на изходните данни тяхното извеждане на екран се осъществява само когато буферът се окаже пълен, когато изходът явно се изпълнява от програмата или автоматично в края на програмата. В езика C++ има функция-елемент tie, която се използва за синхронизация, т.е. свързване на изпълнението на операциите с потоци за вход и за изход. Функцията tie гарантира, че изведеното ще се покаже преди следващото въвеждане, което е много важно при създаване на интерактивни приложения. За да се синхронизират потока за вход inputStream от клас istream и потока за изход outputStream от клас ostream се използва следния оператор:

```
inputStream.tie (&outputStream);
```

Тази връзка се разпада със следното извикване:
inputStream.tie (0);
В C++ автоматично е създадена такава връзка между cin и cout, т.е. автоматично се извиква cin.tie (&cout).

Извеждане на променливи от тип char *

Операцията << може да се използва за извеждане на данни от тип char * като символен низ, който завършва с нулев байт (‘\0’).
Например:
char *string = “Проверка”;
cout << string << endl;
Чрез явно преобразуване на типа на указателя string към void * можем да изведем стойността на самия указател string, т.е. адреса на първия символ на низа:
cout << (void *) string << endl;
Това се отнася за всеки указател към променлива, ако програмистът иска да изведе нейния адрес.

Извеждане на символи с помощта на функцията-елемент put; слепени извиквания

Функцията-елемент put извежда един символ, например оператора
cout.put (‘A’);
ще изведе символа ‘A’. Извикванията на put могат да бъдат слепени – например:
cout.put (‘A’).put (‘B’);
Това е възможно, тъй като операцията ‘.’ е ляво-асоциативна и функцията-елемент put връща псевдоним на обекта, за който е била извикана. Аргументът на функцията put може да бъде произволен израз, чиято стойност е ASCII код на символ, например:
cout.put (65);
ще изведе символа ‘A’.

Неформатиран вход-изход с помощта на функциите read, gcount и write

Неформатиран вход и изход се извършва с помощта на функциите-елементи read и write.

Функцията-елемент read от класа istream въвежда определен брой байтове в символен масив от паметта. Тези байтове не се преобразуват. Ако са прочетени по-малък брой символи, се включва флага failbit.
Функцията-елемент write от класа ostream извежда определен брой байтове от символен масив в паметта, включително и нулевите байтове. Изведените байтове не се преобразуват.
Функцията-елемент gcount от класа istream връща броя на символите, прочетени от последната операция за вход.
Примерна програма:

```
#include <iostream.h>
const int SIZE = 80;
void main ()
 { char buffer[SIZE];
   cout << "Въведете изречение: ";
   cin.read (buffer, 20);
   cout.write (buffer, cin.gcount());
   cout << endl;
```

```
}
```
Резултат:
Въведете изречение:
Използване на функциите read, gcount и write
Използване на функци

3. Форматиран вход на потоци.

Входният поток е разбит на елементи. Между елементите има един или повече празни символи (интервал, табулация, символ за нов ред).
Операцията за въвеждане от поток (>>) връща нулева стойност, т.е. false, ако срещне в потока символа за край на файла. В противен случай тя връща псевдоним на левия си операнд, т.е. на обекта, за който тя е била извикана – това позволява слепване на операцията за въвеждане.

Операцията за въвеждане включва флага failbit, т.е. задава му стойност 1, при въвеждане на данни от неправилен тип и включва флага badbit при неуспешно завършване на операцията.
Типична грешка е да се въвеждат данни от поток от клас ostream или да се извеждат данни в поток от клас istream.
Много често операцията за въвеждане от поток се използва като условие в оператор за цикъл.
Примерна програма:

```
#include <iostream.h>
#include <iomanip.h>
void main ()
 { double grade, highestgrade = -1;
   cout << "Въведете оценка или Ctrl+Z за край: ";
   while (cin >> grade)
    { if (highestgrade < grade) highestgrade = grade;
      cout << "Въведете оценка или Ctrl+Z за край: ";
    }
   cout << endl << "Най-високата оценка е: " << setprecision(2) <<
```

```
        highestgrade << endl;
 }
```

Изразът cin >> grade може да бъде използван като условие, тъй като производният клас istream наследява от базовия клас ios предефинирана операция за преобразуване на тип поток в указател от тип void *, при това стойността на указателя е 0 (NULL), ако е възникнала грешка при опит за въвеждане на данни или ако е стигнат символ за край на файла. За да може да използва израз от тип void * като условие, компилаторът извършва вградено преобразуване на израза към целочислен тип.

Функции-елементи get и getline

Функцията-елемент get има три варианта:

- без параметри – функцията въвежда един символ от указания поток, включително и празните символи (‘ ‘, ‘\t’, ‘\n’) и връща този символ като резултат. Връща EOF, ако се достигне край на файла.

Пример:
int c;
cout << “Въведете низ и Ctrl+Z за край: “;
while ((c = cin.get()) != EOF) cout.put (c);

- с един параметър от тип char – функцията въвежда един символ от входния поток, включително и празните символи и го присвоява на символния аргумент. Тя връща 0, ако се достигне символ за край на файла, а в останалите случаи връща псевдоним на обекта за който е извикана.
- с три параметъра – указател от тип char *, максимален брой символи и ограничител, който по премълчаване е ‘\n’. Този вариант на get чете символи от входния поток докато техният брой стане с 1 по-малко от зададения брой или докато срещне ограничителя. Функцията автоматично добавя нулев байт към низа. Ограничителят не се записва в символния масив, а остава във входния поток.

Пример:

```
#include <iostream.h>
const int SIZE = 80;
void main ()
 { char buffer1[SIZE], buffer2[SIZE];
   cout << "Въведете изречение: ";
   cin >> buffer1;
   cout << buffer1 << endl;
   cin.get(buffer2, SIZE);
   cout << buffer2 << endl;
 }
```

Резултат:

```
Въведете изречение:
Сравняване на въвеждане на низ с помощта на cin и cin.get
Сравняване
 на въвеждане на низ с помощта на cin и cin.get
```

Функцията-елемент getline има същите параметри като третия вариант на get и действа по същия начин с тази разлика, че тя чете символа ограничител и го премахва, т.е. не го записва в символния масив.
Пример:
char buffer[80];
cout << “Въведете изречение: “;
cin.getline (buffer, SIZE);
cout << buffer << endl;
Резултат:
Въведете изречение:
Използване на функция-елемент getline
Използване на функция-елемент getline
Функцията-елемент ignore приема два параметъра – брой символи и символ ограничител. Тя пропуска зададения брой символи (по премълчаване 1) или завършва своята работа при откриване на ограничителя, при това тя го премахва от входния поток. По премълчаване ограничителят е EOF.

Функцията-елемент putback връща символ обратно във входния поток. Обикновено преди това символът е прочетен с get. Функцията е полезна за приложения, които преглеждат входния поток с цел търсене на запис, който започва с даден символ. Когато този символ е въведен, приложението го връща обратно във входния поток, за да бъде включен в данните, които ще се въведат по-нататък.
Функцията-елемент peek връща поредния символ от входния поток, включително и празните символи, но не го премахва от потока.

4. Манипулатори на потоци.

Манипулаторите на потоци се използват за форматиране. Те позволяват да се изпълняват следните операции:

- задаване на минимална ширина на поле;
- задаване на точност за числа с плаваща точка;
- включване и изключване на флагове на формата;

- задаване на запълващ символ на поле;
- изчистване на буферите на потоци;
- извеждане на символ за нов ред и изчистване на буфера;
- извеждане на нулев байт;
- пропускане на символите разделители във входния поток.

Основата на целите числа в поток се задава с манипулаторите:

- dec (decimal) – за десетични цели числа;
- oct (octal) – за осмични цели числа;
- hex (hexal) – за шестнайсетични цели числа.

По премълчаване целите числа се интерпретират като десетични. Основата на целите числа в поток може да се зададе и чрез манипулатора setbase, който има един цял параметър, приемащ стойности 8, 10, 16, задаващи съответната основа. Тъй като setbase има параметър, той се нарича **параметризиран манипулатор.** setbase не е дефиниран във всички реализации. Използването на такива манипулатори изисква включването на заглавния файл iomanip.h. Основата на бройната система остава една и съща, докато не бъде променена отново.

Пример:

```
int n = 20;
cout << n << endl                  //20
     << hex << n << endl           //14
     << oct << n << endl           //24
     << dec << n << endl;          //20
```

С помощта на манипулатора setprecision или функцията-елемент precision се задава точността на извежданите числа с плаваща точка, т.е. броя на изведените цифри в дробната част. По премълчаване точността е 6 цифри. Зададената точност действа за всички следващи операции за извеждане, докато не бъде явно зададена отново. Функцията-елемент precision без параметри връща текущата стойност на точността.

Пример:

```
#include <iostream.h>
#include <iomanip.h>
#include <math.h>
```

```
void main ()
 { double root2 = sqrt (2);
   for (int i = 0; i <= 9; i++)
     { cout.precision (i);
       cout << root2 << endl;
     }
 }
```

Програмата ще върне:

```
1.
1.
1.4
1.41
1.414
1.4142
1.41421
1.414214
1.4142136
1.41421356
```

Ще отбележим, че при извеждането на числа с плаваща точка се извършва закръгляне. Ако използваме манипулатора setprecision тялото на цикъла трябва да изглежда така:
cout << setprecision (i) << root2 << endl;
Ширината на полето, т.е. броят на символните позиции, в които стойността се извежда или броя на символите, които се въвеждат, се задава с функцията-елемент width от класа ios. Ако обработваната стойност има по-малко символи от зададената ширина на полето, то в излишните позиции се извежда запълващ символ (по премълчаване ‘ ‘), при това подравняването по премълчаване е отдясно. Ако броят на символите в обработваната стойност е по-голям от зададената ширина на полето, то излишните символи не се отчитат и стойността ще бъде напълно изведена. Зададената ширина на полето влияе само на следващата операция за извеждане или въвеждане – след това ширината на полето неявно се задава 0, т.е. то ще бъде толкова широко, колкото е необходимо. Функцията width без параметри връща текущата ширина на полето.

За задаване на ширина на полето може да се използва и параметризирания манипулатор setw. При въвеждане на низ се четат поне с един символ по-малко от зададената ширина, за да може да се разположи нулев байт в края на въвеждания низ.
Пример:

```
#include <iostream.h>
void main ()
 { int w = 4;
   char string[10];
   cout << "Въведете изречение: "; cin.width (5);
   while (cin >> string)
     { cout.width (w++);
       cout << string << endl;
       cin.width (5);
     }
 }
```

Ако въведем изречението: Това е проверка.
програмата ще върне

Позиции

1	2	3	4	5	6	7	8
Т	о	в	а				
				е			
		п	р	о	в		
			е	р	к	а	
							.

Потребителите могат да създават свои собствени манипулатори на потоци, включително и параметризирани. Пример:

```
#include <iostream.h>
ostream &bell (ostream &output)
 { return output << '\a'; } //извеждане на звук
ostream &ret (ostream &output)
 { return output << '\r'; } //връщане в началото на реда
ostream &tab (ostream &output)
 { return output << '\t'; } //отместване с 8 символа
```

```
ostream &endline (ostream &output)
 { return output << '\n' << flush; } //аналогичен манипулатор на endl
void main ()
 { cout << endline << 'a' << tab << 'b' << tab << 'c' << endline
          << "!!!!!";
  for (int i = 1; i <= 10; i++) cout << bell;
  cout << ret << "_______" << endline;
 }
```

Програмата връща:

```
a    b    c

_______
```

5. Флагове за състояние на формата.

Различни флагове за състоянието задават вида на форматирането, което се извършва при изпълнение на операциите за вход и изход. Определянето на стойността на флаговете се

управлява от функциите-елементи setf, unsetf, flags. Всеки от флаговете за състоянието на формата е определен като стойност от изброен тип в класа ios.
Състоянието на формата се определя от защитената данна-елемент x_flags от тип long в класа ios. Всеки флаг представлява отделен бит в x_flags – той има стойност 1, ако е вдигнат и 0 в противен случай.
За обединяване на няколко флага в една стойност от тип long може да се използва операцията поразрядно логическо или (‘|’).
Извикването на функцията-елемент flags с един параметър от тип long задава нова стойност за променливата x_flags, т.е. задава нови стойности за всички флагове за състоянието на формата, и връща нейната предишна стойност. Ако функцията бъде извикана без параметри, тя само връща текущата стойност на x_flags.
Например след извикването:
cout.flags (ios::fixed | ios::showpoint)
флаговете fixed и showpoint ще бъдат вдигнати (ще имат стойност 1), а всички останали флагове ще имат стойност 0.

Функцията-елемент unsetf изчиства, т.е. анулира указаните флагове в променливата x_flags и връща нейната предишната стойност.
Функцията-елемент setf има два варианта:
long setf (long setbits, long resetbits);
Този вариант анулира битовете resetbits в променливата x_flags и включва битовете setbits.
long setf (long setbits);
Този вариант включва битовете setbits в променливата x_flags.
Извикванията cout.flags (cout.flags() | setbits) и cout.setf (setbits) са еквивалентни.
Параметризираните манипулатори setiosflags и resetiosflags изпълняват същите действия както функциите setf и unsetf съответно.
Вдигнатият флаг skipws показва, че операцията въвеждане от поток (>>) трябва да пропуска символите разделители (‘ ‘, ‘\n’, ‘\t’).
По премълчаване skipws е вдигнат, т.е. разделителите се пропускат.

Ако флагът skipws е вдигнат, то с помощта на извикването cin.unsetf (ios::skipws) се отменя пропускането на разделителните символи. Манипулаторът ws вдига флага skipws.
Вдигнатият флаг showpoint указва, че при извеждане на числа с плаваща точка, задължително ще се покаже десетичната точка и нулите в края на дробната част на числото в съответствие със зададената точност.

Пример:

```
cout << 9.9900 << endl
        << 9.9000 << endl
        << 9.0000 << endl;
cout.setf (ios::showpoint);
cout << 9.9900 << endl
        << 9.9000 << endl
        << 9.0000 << endl;
```

Този фрагмент ще изведе:

```
9.99
9.9
9
9.99000
9.90000
9.00000
```

Флаговете left и right задават подравняване на изведеното съответно отляво и отдясно на полето. Празните позиции се заемат от запълващия символ (по подразбиране ‘ ‘). По премълчаване подравняването е отдясно.

Пример:

```
int x = 12345;
cout << setw (10) << x << endl << setw (10);
cout.setf (ios::left, ios::adjustfield);
cout << x << endl;
cout.unsetf (ios::left);
cout << setw (10) << x << endl << setw (10)
        << setiosflags (ios::left) << x << endl
        << setw (10) << resetiosflags (ios::left)
        << x << endl;
```

Фрагментът ще изведе:

```
_ _ _ _ _ 12345
12345 _ _ _ _ _
_ _ _ _ _ 12345
12345 _ _ _ _ _
_ _ _ _ _ 12345
```

Аргументът ios::adjustfield, който съдържа трите флага left, right, internal, е необходимо да се задава като параметър на функцията setf, когато тя се използва за вдигане на тези флаговете, тъй като чрез него първоначално те се изчистват, което гарантира, че след изпълнението на setf точно един от трите флага е вдигнат – това се налага, тъй като те взаимно се изключват.
Вдигнатият флаг internal показва, че знакът на числото или основата при вдигнат флаг showbase трябва да се подравнява отляво, стойността на числото да се подравнява отдясно, а в междинните позиции да се вмъкнат запълващи символи.
Вдигането на флага showpos води до извеждане на ‘+’ за положителни числа.

Пример:

```
cout << setiosflags (ios::internal | ios::showpos)
        << setw (10) << 123 << endl;
int x = 10000;
cout.setf (ios::showbase);
cout << setw (10) << x << endl;
cout.setf (ios::left, ios::adjustfield);
cout << setw (10) << x << endl;
cout.setf (ios::internal, ios::adjustfield);
cout << setw (10) << hex << x << endl;
cout.setf (ios::right, ios::adjustfield);
cout.fill ('*');
cout << setw (10) << dec << x << endl;
cout.setf (ios::left, ios::adjustfield);
cout << setw (10) << setfill ('%') << x << endl;
cout.setf (ios::internal, ios::adjustfield);
cout << setw (10) << setfill ('*') << hex << x << endl;
```

Фрагментът ще изведе:

Позиции

1	2	3	4	5	6	7	8	9	10
+							1	2	3
+					1	0	0	0	0
+	1	0	0	0	0				
0	x					2	7	1	0
*	*	*	*	+	1	0	0	0	0
+	1	0	0	0	0	%	%	%	%
0	x	*	*	*	*	2	7	1	0

Функцията-елемент fill задава запълващ символ и връща предишния такъв. Манипулаторът setfill има същото действие. Статичният елемент basefield от ios, аналогично на adjustfield, включва флаговете dec, oct, hex. Тези флагове показват как да се интерпретират целите числа – като десетични, осмични или шестнайсетични.

По аналогични причини (трите флага взаимно се изключват) при извикване на setf се указва като втори аргумент ios::basefield. Ако нито един от флаговете не е вдигнат, то по премълчаване целите числа при операция извеждане в поток се интерпретират като десетични. При операцията въвеждане от поток данните по премълчаване се обработват в тази форма, в която се посочват – целите започващи с 0 като осмични, започващи с 0x или 0X като шестнайсетични, а всички останали като десетични.

Ако за потока е зададена основата, то всички цели числа в този поток се обработват при тази основа, докато не бъде зададена нова основа.

При вдигнат флаг showbase се извежда основата на целите числа – осмичните започват с 0, шестнайсетичните с 0X или 0x.

Пример:

```
int x = 100;
cout << setiosflags (ios::showbase) << x << endl
        << oct << x << endl << hex << x << endl;
```

Фрагментът извежда:

```
100
0144
0x64
```

Флаговете scientific и fixed се съдържат в статичния елемент floatfield от класа ios. Те се използват в setf аналогично на adjustfield и на basefield. Тези флагове управляват формата за извеждане на числата с плаваща точка - вдигнатият флаг scientific води до извеждане в експоненциален формат, докато вдигнатият флаг fixed води до извеждане с фиксирана точка. Ако тези флагове не са вдигнати, стойността на числото с плаваща точка определя формата на неговото извеждане.

Пример:

```
double x = .001234567, y = 1.946e09;
cout << x <<'\t' << y << endl;
cout.setf (ios::scientific, ios::floatfield);
cout << x << '\t' << y << endl;
cout.unsetf (ios::scientific);
cout << x << '\t' << y << endl;
cout.setf (ios::fixed, ios::floatfield);
cout << x << '\t' << y << endl;
```

Фрагментът извежда:

```
0.00123457              1.946e+009
1.234567e-003           1.946000e+009
0.00123457              1.946e+009
0.001235          1946000000.000000
```

Вдигнат флаг uppercase води до извеждане на главно X и на главни шестнайсетични цифри, които са букви ('A', 'B', 'C', 'D', 'E', 'F').

Например:
cout << setiosflags (ios::uppercase) << hex << 123456789 << endl;
cout << setiosflags (ios::showbase) << hex << 123456789 << endl;
Фрагментът извежда:
75BCD15
0X75BCD15

Състоянието на потока може да бъде проверено с помощта на битове от класа ios, които са записани в защитения елемент state от тип long.
Битът eofbit за входен поток автоматично се включва, ако се срещне признак за край на файла. Потребителят може да използва функцията-елемент eof за да определи стойността на бита – тя връща стойност истина, ако eofbit е включен и лъжа в противен случай.
Битът failbit се включва, ако в потока възникне грешка при форматирането, но символи не се губят. Обикновено след тази грешка данните могат да се възстановят.

Функцията-елемент fail, аналогично на eof може да се използва за да се определи дали failbit е включен.
Битът badbit се включва при възникване на грешка, която води до загуба на данни. Обикновено след такава грешка данните не могат да се възстановят. Функцията-елемент bad може да се използва за проверка на стойността на badbit.
Битът goodbit се включва, ако нито един от битовете eofbit, failbit, badbit не е включен, т.е. ако не са възникнали грешки. Функцията-елемент good връща истина, ако goodbit е включен и лъжа в противен случай.
Функцията-елемент rdstate връща стойността на защитения елемент state, който съдържа стойностите на всички битове, определящи състоянието на потока. Върнатата стойност евентуално може да се използва за проверка на битовете eofbit, failbit, badbit, goodbit, но по-удобно средство е използването на функциите-елементи eof, fail, bad, good.

Функцията-елемент clear променя стойността на защитения елемент state от класа ios, в който са записани битовете за състоянието на потока. Обикновено тя се използва за възстановяване на потока в нормално състояние, т.е. тя изключва битовете eofbit, failbit, badbit и включва бита goodbit. Функцията има един параметър, който по премълчаване има стойност goodbit, така че например извикването cin.clear () изчиства състоянието на входния поток cin и включва бита goodbit. Оператора cin.clear (ios::failbit) включва бита failbit (и изключва всички останали битове) и той може да се използва ако програмистът срещне проблем при обработката на потребителски тип от данни със cin.
Пример:

```
int x;
cout << cin.rdstate () << ‘ ‘
        << cin.eof () << ‘ ‘
        << cin.fail () << ‘ ‘
        << cin.bad () << ‘ ‘
        << cin.good () << endl;
```

```
cin >> x; //въвеждаме символ
cout << cin.rdstate () << ‘ ‘
        << cin.eof () << ‘ ‘
        << cin.fail () << ‘ ‘
        << cin.bad () << ‘ ‘
        << cin.good () << endl;
cin.clear ();
cout << cin.rdstate () << ‘ ‘
        << cin.eof () << ‘ ‘
        << cin.fail () << ‘ ‘
        << cin.bad () << ‘ ‘
        << cin.good () << endl;
```

Фрагментът връща:

```
0 0 0 0 1
2 0 2 0 0   //след въвеждане на символа A
0 0 0 0 1
```

6. Пространства от имена.

Една програма включва много идентификатори, които са с различна област на действие. Понякога идентификатор от една област на действие може да бъде дублиран с идентификатор със същото име от друга област на действие. Дублирането на идентификатори често възниква в библиотеки от независими разработчици, в които се използват едни и същи имена за външни (глобални) идентификатори. В такива случаи обикновено се генерира грешка по време на компилация.
В стандарта на C++ се предлага решение на този проблем с помощта на **пространства от имена (namespace)**. Всяко пространство от имена определя област на действие, в която са включени идентификатори. За да се използва елемент от пространството от имена, неговото име трябва да се обяви заедно с името на пространството и бинарната операция за разрешаване на област на действие по следния начин:
име_на_пространство::елемент_на_пространство

Друга възможност е използване на оператор using. Операторът
using namespace име_на_пространство;
поставен в началото на файл, в който се използват елементите от пространството от имена, указва, че всички елементи от това пространство от имена могат да се използват във файла без да се задава името на пространството пред всеки елемент и операция за разрешаване на област за действие (::).
В стандарта на ANSI C++ дефинициите на всички класове, обекти и функции в стандартната библиотека на C++ се намират в пространството от имена std, така че в програмите, които са съобразени с този стандарт, трябва да се поставя std:: пред всички стандартни класове, обекти, функции и т.н. Друг начин е използването на оператор
using namespace std;
в началото на програмата. Това обаче се смята за лоша практика, тъй като употребата на този оператор е смислена само ако членовете на това пространство ще се използват често.

Препоръчва се чрез другата форма на using да се укажат само тези идентификатори, които ще се използват – например:
using std::cout; using std::endl;
В стандарта на ANSI C++ заглавните файлове нямат разширение ‘.h’.
Пример (за стандарта на ANSI C++):

```
#include <iostream>
using namespace std;
int myint = 98; // глобална променлива
namespace Example {
  const double PI = 3.14159;
  const double E = 2.71828;
  int myint = 8;
  void printvalues ();
  namespace Inner {  // вложен namespace
      enum years { FISC1 = 1990, FISC2, FISC3 } ;
  }
}
```

```
namespace {  // неименуван namespace
  double d = 88.22;
}
void main ()
 { cout << “d = “ << d << endl;
   cout << “Глобален myint = “ << myint << endl;
   cout << “PI = “ << Example::PI << endl;
   cout << “E = “ << Example::E << endl;
   cout << “Example myint = “ << Example::myint << endl;
   cout << “FISC3 = “ << Example::Inner::FISC3 << endl;
   Example::printvalues ();
 }
void Example::printvalues ()
 { cout << “B printvalue: “ << endl;
   cout << “myint = “ << myint << endl;
   cout << “PI = “ << PI << endl;
   cout << “E = “ << E << endl;
   cout << “d = “ << d << endl;
```

```
  cout << “Глобален myint = “ << ::myint << endl;
  cout << “FISC3 = “ << Inner::FISC3 << endl;
}
```

Резултат:
d = 88.22
Глобален myint = 98
PI = 3.14159
E = 2.71828
Example myint = 8
FISC3 = 1992

В printvalue:
myint = 8
PI = 3.14159
E = 2.71828
d = 88.22
Глобален myint = 98

FISC3 = 1992

При определяне на пространство от имена се използва следния синтаксис:
namespace име_на_пространство
 { идентификатори }
За разлика от тялото на класа и структурата, тялото на пространството не завършва с ‘;’ . Елементът myint от пространството Example има същото име като външната променлива myint, но това не води до синтактична грешка, тъй като двете променливи са в различни пространства от имена. При описанието на функцията printvalues елементите на пространството Example са достъпни непосредствено, тъй като самата функция е от това пространство.
Пространствата от имена могат да съдържат константи, променливи, функции, класове, дори вложени пространства.

Дефинирането на пространства от имена трябва задължително да става в глобалната област на действие или да бъде вложено в друго пространство.
Елементите на пространството без име са част от глобалното пространство от имена – те са достъпни непосредствено. Външните променливи също са част от това пространство. Всички елементи на глобалното пространство от имена са достъпни във всички области на действие, разположени след тяхното дефиниране. При дублиране на вътрешни идентификатори с идентификатори от глобалното пространство може да се използва унарната операция за разрешаване на област на действие. За разлика от външните променливи, елементите, които са дефинирани в тялото на пространството без име, не са достъпни за други първични файлове, т.е. пространството без име заменя спецификатора static за връзка между първичните файлове.

Възможно е да се избегне посочването на името на пространството при използването на даден елемент. Това се постига чрез оператор
using име_на_пространство::елемент_на_пространство;
поставен в началото на блока, в който този елемент се използва.
Пространствата от имена могат да имат псевдоними:
namespace ново_име_на_пространство =
старо_име_на_пространство
Типична грешка е главната функция main да се поставя в пространство от имена или да се дефинира пространство от имена в тялото на функция, тъй като пространствата от имена засягат външни променливи и област на действие файл.

7. Преобразуване на типовете.

В стандарта на C++ има четири операции за преобразуване на типовете. За предпочитане е те да се използват вместо старото преобразуване, включено в C и C++ - въпреки, че са по-малко мощни, те са по-конкретни, което дава възможност за по-точен контрол.
Друго предимство на новите операции за преобразуване е, че те имат съвършено различно предназначение за разлика от старото преобразуване, което се използва във всички случаи.
Операцията **static_cast** служи за преобразуване в случаите, в които проверката на типа се извършва по време на компилация. С нея се извършват стандартни преобразувания – например void* в char*, int в double и т.н., както и обратните им преобразувания.
Синтаксисът на операцията е:
static_cast <нов_тип> (израз)
Извършването на неразрешено преобразуване с помощта на static_cast води до синтактична грешка.

Примери за неразрешено преобразуване са:

- преобразуване на константни в неконстантни типове и обратно;
- преобразуване на типове на указатели и псевдоними в типове, които не са в отношение на открито наследяване;
- преобразуване към тип, за който няма съответен конструктор или операция за преобразуване.

Пример:

```
#include <iostream.h>
class BaseClass {
 public:
    void f () const
      { cout << “Base” << endl; }
} ;
class DerivedClass : public BaseClass {
 public:
    void f () const
     { cout << “Derived” << endl; }
} ;
```

```
void test (BaseClass *);
void main ()
 { double d = 8.22;
   int x = static_cast < int > (d);
   cout << “d = “ << d << endl
        << “x = “ << x << endl;
   BaseClass *basePtr = new DerivedClass;
   test (basePtr);
   delete basePtr;
 }
void test (BaseClass *basePtr)
 { DerivedClass *derivedPtr;
   derivedPtr = static_cast <DerivedClass*> (basePtr);
   derivedPtr -> f ();  // извикване на функцията f от класа
DerivedClass
 }
```

Резултат:
d = 8.22
x = 8
Derived

Операцията **const_cast** премахва спецификаторите const и volatile чрез преобразуване. Спецификаторът за тип **volatile** се използва, когато достъпът до обектите от този тип ще се осъществява по-начин, който не може да бъде предвиден от компилатора. Той се използва за подтискане на различни видове оптимизации.
Синтаксисът на операцията const_cast е:
const_cast <нов_тип> (израз)
Ще отбележим, че операцията const_cast не може директно да премахне спецификаторите const и volatile от тип на променлива – тя прави това само в рамките на конкретния израз.

Пример:

```
#include <iostream.h>
class ConstCastTest {
 public:
      void setNumber (int);
      int getNumber () const;
      void printNumber () const;
 private:
      int number;
} ;
void ConstCastTest::setNumber (int num)
 {  number = num; }
int ConstCastTest::getNumber () const
 {  return number; }
void ConstCastTest::printNumber () const
 { cout << “Числото след модификация: “;
   const_cast <ConstCastTest *> (this) -> number--;
   cout << number << endl;
```

```
 }
void main ()
 { ConstCastTest x;
   x.setNumber (8);
   cout << x.getNumber() << endl;
   x.printNumber();
 }
```

Резултат:
8
Числото след модификация: 7
Константната функция-елемент printNumber модифицира стойността на number с
const_cast <ConstCastTest *> (this) -> number--;
В константната функция-елемент printNumber типът на указателя this е константен тип ConstCastTest *. Операцията премахва константността на указателя this с const_cast.

Типът на този указател за останалата част от този оператор сега е ConstCastTest *. Това прави възможна модификацията на променливата number.
Операцията **reinterpret_cast** се използва за преобразуване на един тип указател към друг тип указател. Тя също се използва за преобразуване на double в int и обратно. reinterpret_cast се използва и за извършване на опасни обработки, които могат да доведат до сериозни грешки по време на изпълнение. Използването й може да доведе до това, че една и съща програма да се изпълнява различно на различни платформи. Синтаксисът на операцията е:
reinterpret_cast <нов_тип> (израз)
Пример:

```
#include <iostream.h>
void main ()
 { int x = 120, *ptr = &x;
   cout << *reinterpret_cast <char *> (ptr) << endl;
 }
```

Резултат:
x
В програмата се обявяват цяла променлива и указател.Указателят ptr се инициализира с адреса на x. В оператора
cout << *reinterpret_cast <char *> (ptr) << endl;
се използва операцията reinterpret_cast за преобразуване на ptr (тип int *) в char *. Връщаният адрес дава името на променливата x.

8. Шаблони на функции.

Възможно е дефинирането на няколко функции с еднакви имена, но различен брой и типове на параметрите. Тези функции се наричат **предефинирани**. При извикването на предефинирана функция, компилаторът избира съответния вариант като анализира броя и типа на аргументите при извикването.

Предефинираните функции могат да имат различни или еднакви типове на връщаните резултати, но задължително трябва да имат различни списъци от параметри. Предефинираните функции обикновено се използват за извършване на близки по обработка дейности над различни типове данни.
Ако за всеки тип данни трябва да се извършват идентични обработки, то по-компактно и удобно решение е използването на **шаблони на функции**. Това е една от новите възможности в C++. Програмистът трябва да напише само едно описание на шаблона на функцията, а компилаторът, изхождайки от типа на аргументите, използвани при извикването на функцията, автоматично ще генерира обектният код на функцията за тези типове.
Например може да се напише един шаблон на функция за сортиране на масив на основата на който C++, в зависимост от типа на елементите на масива, за който е извикана функцията, автоматично ще генерира отделен вариант, сортиращ масив от конкретния тип.

Шаблоните на функции, подобно на макросите, подпомагат повторното използване на програмно осигуряване, но за разлика от макросите, шаблоните на функции помагат да се отстранят много типове грешки в резултат на пълната проверка на типовете. Всяко описание на шаблон на функция започва с ключовата дума template, след която следва списък от формалните параметри на шаблона, поставен в < >. Всеки параметър трябва да се предшества от ключовата дума class или typename, използването им е еквивалентно. Например:

```
template <class T>
template <typename ElementType>  // може да се използва class вместо typename
template <class BorderType, class FieldType>
```

Формалните параметри в описанието на шаблона означават всеки вграден тип или тип, дефиниран от потребителя. Те могат да се използват при указване на типовете на параметрите на функцията, типа на връщания резултат или типа на променливите, дефинирани в тялото на функцията.

Поради тази причина формалните параметри на шаблона още се наричат параметри на типа.
След заглавието следва обикновено описание на самата функция.
Пример: програма, която извежда съдържанието на масиви от различни типове

```
#include <iostream.h>
template <class T>
void printArray (const T *array, const int count)
 { for (int i = 0; i < count; i++)
     cout << array[i] << ' ';
   cout << endl;
 }
void main ()
 { const int aCount = 5, bCount = 7, cCount = 6;
   int a[aCount] = { 1, 2, 3, 4, 5 } ;
   double b[bCount] = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 } ;
   char c[cCount] = "HELLO";
   cout << "Масив a: " << endl;
```

```
  printArray (a, aCount);  //шаблонна функция за цял тип
  cout << “Масив b: “ << endl;
  printArray (b, bCount);  //шаблонна функция за тип double
  cout << “Масив c: “ << endl;
  printArray (c, cCount);  //шаблонна функция за тип char
}
```

Резултат:
Масив a:
1 2 3 4 5
Масив b:
1.1 2.2 3.3 4.4 5.5 6.6 7.7
Масив c:
H E L L O

В шаблона на функцията printArray е дефиниран формален параметър T за типа на масива, който ще се извежда.

Когато компилаторът открие в програмата извикване на функцията printArray, той заменя T в цялата област на дефиниране на шаблона с типа на първия фактически параметър от обръщението и по този начин C++ създава шаблонна функция за извеждане на масив от указания тип. В разгледания пример шаблоните позволяват на програмиста да избегне необходимостта от написване на три отделни предефинирани функции със следните прототипи:

```
void printArray (const int *, const int)
void printArray (const double *, const int)
void printArray (const char *, const int)
```

При използване на шаблони трябва да се отчита, че програмистът може да създаде твърде много копия на шаблонни функции и шаблонни класове. За тези копия могат да са нужни значителни ресурси от памет.

Шаблонните функции и предефинираните функции са тясно свързани. Всички шаблонни функции имат едно и също име и затова компилаторът използва механизма на предефинирането за да осигури извикването на съответната функция.
Самите шаблони могат да бъдат предефинирани – могат да се дефинират други шаблони, които имат същото име на функция, но различен набор от параметри. Например шаблона на функцията printArray може да бъде предефиниран с друг шаблон, който има допълнителни параметри lowsubscript и highsubscript за определяне коя част на масива да бъде изведена.
Шаблон на функция може също да бъде предефиниран, ако се въведе друга нешаблонна функция със същото име, но с друг набор от параметри. Например шаблона на функцията printArray може да бъде предефиниран с нешаблонна функция, която извежда символен масив в подходящо табулиран вид по стълбове.

Ако шаблон се извиква с потребителски тип като параметър и този шаблон използва операции с обекти от този тип, то такива операции задължително трябва да бъдат предефинирани – в противен случай свързващият редактор ще изведе съобщение за грешка, тъй като компилаторът ще генерира извикване на съответната предефинираща функция на операция без да обърне внимание, че тя не съществува.
Компилаторът определя коя функция съответства на дадено извикване по следния начин - отначало компилаторът се опитва да намери функция, която точно съответства по име и тип на фактическите параметри в извикването. Ако при този опит компилаторът не успее, той търси шаблон на функция с указаното име в извикването, с помощта на който той може да генерира шаблонна функция с точно съответствие на типа на фактическите параметри. Важно е да се отбележи, че при търсене на шаблон не се извършва автоматично преобразуване на типовете.

Ако и този опит е неуспешен, компилаторът търси предефинирана функция. Ако съответната функция не е намерена или в една от трите стъпки са намерени няколко такива функции, компилаторът дава съобщение за грешка.

9. Информация за типа по време на изпълнение. Булев тип.

Информацията за типа по време на изпълнение (RTTI) предоставя средства да се установи типа на даден обект по време на изпълнение дори когато е известен само указател или псевдоним на този обект.
RTTI е предназначена за използване в йерархии на полиморфно наследяване (с виртуални функции). За използване на RTTI някои компилатори изискват включване на RTTI възможностите в настройките. Ще разгледаме две операции, които спадат към средствата на RTTI – dynamic_cast и typeid.

Операцията typeid приема един аргумент и връща псевдоним на константен обект от клас type_info, който представлява типа на аргумента. Самият аргумент може да бъде име на тип и тогава typeid ще осигури информация за този тип. Класът type_info има функция-елемент name, която връща името на типа като константен низ от символи, както и предефинирани операции ==, !=, с които може да се сравняват типовете на изрази. Използването на typeid в switch-подобни конструкции за определяне на типа на обект в някаква йерархия от класове е недопустимо използване на RTTI – вместо това трябва да се използват виртуални функции. Използването на typeid изисква включването на заглавния файл typeinfo.
Пример:

```
#include <iostream.h>
#include <typeinfo>
template <class T>
T Max (T val1, T val2, T val3)
 { T max = val1;
```

```
   if (max < val2) max = val2;
   if (max < val3) max = val3;
   const char *datatype = typeid (T).name();  // получаване на името
на типа, т.е. int или double
   cout << datatype << “-типовете се сравняват:\nMax = “;
   return max;
 }
void main ()
 { int a = 8, b = 88, c = 22;
   double d = 95.96, e = 78.59, f = 83.89;
   cout << Max (a, b, c) << endl;
   cout << Max (d, e, f) << endl;
 }
```

Резултат:
int – типовете се сравняват
Max = 88
double – типовете се сравняват

Max = 95.96

Операцията dynamic_cast е четвъртата операция за преобразуване на типовете в стандарта на C++. Тя се използва само с указатели и псевдоними и осигурява извършване на правилни преобразувания по време на изпълнение. За разлика от static_cast, dynamic_cast проверява дали преобразуването е допустимо и връща нулев указател, ако това не е така. Операцията dynamic_cast най-често се използва за понижаващо преобразуване на указател от базов клас към указател от производен клас.

Пример: Тук се изискват настройки за работа в режим RTTI, ProjectSetting, C/C++, Category – C++ Language, Enable Run-Time Type Information (RTTI)

```
#include <iostream.h>
const double PI = 3.14159;
class Shape {
  public :
    virtual double area () const
       { return 0; }
} ;
class Circle : public Shape {
  public:
    Circle (double r = 1)
       { radius = r; }
    virtual double area () const
       { return PI*radius*radius; }
  protected:
     double radius;
```

```
} ;
class Cylinder : public Circle {
  public:
    Cylinder (double h = 1)
      { height = h; }
    virtual double area () const
       { return 2*PI*radius*height + 2*Circle::area(); }
  private:
     double height;
} ;
void outputShapeArea (const Shape *);  // прототип
void main ()
 { Circle circle;
   Cylinder cylinder;
   Shape *ptr = 0;
   outputShapeArea (&circle);  // извежда лице на кръг
   outputShapeArea (&cylinder); // извежда лице на цилиндър
   outputShapeArea (ptr);  // опит за извеждане на лице
```

```
 }
void outputShapeArea (const Shape *shapePtr)
 { const Circle *circlePtr;
   const Cylinder *cylinderPtr;
   cylinderPtr = dynamic_cast <const Cylinder *> (shapePtr);
//привежда Shape * към тип Cylinder *
   if (cylinderPtr != 0) // ако е истина, извиква area ()
      cout << “Пълна повърхнина на цилиндъра: “
              << shapePtr -> area ();
   else {  // shapePtr не сочи към цилиндър
         circlePtr = dynamic_cast <const Circle *> (shapePtr);  //привежда
Shape * към тип Circle *
         if (circlePtr != 0)  // ако е истина, извиква area ()
           cout << “Лице на кръга: “
                   << shapePtr -> area ();
       else cout << “Нито кръг, нито цилиндър!”;
   }
  cout << endl;
```

```
}
```

Резултат:
Лице на кръга: 3.14159
Пълна повърхнина на цилиндъра: 12.5664
Нито кръг, нито цилиндър!

При опит за използване на dynamic_cast за преобразуване на указател от тип void * се получава съобщение за синтактична грешка.

Булевият тип bool е допълнение към стандарта на C++. За променливите от тип bool се разпределя един байт и те могат да приемат две стойности – вградените булеви константи true или false.
Например:
bool bool1 = true, bool2 = false;
При извеждане на променливи от тип bool true се извежда като 1, а

false като 0.
Например:
cout << bool1 << '\n' << bool2 << endl;
Променливи от тип int, double, указатели и др. могат да бъдат неявно преобразувани към тип bool – нулевите стойности се преобразуват към false, ненулевите към true.

10. Шаблони на класове

Потребителят може да използва понятието стек независимо от типа на данните, намиращи се в него. Този тип трябва да бъде зададен едва тогава, когато стекът реално се създава. Съществуват възможности за създаване на универсално програмно осигуряване, което може да се използва повторно – достатъчно е да се създаде общо описание на понятието стек и на основата на този родов клас да се създадат класове, които са специфични версии за конкретния тип данни. Тази възможност се дава в C++ от шаблоните на класове.

Шаблоните на класове често се наричат параметризирани типове, тъй като те имат един или повече параметри на тип, които определят настройката на родовия шаблон на класа за специфичен тип данни при създаване на обект.
За да се използват шаблонни класове е достатъчно веднъж да се опише шаблона на класа. Всеки път, когато се иска реализация на клас за нов тип данни, това се съобщава на компилатора и той създава първичния код за необходимия шаблон за клас. Шаблонът на класа Stack може да служи като основа за създаване на многочислени класове Stack от необходимите за програмата типове – int, double, char и т.н.
Описанието на шаблон на клас изглежда като традиционно описание на клас с тази разлика, че то се предшества от заглавието
template <списък_от_параметри_на_типовете>
Тук списъкът е от идентификатори, предшествани от ключовата дума class. Тези идентификатори означават параметрите на

типовете и те се използват в дефиницията на класа и във функциите-елементи.
В примера по-долу заглавието е template <class T> и T ще определя типа на данните-елементи в стека.

```
#include <iostream>
template <class T>
class Stack {
 public:
    Stack (int = 10);  // конструктор по премълчаване
    ~Stack () { delete [] stackPtr; }
    bool push (const T&);  // добавяне на елемент в стек
    bool pop (T&);  // извличане на елемент от стек
 private:
    int size;  // брой на елементите в стека
    int top;  // връх на стека
    T *stackPtr;  // указател към стека
    bool isEmpty () const { return top == -1; }
    bool isFull () const { return top == size-1; }
```

```
} ;
template <class T>
Stack <T>::Stack (int s)
 { size = s > 0 ? s : 10;
   top = -1;  // в началото стекът е празен
   stackPtr = new T [size];
 }
template <class T>
bool Stack <T>::push (const T&pushval)
 { if (!isFull())
   { stackPtr[++top] = pushval;
     return true;
   }
   return false;
 }
template <class T>
bool Stack <T>::pop (T &popval)
 { if (!isEmpty())
```

```
    { popval = stackPtr[top--];
      return true;
    }
  return false;
}
void main ()
 { Stack <double> doubleStack (5);
   double f = 1.1;
   cout << "Добавяне на елементи в doubleStack:\n";
   while (doubleStack.push (f))
    { cout << f << ' ';
      f += 1.1;
    }
   cout << "\nСтекът е пълен. " << f << " не е добавен.\n";
   cout << "Извличане на елементи от doubleStack:\n";
   while (doubleStack.pop (f))
     cout << f << ' ';
   cout << "\nСтекът е празен." << endl;
```

```
  Stack <int> intStack;
  int i = 1;
  cout << “Добавяне на елементи в intStack:\n”;
  while (intStack.push (i))
   { cout << i << ‘ ‘;
     i++;
   }
  cout << “\nСтекът е пълен. “ << i << “ не е добавен.\n”;
  cout << “Извличане на елементи от intStack:\n”;
  while (intStack.pop (i))
    cout << i << ‘ ‘;
  cout << “\nСтекът е празен.” << endl;
}
```

Резултат:

Добавяне на елементи в doubleStack:

1.1 2.2 3.3 4.4 5.5

Стекът е пълен. 6.6 не е добавен.

Извличане на елементи от doubleStack:

5.5 4.4 3.3 2.2 1.1
Стекът е празен.

Добавяне на елементи в intStack:
1 2 3 4 5 6 7 8 9 10
Стекът е пълен. 11 не е добавен.
Извличане на елементи от intStack:
10 9 8 7 6 5 4 3 2 1
Стекът е празен.

Всяко описание на функция-елемент, което е извън шаблона на класа се предшества от заглавието на този шаблон template <class T> . Описанието е стандартно с тази разлика, че при използване на името на класа винаги трябва да се задава списъка от параметри на типовете, заграден в < >. За разлика от шаблоните на функции, при създаване на обект на шаблонен клас изрично трябва да се посочат конкретните типове, за които ще се създава този обект. В примера по-горе, когато компилаторът срещне

Stack <double> doubleStack (5);
той ще генерира екземпляр на шаблона Stack, в който навсякъде параметъра на типа T ще бъде заменен с double.
Обработките за doubleStack и intStack са почти идентични, което дава възможност за използване на шаблон на функция:

```
template <class T>
void testStack (Stack <T>&s, T value, T increment, const char *sname)
 {   cout << “Добавяне на елементи в “ << sname << “:\n”;
   while (s.push (value))
    { cout << value << ‘ ‘;
      value += increment;
    }
   cout << “\nСтекът е пълен. “ << value << “ не е добавен.\n”;
   cout << “ Извличане на елементи от “ << sname << ”:\n”;
   while (s.pop (value))
     cout << value << ‘ ‘;
   cout << “\nСтекът е празен.” << endl;
```

```
}
```

Главната функция би изглеждала така:

```
void main ()
 { Stack <double> doubleStack (5);
   Stack <int> intStack;
   testStack (doubleStack, 1.1, 1.1, “doubleStack”);
   testStack (intStack, 1, 1, “intStack”);
 }
```

Изходът от програмата е същият, както изходът от по-горната програма.

Резултат:

```
Добавяне на елементи в doubleStack:
1.1  2.2  3.3  4.4  5.5
Стекът е пълен.  6.6  не е добавен.
Извличане на елементи от doubleStack:
5.5  4.4  3.3  2.2  1.1
Стекът е празен.
```

Добавяне на елементи в intStack:
1 2 3 4 5 6 7 8 9 10
Стекът е пълен. 11 не е добавен.
Извличане на елементи от intStack:
10 9 8 7 6 5 4 3 2 1
Стекът е празен.

11. Шаблони и нетипови параметри. Наследяване, приятелство и статични елементи.

Шаблони на класове и нетипови параметри

В шаблоните на функции и класове има възможност да се използват освен параметри за тип и обикновени или т.н. нетипови параметри. Например в разгледания шаблон на класа Stack може да се добави нетипов параметър от тип int, който указва максималният брой на елементите в стека:

template <class T, int maxelements>
При дефинирането Stack <double, 100> s1;
компилаторът ще генерира шаблонен клас от тип Stack <double, 100>.
Фактическите стойности на нетиповите параметри трябва да са константни изрази, тъй като те се обработват по време на компилация.
В скритата част на новия вариант на шаблона на класа Stack вместо stackPtr може да се дефинира масив по следния начин:
T stackHolder [maxelements];
и в този масив да се съхраняват данните на стека. Това ще отстрани загубата на време за динамично разпределяне на памет и ще изключи възможността за възникване на потенциално непоправима грешка по време на изпълнение, ако операцията new не успее да получи необходимия обем памет.
Ако за специфичен тип данни е нужен клас, който не съответства на общия шаблон на класа, то той явно може да бъде определен, като по този начин отменя действието на шаблона за този тип.

Например, ако е даден шаблон на клас:
template <class T> class MyClass
и за конкретен тип, например char *, трябва да се извършват действия, които не съответстват на този шаблон, програмистът трябва да създаде нов клас с име MyClass <char *> .

Шаблони и наследяване

Шаблоните и наследяването са свързани по следния начин:

- шаблон на клас може да бъде производен от шаблонен клас. В този случай, на всеки възможен базов клас съответства фамилия от производни класове;
- шаблон на клас може да бъде производен от нешаблонен клас;
- шаблонен клас може да бъде производен от шаблон на клас;
- нешаблонен клас може да бъде производен от шаблон на клас. В този случай на всеки възможен базов клас съответства точно един производен клас.

Шаблони и приятели

Както знаем, функции и цели класове могат да бъдат обявени като приятели на нешаблонни класове. За шаблоните на класове също могат да бъдат установени отношения на приятелство.
Приятелство може да бъде установено между шаблон на клас и глобална функция, между шаблон на клас и функция-елемент на друг клас (възможно шаблонен) или между шаблон на клас и цял клас (възможно шаблонен). Оформянето на тези приятелски отношения не е проста работа.
Примери:
Нека е дефиниран шаблон на клас X:
template <class T> class X
Ако в дефиницията на шаблона се обяви приятелска функция по следния начин: friend void f1 () ;, то функцията f1 ще бъде приятелска за всеки шаблонен клас, получен от шаблона X.

Нека в дефиницията на шаблона се обяви приятелска функция по следния начин: friend void f2 (X<T> &). Тогава за конкретен тип T, например float, приятелска за класа X<float> ще бъде само функцията f2 (X<float> &). Непосредствено преди описанието на тази функция в програмата трябва да се постави заглавието template <class T>;
Вътре в шаблона на класа може да се обяви функция-елемент на друг клас, приятелска за всеки шаблонен клас, получен от дадения шаблон. За това трябва да се използва име на функция-елемент от друг клас, името на този клас и бинарната операция за разрешаване на област на действие.
Ако в дефиницията на шаблона X е обявена приятелска функция във вида: friend void A::f3 ();, то функцията- елемент f3 от класа A ще бъде приятелска за всеки шаблонен клас, получен от шаблона X.

Нека в дефиницията на шаблона X е обявена приятелска функция във вида: friend void C<T>::f4 (X<T> &);. Тогава за конкретен тип T, например float, приятелска за класа X <float> ще бъде само функцията-елемент C<float>::f4 (X<float> &) на класа C<float>.
Ако в дефиницията на шаблона X е обявен приятелски клас по следния начин: friend class Y;, то всички функции-елементи на класа Y ще бъдат приятелски за всеки шаблонен клас, получен от шаблона X.
Ако в дефиницията на шаблона X е обявен приятелски клас по следния начин: friend class Z<T>;, то при създаване на шаблонен клас с конкретен тип T, например float, всички функции-елементи на класа Z<float> ще бъдат приятелски на шаблонния клас X<float>.

Шаблони и статични елементи

Както знаем, в нешаблонен клас едно копие на статична данна-елемент се използва от всички обекти на класа и статичната данна-елемент трябва да се инициализира в областта на действие файл.

Всеки шаблонен клас, получен от съответния шаблон на клас, има собствено копие на всяка статична данна-елемент от шаблона – всички обекти от този клас използват своя статична данна-елемент.

Както при нешаблонни класове, статичните данни-елементи на шаблонните класове трябва да бъдат инициализирани в областта на действие файл. Всеки шаблонен клас получава собствено копие на статична функция-елемент на шаблон на клас.

12. Списък. Стек. Опашка. Дек

Под структура от данни се разбира организирана информация, която може да бъде описана, създадена и обработена с помощта на програма.

За да се определи една структура от данни е необходимо да се направи:

- логическо описание на структурата, което я описва на базата на декомпозицията ѝ на по-прости структури, а също на декомпозиция на операциите над структурата на по-прости операции.
- физическо представяне на структурата, което дава методи за представяне на структурата в паметта на компютъра.

Структурите числа и символи се състоят от една компонента и се наричат прости, или скаларни.

Структури от данни, компонентите на които са редици от елементи, се наричат съставни.

Структури от данни, за които операциите включване и изключване на елемент не са допустими, се наричат статични, в противен случай - динамични.

Под **списък** ще разбираме крайно наредено множество от n еднотипни елементи $x_1, x_2, \ldots, x_n$. Ако n = 0, списъкът е празен. Ако n > 0, x_1 е първи елемент на списъка, x_n е последен елемент на списъка. Ако k > 1, казваме че x_{k-1} предшества x_k, ако k < n, казваме че x_{k+1} следва x_k.

Всеки елемент на списъка се състои от краен брой полета, като данните в тях може да са от различен тип – числови, текстови, за връзка с други елементи и др. Със списъци могат да се изпълняват разнообразни операции:

- търсене на елемент с номер k и осигуряване на достъп до съдържанието му;
- включване на нов елемент в списъка, непосредствено преди или след елемента с номер k;
- изключване на елемент с номер k от списъка;
- обединяване на два или повече списъка в един списък;

- разбиване на списък на няколко отделни списъка;
- създаване на копие на списък;
- определяне на броя на елементите на списък;
- търсене на елемент с определено съдържание на дадено поле (търсене по ключ);
- сортиране на елементите на списък във възходящ или низходящ ред по отношение на някакво поле (сортиране по ключ).

Съществуват различни начини за представяне на списъци в зависимост от вида на операциите, които трябва да се извършват над елементите им. Не съществува начин за представяне на списъци, при който всички операции над елементите на списъка и над самия списък като цяло се изпълняват еднакво ефективно. Най-често се използват списъци, при които включването и изключването на елементи се извършва преди първия или след последния елемент на списъка.

Стек ще наричаме списък, при който включването на нови елементи, както и изключването на елементи се извършва само от

едната страна на списъка, наречена **връх** на стека. Възможен е пряк достъп само до елемента, който се намира във върха на стека. Синоним на стек е термина LIFO (last in, first out – последен влязъл, първи излязъл). Примери:

- извиканата функция трябва да знае къде да върне управлението на извикващата функция, когато приключи нейното изпълнение. За целта адресът за връщане се добавя в стек, както при рекурсивно, така и при нерекурсивно обръщение;
- стек се използва при разпределяне и освобождаване на памет за автоматичните променливи;
- стек се използва от компилатора при пресмятане на изрази.

Опашка се нарича списък, при който операцията включване на елемент е допустима само за единия край на списъка, който се нарича край на опашката, а операцията изключване на елемент е допустима само за другия край на списъка, който се нарича начало на опашката. Възможен е пряк достъп само до елемента, който се намира в началото на опашката. Синоним на опашка е термина FIFO (first in, first out – първи влязъл, първи излязъл).

Примери:

- обикновено компютрите имат един процесор и ако той се използва едновременно от няколко потребители, тогава се създава опашка за заявките за ползване на процесора;
- буферите за вход и изход са организирани като опашка;
- по една компютърна мрежа се предават информационни пакети. Във всеки възел на мрежата постъпват пакети, те се подреждат в опашка и след това се разпределят един по един към тяхната крайна дестинация.

Дек се нарича списък, при който всички добавяния и изключвания на елементи могат да се извършват както в единия, така и в другия край. В практиката понякога се използват декове, при които в някой от краищата е забранено добавянето или изключването на елементи – наричат се декове с ограничен вход или изход.

13. Последователен списък и стек. Основни операции

За представяне на последователни списъци се използва последователно разпределена памет, т.е. елементите на списъка се записват в последователно разположени полета в оперативната памет на компютъра – това означава, че разполагането е от вида:
$loc(x_{k+1}) = loc(x_k) + C$
тук $loc(x_k)$ е адресът на елемента x_k, константата C най-често е дължината в байтове на един елемент от списъка.
В общия случай $loc(x_k) = L_0 + C*k$, където L_0 е адресът на хипотетичен елемент x_0, който предхожда първия елемент x_1 на списъка.
Алгоритъм за добавяне на елемент:
n++; x[n] = addvalue;
Алгоритъм за изключване на елемент:
savevalue = x[n]; n--;

Недостатъкът на тези алгоритми е, че те не отчитат случаите, при които няма достатъчно памет за добавяне на нов елемент или когато се прави опит за изключване на елемент от празен стек.
Ще ги модифицираме - нека M е максималният допустим индекс за върха на стека.
Алгоритъм за добавяне на елемент:
ако n==M, препълване; край;
ако n < M, n++; x[n] = addvalue;
Алгоритъм за изключване на елемент:
ако n==0, празен стек; край;
ако n > 0, savevalue = x[n]; n--;
Примерна програма (със средствата на C)

```
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
#define M 10
typedef float ElementType;
```

```
struct Stack {
    int t;
    ElementType StackArray[M];
} ;
void push (struct Stack *s, ElementType x)
 { if (s->t == M)
     { cout << “Препълване на стека!” << endl;
       exit (1);
     }
  s->StackArray[s->t++] = x;
}
ElementType pop ( struct Stack *s)
 { ElementType x;
   if (s->t==0)
     { cout << “Празен стек!” << endl;
       exit (1);
     }
   x = s->StackArray[--s->t];
```

```
  return x;
}
void main ()
{ Stack s; int i;
  s.t = 0;
  for (i = 1; i <= 10; i++)
    push (&s, 1.1f);
  for (i = 1; i <= 10; i++)
    cout << pop(&s) << endl;
}
```

Резултат:

```
1.1
1.1
1.1
1.1
1.1
1.1
1.1
```

1.1
1.1
1.1

14. Последователна опашка. Основни операции

При организация на последователни опашки се използват два номера на елементи f и r. f указва елемент, който се намира непосредствено преди първия елемент в опашката и r е номерът на последния елемент в опашката.
Алгоритъм за добавяне на елемент:
r = r+1, x[r] = addvalue
Алгоритъм за изключване на елемент:
f = f +1, savevalue = x[f]
Тази реализация на основните операции с опашка е твърде разточителна по отношение на памет, тъй като при добавяне и изключване на елементи, опашката се движи в паметта.

Удачно е тези алгоритми да се използват, когато опашката често става празна и в този случай запълването може да започне отново в началото на участъка от паметта, разпределен за опашката.
Друга реализация на опашка е **методът на пръстена** – при този метод се счита, че елементите образуват затворен контур в паметта с дължина M елемента – това означава, че след елемента с номер M стои елементът с номер 1. Така ако общия брой на елементите не надхвърля M-1, е възможно добавяне и изключване на елементи без това да води до недостиг на оперативна памет.
За удобство ще номерираме елементите от 0 до M-1.
Алгоритъм за добавяне на елемент:
r = (r+1) mod M, x[r] = addvalue
Алгоритъм за изключване на елемент:
f = (f+1) mod M, savevalue = x[f]
Тъй като са възможни конфликтни ситуации при добавяне на елемент към пълна опашка или при изключване на елемент от празна опашка, алгоритмите за добавяне и изключване могат да се модифицират.

Алгоритъм за добавяне на елемент:
ако (r+1) mod M == f, препълване, стоп
иначе r = (r+1) mod M, x[r] = addvalue
Алгоритъм за изключване на елемент:
ако r == f, празна опашка, стоп
иначе f = (f+1) mod M, savevalue = x[f]
Примерна програма (със средствата на C):

```
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
#define M 10
typedef float ElementType;
struct Queue {
 int r, f;
 ElementType QueueArray[M];
} ;
```

```
void pushq (struct Queue *q, ElementType x)
 { if ( (q -> r + 1) % M == q -> f)
    { cout << “Препълване!” << endl;
      exit (1);
    }
  q -> r = (q -> r + 1) % M;
  q -> QueueArray[q -> r] = x;
 }
ElementType popq (struct Queue *q)
 { if (q -> r == q -> f)
     { cout << “Празна опашка!” << endl;
       exit (1);
     }
   q -> f = (q -> f + 1) % M;
   return q -> QueueArray[q -> f];
 }
```

```
void main ()
 { struct Queue mem = { 0, 0 } ; //създаване на празна опашка
   ElementType y = 3.14f; int i;
   for (i = 1; i <= 9; i++)
     pushq (&mem, y+=1.1f);
   for (i = 1; i <= 9; i++)
     cout << popq (&mem) << endl;
 }
```

Резултат:

```
4.24
5.34
6.44
7.54
8.64
9.74
10.84
11.94
13.04
```

15. Свързан списък. Създаване на свързан списък

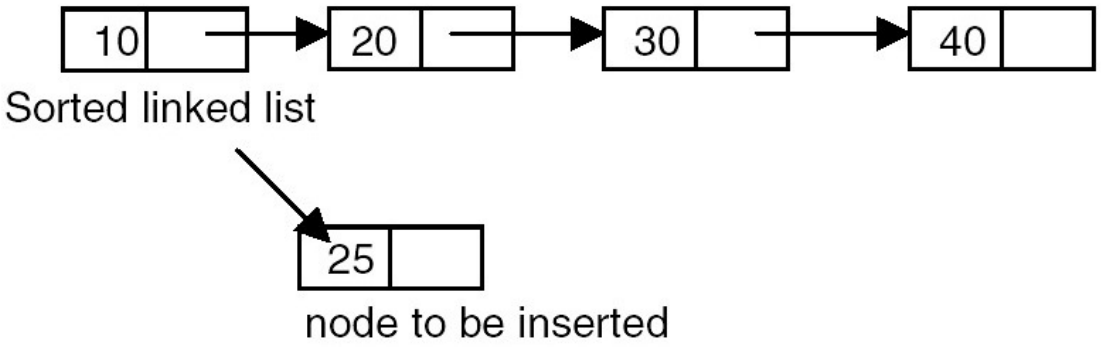
При свързаните списъци последователните елементи се съхраняват на различни места в оперативната памет, а не в последователно разположени полета. Връзката между отделните елементи се осъществява чрез указател към следващия елемент. Ако елементът е последен в списъка, стойността на този указател трябва да бъде някаква различима стойност (например NULL). За задаване на списък е достатъчен указател към първия елемент на списъка. Съпоставка между последователен и свързан списък:

- при последователния списък освен адреса на първия елемент трябва да се знае и максималният брой елементи, които ще участват в списъка. При свързан списък няма нужда предварително да се знае този брой;
- в свързан списък лесно може да се добавя или изключва елемент на произволно място вътре в списъка, докато при

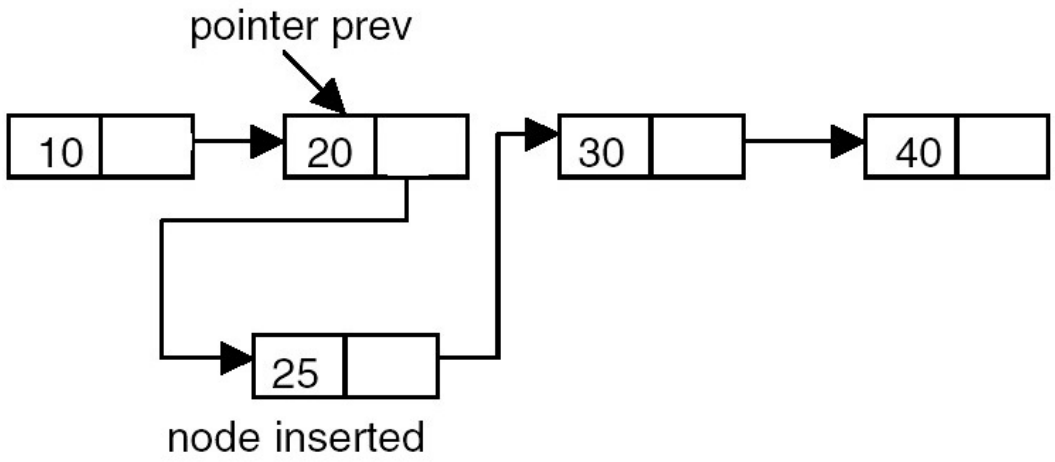
последователен списък трябва допълнително да се разместват елементите;

- обединяването на списъци е лесно изпълнима операция при свързани списъци за разлика от последователните списъци;
- при свързаните списъци за всеки елемент е нужна памет за указателите, при последователните от такива указатели няма нужда;
- при свързаните списъци достъпът до елементите е неефективен - например за да се достигне до последния елемент на списъка трябва да се премине през всички останали. Това не е така при последователните списъци – там с константна сложност можем да достигнем до произволен елемент на списъка.

В общия случай елементите на свързаните списъци се състоят от две полета – info (данните, записани в елемента) и link (указател към следващия елемент).



Before insertion



After insertion

Примерна програма (със средствата на C):

```
#include <iostream.h>
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
struct Student {
 unsigned long nomer;
 float sr_uspeh;
 struct Student *next;
} ;
void main ()
 { int n; char c;
   struct Student *first, *ps;
   n = sizeof (Student);
   first = NULL; //създаване на празен списък
   do {
     ps = (struct Student *) malloc (n);
     if (ps == NULL)
```

```
      { cout << “Няма свободна памет!”<< endl;
        exit (1);
      }
     cout << “Въведете фак. номер: “ << endl;
     cin >> ps->nomer;
     cout << “Въведете ср. успех: “ << endl;
     cin >> ps->sr_uspeh;
     ps -> next = first;
     first = ps;
     cout << “Край на въвеждането (Y/N) ?: “ << endl;
     fflush (stdin);
     c = getchar ();
  } while ( c != ‘y’ && c != ‘Y’);
}
```

Резултат:

Въведете фак. номер:

431

Въведете ср. успех:

```
3.2
Край на въвеждането (Y/N) ?:
n
Въведете фак. номер:
321
Въведете ср. успех:
4.2
Край на въвеждането (Y/N) ?:
n
Въведете фак. номер:
532
Въведете ср. успех:
5.1
Край на въвеждането (Y/N) ?:
Y
Създава се списъка:
Първи елемент на списъка:
532   5.1
```

321 4.2
431 3.2
Последен елемент на списъка:

16. Търсене на елемент по ключ. Добавяне на нов елемент в свързан списък

Ще разгледаме функция, която търси елемент в свързан списък с определен ключ и го актуализира. Ще използваме средствата на C++. Елементите ще имат тип Student.

```
struct Student {
 unsigned long nomer;
 float sr_uspeh;
 struct Student *next;
} ;
```

```
void Search (Student *first, unsigned long key, float value)
 { Student *ptr = first;
   while (ptr != NULL)
     if (ptr -> nomer == key)
       break;
     else ptr = ptr -> next;
   if (ptr != NULL) ptr -> sr_uspeh = value;
   else cout << “Няма елемент с номер “ << key << “!” << endl;
 }
```

Ще разгледаме функция за добавяне на нов елемент в свързан списък след k-тия елемент ($k \geq 1$). Ще използваме средствата на C++. Елементите ще имат тип Student.

```
void add (Student *first, int k)
 { int i = 1;
   Student *ptr = first, *ptr1;
   while ( i < k && ptr != NULL)
     { i++; ptr = ptr -> next; }
   if (ptr == NULL)
```

```
    { cout << “Няма “ << k << “ елемента в списъка!” << endl;
      return;
    }
   ptr1 = new Student;
   if (ptr1 == NULL)
    { cout << “Няма свободна памет!” << endl;
      return;
    }
   ptr1 -> next = ptr -> next;
   ptr -> next = ptr1;
   cout << “Въведете факултетен номер: “;
   cin >> ptr1 -> nomer;
   cout << “Въведете среден успех: “;
   cin >> ptr1 -> sr_uspeh;
 }
```

Ще разгледаме функция за извеждане на елементите в свързан списък.

```
void printlist(Student *first)
{ Student *ptr = first;
   while (ptr != NULL)
   {cout << ptr -> nomer << " " << ptr -> sr_uspeh << endl;
     ptr = ptr -> next;
   }
}
```

Нов вариант на функцията main с използване на функциите Search, add и printlist.

```
void main ()
 { int n; char c;
   struct Student *first, *ps;
   n = sizeof (Student);
   first = NULL; //създаване на празен списък
   do {
     ps = (struct Student *) malloc (n);
     if (ps == NULL)
      { cout << "Няма свободна памет!"<< endl;
```

```
      exit (1);
     }
    cout << "Въведете фак. номер: " << endl;
    cin >> ps->nomer;
    cout << "Въведете ср. успех: " << endl;
    cin >> ps->sr_uspeh;
    ps -> next = first; first = ps;
    cout << "Край на въвеждането (Y/N) ?: " << endl;
    fflush (stdin);
    c = getchar ();
  } while ( c != 'y' && c != 'Y');
Search(first, 1234L, 3.3);
printlist(first);
add(first,2);
printlist(first);
}
```

Резултат:

Въведете фак. номер:

```
431
Въведете ср. успех:
3.2
Край на въвеждането (Y/N) ?:
n
Въведете фак. номер:
321
Въведете ср. успех:
4.2
Край на въвеждането (Y/N) ?:
n
Въведете фак. номер:
532
Въведете ср. успех:
5.1
Край на въвеждането (Y/N) ?:
Y
Няма елемент с номер 1234!
```

```
532   5.1
321   4.2
431   3.2
Въведете фак. номер:
888
Въведете ср. успех:
5.6
532   5.1
321   4.2
888   5.6
431   3.2
```

17. Пример за обектно-ориентирана реализация на свързан списък

Програмата използва шаблон за клас List за манипулации със списък от данни от цял тип и списък от данни от реален тип. В примера се използват два шаблона на класове List и ListNode, като всеки обект на класа List е свързан списък от обекти на класа ListNode. Шаблонът на класа ListNode се състои от скрити елементи data и nextPtr. Елементът data на класа ListNode съхранява стойност от тип NodeType и този тип на параметър се предава на шаблона на класа. Елементът указател nextPtr на класа ListNode съхранява указател към следващия обект на класа ListNode в свързания списък.
Програмата дава възможност да се извършват следните операции над списъци:

- добавяне на елемент в началото на списъка - функция insertAtBeg;
- добавяне на елемент в края на списъка - функция insertAtEnd;

- изключване на елемент от началото на списъка - функция removeFromBeg;
- изключване на елемент от края на списъка - функция removeFromEnd;
- завършване обработката на списъка.

Шаблонът на класа List се състои от скритите елементи first (указател към първия обект от списъка от клас ListNode) и last (указател към последния обект от списъка от клас ListNode). Конструкторът по премълчаване задава на двата указателя начални стойности NULL. Деструкторът осигурява унищожаване на всички обекти от клас ListNode, влизащи в обект от клас List, при унищожаване на самия този обект от клас List. Основни функции-елементи на шаблона на класа List са insert AtBeg, insertAtEnd, removeFromBeg и removeFromEnd. Функцията isEmpty е предикатна функция или предикат - тя във всички случаи не променя списъка, а само установява празен ли е той (т.е. дали указателят към първия възел от списъка е нулев). Ако списъкът е празен, то се

връща 1; в противен случай се връща 0. Функцията print извежда на екрана съдържанието на списъка.

файл listnode.h

```
#ifndef LISTNODE_H
#define LISTNODE_H
template <class NodeType> class List;  //  предварителна декларация
template <class NodeType>
class ListNode {
 friend class List <NodeType>;  // List е приятелски клас
 public:
   ListNode (const NodeType &);  // конструктор
   NodeType getData () const;  //връща данните от възела
 private:
   NodeType data;  //  данни
   ListNode <NodeType> *next;  // следващ елемент на списъка
} ;
// конструктор
template <class NodeType>
```

```
ListNode <NodeType>::ListNode (const NodeType &info)
  : data (info), next (NULL) { }
template <class NodeType>
NodeType ListNode <NodeType>::getData () const
 { return data; }  //  връща копие на данните от възела
#endif
```

файл list.h

```
#ifndef LIST_H
#define LIST_H
#include <iostream.h>
#include <assert.h>
#include “listnode.h”
template <class NodeType>
class List {
  public:
     List ();
     ~List ();
```

```
    void insertAtBeg (const NodeType &);
    void insertAtEnd (const NodeType &);
    bool removeFromBeg (NodeType &);
    bool removeFromEnd (NodeType &);
    bool isEmpty () const;
    void print () const;
  private:
    ListNode <NodeType> *first;  // указател към първия елемент
    ListNode <NodeType> *last;  // указател към последния елемент
    ListNode <NodeType>* getNewNode (const NodeType &); // връща
указател към новосъздаден възел
} ;
template <class NodeType>
List <NodeType>::List () : first (NULL), last (NULL)
{ }
template <class NodeType>
List <NodeType>::~List ()
 { if (!isEmpty())
```

```
    { cout << “Изключване на обекти: “;
      ListNode <NodeType> *current = first, *temp;
      while (current != NULL)  // изтриване на останалите обекти
       { temp = current;
         cout << temp -> data << ‘\n’;
         current = current -> next;
         delete temp;
       }
    }
   cout << “Всички обекти са изключени!” << endl;
 }
template <class NodeType>
bool List<NodeType>::isEmpty () const  //  празен ли е списъкът?
 { return first==NULL; }
template <class NodeType>
ListNode <NodeType> *List <NodeType>::getNewNode
      (const NodeType &value)
 { ListNode <NodeType> *ptr = new ListNode <NodeType> (value);
```

```
   assert (ptr != NULL);
   return ptr;
 }
template <class NodeType>
void List <NodeType>::insertAtBeg (const NodeType &value)
 { ListNode <NodeType> *ptr = getNewNode (value);
   if ( isEmpty () )
    first = last = ptr;
   else { ptr -> next = first;
            first = ptr;
          }
 }
template <class NodeType>
void List <NodeType>::insertAtEnd (const NodeType &value)
 { ListNode <NodeType> *ptr = getNewNode (value);
   if ( isEmpty () )
    first = last = ptr;
   else { last -> next = ptr;
```

```
          last = ptr;
        }
 }
template <class NodeType>
bool List<NodeType>::removeFromBeg (NodeType &value)
 { if ( isEmpty () ) return false;
   ListNode <NodeType> *temp = first;
   if (first == last)
    first = last = NULL;
   else first = first -> next;
   value = temp -> data;
   delete temp;
   return true;
 }
template <class NodeType>
bool List<NodeType>::removeFromEnd (NodeType &value)
 { if ( isEmpty () ) return false;
   ListNode <NodeType> *temp = last;
```

```
   if (first == last)
    first = last = NULL;
   else { ListNode <NodeType> *current = first;
            while (current -> next != last)
             current = current -> next;
            last = current;
            last -> next = NULL;
          }
   value = temp -> data;
   delete temp;
   return true;
 }
template <class NodeType>  // извежда на екрана съдържанието на
List
void List<NodeType>::print () const
 { if ( isEmpty () )
    { cout << “Списъкът е празен!\n”;
      return;
```

```
   }
  cout << “Списък: “;
  ListNode <NodeType> *current = first;
  while (current != NULL)
   { cout << current -> data << ‘ ‘;
     current = current -> next;
   }
  cout << endl;
}
#endif
```

файл testlist.cpp

```
#include “list.h”
void instructions ();
template <class T>  // проверка на списък от числа (цели/реални)
void testList ( List <T> &listObject, const char *type)
 { cout << “Тестване на списък със стойности от тип “ << type
          << endl;
```

```
instructions ();
int choice;
T value;
do {
    cout << “? “;
    cin >> choice;
    switch (choice)
      { case 1 : cout << “Въведете “ << type << “: “;
                 cin >> value;
                 listObject.insertAtBeg (value);
                 listObject.print ();
                 break;
        case 2 : cout << “Въведете “ << type << “: “;
                 cin >> value;
                 listObject.insertAtEnd (value);
                 listObject.print ();
                 break;
        case 3 : if (listObject.removeFromBeg (value))
```

```
                    cout << value << " е изключен от списъка\n";
                else cout << "Неуспешно изключване\n";
                listObject.print ();
                break;
        case 4 : if (listObject.removeFromEnd (value))
                    cout << value << " е изключен от списъка\n";
                else cout << "Неуспешно изключване\n";
                listObject.print ();
                break;
        }
  } while (choice != 5);
  cout << "Край на тестването на списъка" << endl;
}
void instructions ()
 { cout << "Въведете номера на действието:\n";
   cout << "1 – за включване в началото на списъка\n";
   cout << "2 – за включване в края на списъка\n";
   cout << "3 – за изключване от началото на списъка\n";
```

```
  cout << “4 – за изключване от края на списъка\n”;
  cout << “5 – за завършване обработката на списъка\n”;
 }
void main ()
 { List <int> integerList;
  testList (integerList, “цяло число”);
  List <double> doubleList;
  testList (doubleList, “число с плаваща точка”);
 }
```

Резултат:

```
Тестване на списък със стойности от тип цяло число
Въведете номера на действието:
1 – за включване в началото на списъка
2 – за включване в края на списъка
3 – за изключване от началото на списъка
4 – за изключване от края на списъка
5 – за завършване обработката на списъка
? 1
```

```
Въведете цяло число: 1
Списък: 1
? 1
Въведете цяло число: 2
Списък: 2 1
? 2
Въведете цяло число: 3
Списък: 2 1 3
? 2
Въведете цяло число: 4
Списък: 2 1 3 4
? 3
2 е изключен от списъка
Списък: 1 3 4
? 3
1 е изключен от списъка
Списък:  3 4
? 4
```

```
4 е изключен от списъка
Списък: 3
? 4
3 е изключен от списъка
Списъкът е празен!
? 5
Край на тестването на списъка
Тестване на списък със стойности от тип число с плаваща точка
Въведете номера на действието:
1 – за включване в началото на списъка
2 – за включване в края на списъка
3 – за изключване от началото на списъка
4 – за изключване от края на списъка
5 – за завършване обработката на списъка
? 1
Въведете число с плаваща точка: 1.1
Списък: 1.1
? 1
```

```
Въведете число с плаваща точка: 2.2
Списък: 2.2 1.1
? 2
Въведете число с плаваща точка: 3.3
Списък: 2.2 1.1 3.3
? 2
Въведете число с плаваща точка: 4.4
Списък: 2.2 1.1 3.3 4.4
? 3
2.2 е изключен от списъка
Списък: 1.1 3.3 4.4
? 3
1.1 е изключен от списъка
Списък: 3.3 4.4
? 4
4.4 е изключен от списъка
Списък: 3.3
? 4
```

```
3.3 е изключен от списъка
Списъкът е празен!
? 5
Край на тестването на списъка
```

18. Свързан стек. Основни операции

Със средствата на C++ ще опишем основните операции – включване и изключване на елементи за свързан стек, т.е. стек представен чрез свързан списък.

```
#include <iostream.h>
#include <stdlib.h>
typedef int INFOTYPE;
struct Stack
 { INFOTYPE info;
   Stack *next;
 } ;
```

```
void push (Stack *&s, INFOTYPE x)
 { Stack *p;
   if ( (p = new Stack) == NULL)
    { cout << “Няма свободна памет”;
      exit (1);
    }
   p -> info = x;
   p -> next = s;
   s = p;
 }
INFOTYPE pop (Stack *&s)
 { INFOTYPE x;
   Stack *p;
  if (!s) { cout << “Стекът е празен\n”;
            exit (1);
           }
  x = s -> info;
  p = s;
```

```
  s = s -> next;
  delete p;
  return x;
 }
void main ()
 { INFOTYPE y; Stack *stack = NULL;
   y = 10;
   push (stack, y);
   y = pop (stack);
   y = pop (stack);
 }
```

Резултат:

Стекът е празен

19. Пример за обектно-ориентирана реализация на свързан стек

Ще се възползваме от тясната връзка между свързаните списъци и стекове и при реализацията на клас за стек повторно ще използваме класа List. Ще разгледаме две разновидности на повторното използване:

- реализиране на клас за стек чрез скрито наследяване на класа List;
- реализиране на клас за стек чрез влагане на класа List.

Двата класа ще реализираме като шаблони, за да могат да се използват повторно.
Програмата създава шаблон на класа Stack посредством скрито наследяване на шаблона на класа List. Стекът ще има функции-елементи push, pop, isStackEmpty и printStack. Те по същество са съответно функциите от шаблона на класа List insertAtBeg, removeFromBeg, isEmpty и print. Шаблонът на класа List включва и други функции-елементи (например insertAtEnd и removeFromEnd),

които не искаме да правим достъпни с открития интерфейс на класа Stack. Така че, когато казваме, че шаблонът на класа Stack трябва да наследи шаблона на класа List, тогава задаваме скрито наследяване. То води до това, че всички функции-елементи на шаблона на класа List в шаблона на класа Stack стават скрити. Когато реализираме функциите-елементи на Stack, те трябва да извикват съответстващите функции-елементи на класа List: push трябва да извиква insertAtBeg, pop - removeFromBeg, isStackEmpty - isEmpty, а функцията-елемент printStack трябва да извиква print.

файл stack.h

```
#ifndef STACK_H
#define STACK_H
#include “list.h”
template <class StackType>
class Stack : private List <StackType> {
 public :
     void push (const StackType &value)
```

```
      {  insertAtBeg (value); }
    bool pop (StackType &value)
      { return removeFromBeg (value); }
    bool isStackEmpty () const
      { return isEmpty (); }
    void printStack () const
      { print (); }
} ;
#endif
```

файл teststack.cpp

```
#include “stack.h”
void main ()
 { Stack <int> intStack;
   int popint, i;
   cout << “Обработка на стек с цели числа” << endl;
   for (i = 0; i < 4; i++)
    { intStack.push (i);
```

```
   intStack.printStack ();
 }
while (!intStack.isStackEmpty())
 { intStack.pop (popint);
   cout << popint << " е изключено от стека!" << endl;
   intStack.printStack ();
 }
Stack <double> doubleStack;
double val = 1.1, popval;
cout << "Обработка на стек с числа с плаваща точка" << endl;
for (i = 0; i < 4; i++)
 { doubleStack.push (val);
   doubleStack.printStack ();
   val += 1.1;
 }
while (!doubleStack.isStackEmpty())
 { doubleStack.pop (popval);
   cout << popval << " е изключено от стека!" << endl;
```

```
    doubleStack.printStack ();
  }
}
```

Резултат:

```
Обработка на стек с цели числа
Списък: 0
Списък: 1 0
Списък: 2 1 0
Списък: 3 2 1 0
3 е изключено от стека
Списък: 2 1 0
2 е изключено от стека
Списък: 1 0
1 е изключено от стека
Списък: 0
0 е изключено от стека
Списъкът е празен
Обработка на стек с числа с плаваща точка
```

```
Списък: 1.1
Списък: 2.2 1.1
Списък: 3.3 2.2 1.1
Списък: 4.4 3.3 2.2 1.1
4.4 е изключено от стека
Списък: 3.3 2.2 1.1
3.3 е изключено от стека
Списък: 2.2 1.1
2.2 е изключено от стека
Списък: 1.1
1.1 е изключено от стека
Списъкът е празен
Всички обекти са изключени!
Всички обекти са изключени!
```

Сега ще опишем шаблон на клас, който реализира стек чрез влагане на класа List. Определянето на шаблона на класа Stack включва обект-елемент s от тип List <StackType>.

файл stack-c.h

```
#ifndef STACK_C_H
#define STACK_C_H
#include “list.h”
template <class StackType>
class Stack {
 public:
     // няма конструктор, инициализира се конструктора на List
    void push (const StackType &value)
      { s.insertAtBeg(value); }
    bool pop (StackType &value)
      { return s.removeFromBeg(value); }
    bool isStackEmpty () const
      { return s.isEmpty (); }
    void printStack () const
      { s.print (); }
 private:
```

```
    List <StackType> s;
} ;
#endif
```

Откритият интерфейс на класа Stack не позволява на потребителя да използва функциите-елементи от класа List, тъй като обектът s е скрит. За тестване на stack-c.h можем да използва същия файл teststack.h с тази разлика, че в началото включваме stack-c.h вместо stack.h.

20. Свързана опашка. Основни операции

Ще реализираме опашка с фиктивен елемент в началото. Първият реален елемент на опашката е непосредствено след фиктивния. Опашката е празна, когато указателят към фиктивния елемент съвпада с указателя към края на опашката.

```
#include <iostream.h>
typedef float InfoType;
struct Queue_El {
  InfoType info;
  Queue_El *next;
} ;
void enqueue (Queue_El *&r, InfoType x)
 { Queue_El *p = new Queue_El;
   if (p==NULL)
     { cout << “Няма свободна памет!” << endl;
       exit (1);
     }
   p -> info = x; p -> next = NULL;
   r -> next = p; r = p;
 }
InfoType dequeue (Queue_El *f, Queue_El *&r)
 { InfoType x; Queue_El *p = f -> next;
   if (p==NULL) //еквивалентно е на f == r
```

```
    { cout << “Опашката е празна!” << endl;
      exit (1);
    }
   f -> next = p -> next;
   x = p -> info;
   delete p;
   if (f -> next == NULL) r = f;
   return x;
 }
void main ()
 { Queue_El Q = { 0, NULL}, *F = &Q, *R = F;
   InfoType y = 3.14f;
   enqueue (R, y);
   y = dequeue (F, R);
 }
```

Резултат:

!!! Няма изход !!!

21. Пример за обектно-ориентирана реализация на свързана опашка

Ще разгледаме програма, която създава шаблон на класа Queue като използва скрито наследяване на класа List. Функциите-елементи, необходими за обработката на опашката по същество са реализирани в шаблона на класа List. Този шаблон включва други функции-елементи и не е желателно те да са достъпни за шаблона на класа Queue чрез открит интерфейс. Поради това използваме скрито наследяване и в резултат всички функции-елементи на шаблона на класа List стават скрити функции-елементи в шаблона на класа Queue.
Когато реализираме функциите-елементи на класа Queue, те трябва да извикват съответстващите функции-елементи на класа List: enqueue трябва да извиква insertAtEnd, dequeue - removeFromBeg, isQueueEmpty - isEmpty, а функцията-елемент printQueue трябва да извиква print.

файл queue.h

```
#ifndef QUEUE_H
#define QUEUE_H
#include “list.h”
template <class QueueType>
class Queue : private List <QueueType> {
 public:
    void enqueue (const QueueType &d)
      { insertAtEnd (d); }
    bool dequeue (QueueType &d)
      { return removeFromBeg (d); }
    bool isQueueEmpty () const
      { return isEmpty (); }
    void printQueue () const
      { print (); }
} ;
#endif
```

файл testqueue.cpp

```
#include “queue.h”
void main ()
 { Queue <int> intQueue;
   int dequeueInt, i;
   cout << “Обработка на опашка с цели числа” << endl;
   for (i = 0; i < 4; i++)
    { intQueue.enqueue (i);
      intQueue.printQueue();
    }
   while (!intQueue.isQueueEmpty())
    { intQueue.dequeue (dequeueInt);
      cout << dequeueInt << “ е изключено!” << endl;
      intQueue.printQueue ();
    }
   Queue <double> doubleQueue;
   double dequeueDouble, val = 1.1;
   cout << “Обработка на опашка с числа с плаваща точка” << endl;
```

```
   for (i = 0; i < 4; i++)
    { doubleQueue.enqueue (val);
      doubleQueue.printQueue();
      val += 1.1;
    }
   while (!doubleQueue.isQueueEmpty())
    { doubleQueue.dequeue (dequeueDouble);
      cout << dequeueDouble << " е изключено!" << endl;
      doubleQueue.printQueue ();
    }
 }
```

Резултат:

```
Обработка на опашка с цели числа
Списък: 0
Списък: 0 1
Списък: 0 1 2
Списък: 0 1 2 3
0 е изключено!
```

```
Списък: 1 2 3
1 е изключено!
Списък: 2 3
2 е изключено!
Списък: 3
3 е изключено!
Списъкът е празен!
Обработка на опашка с числа с плаваща точка
Списък: 1.1
Списък: 1.1 2.2
Списък: 1.1 2.2 3.3
Списък: 1.1 2.2 3.3 4.4
1.1 е изключено!
Списък: 2.2 3.3 4.4
2.2 е изключено!
Списък: 3.3 4.4
3.3 е изключено!
Списък: 4.4
```

4.4 е изключено!
Списъкът е празен!
Всички обекти са изключени!
Всички обекти са изключени!

22. Пример за определяне на принадлежност на низ към формален език.

Формален език се дефинира с формална граматика Г = <N,T,S,P>, където N е множество от нетерминални символи, T е множество от терминални символи, S $\in$ N е начален нетерминал, P е множество от продукции, които имат синтактичен характер и те определят начина по който се строят думи в езика.
Тук разглеждаме формалният език, породен от следната граматика Г = < { S }, { a, b, c}, S, { S -> aSa, S -> bSb, S -> c } >.

Лесно се вижда, че езикът на тази граматика се състои от всички думи $\alpha c \alpha'$, където α' е получена от α чрез инвертиране и α се състои само от a, b.
Ще разгледаме програма, която разпознава дали даден низ принадлежи на този език. За целта използваме следния алгоритъм:

- въвеждаме низа в стек до срещане на c;
- след това сравняваме останалите символи със символите в стека и ако има пълно съвпадение, то низът е от езика.

За реализацията ще опишем шаблон на стек;

файл stack2.h

```
#ifndef STACK2_H
#define STACK2_H
template <class T>
struct item {
 T value;
```

```
 item *next;
} ;
template <class T>
class Stack {
 public:
   Stack ();
   ~Stack ();
   void push (const T&);
   bool pop (T&);
   T top () const;
   bool isEmpty () const;
 private:
   item <T> *first;
} ;
template <class T>
Stack <T>::Stack () : first (NULL) { }
template <class T>
```

```
Stack <T>::~Stack ()
 { item <T> *ptr;
   while (first)
     { ptr = first;
       first = first -> next;
       delete ptr;
     }
  }
template <class T>
void Stack <T>::push (const T&x)
 { item <T> *ptr = new item <T>;
   //считаме, че е включена опция за стандартно прекъсване при
   //изпълнението на new когато няма свободна памет и за това
   //не правим проверка дали операцията е успешна
   ptr -> value = x;
   ptr -> next = first;
   first = ptr;
 }
```

```
template <class T>
bool Stack <T>::pop (T &x)
 { if (isEmpty()) return false;
   item <T> *ptr = first;
   first = first -> next;
   x = ptr -> value;
   delete ptr;
   return true;
 }
template <class T>
T Stack <T>::top () const
 { return first -> value; }
//предполагаме, че функцията top няма да се извиква за празен
стек
template <class T>
bool Stack <T>::isEmpty () const
 { return first==NULL; }
#endif
```

Сега ще опишем самата програма;

```
#include <iostream.h>
#include "stack2.h"
void Error ()
 { cout << "Низът не е от езика!" << endl; }
void main ()
 { Stack <char> chStack;
   char ch, R;
   cout << "Въведете низ: ";
   ch = cin.get ();
   if (ch < 'a' || ch > 'c') { Error(); return; }
   while (ch != 'c')
    { chStack.push (ch);
      ch = cin.get ();
      if ( ch < 'a' || ch > 'c') { Error(); return; }
    }
   ch = cin.get ();
   if (ch != 'a' && ch != 'b' && ch != '\n') { Error(); return; }
```

```
  while (ch != '\n')
   { if (!chStack.isEmpty())
      if (ch != chStack.top())
       { Error(); return; }
      else;
     else { Error(); return; }
    chStack.pop (R);
    ch = cin.get ();
    if (ch != 'a' && ch != 'b' && ch != '\n') { Error(); return; }
   }
  if (!chStack.isEmpty ()) { Error(); return; }
  cout << "Низът е от езика!" << endl;
}
```

Резултат:

Въведете низ: c

Низът е от езика!

Въведете низ: aca

Низът е от езика!

Въведете низ: abacaba
Низът е от езика!
Въведете низ: aabcaab
Низът не е от езика!

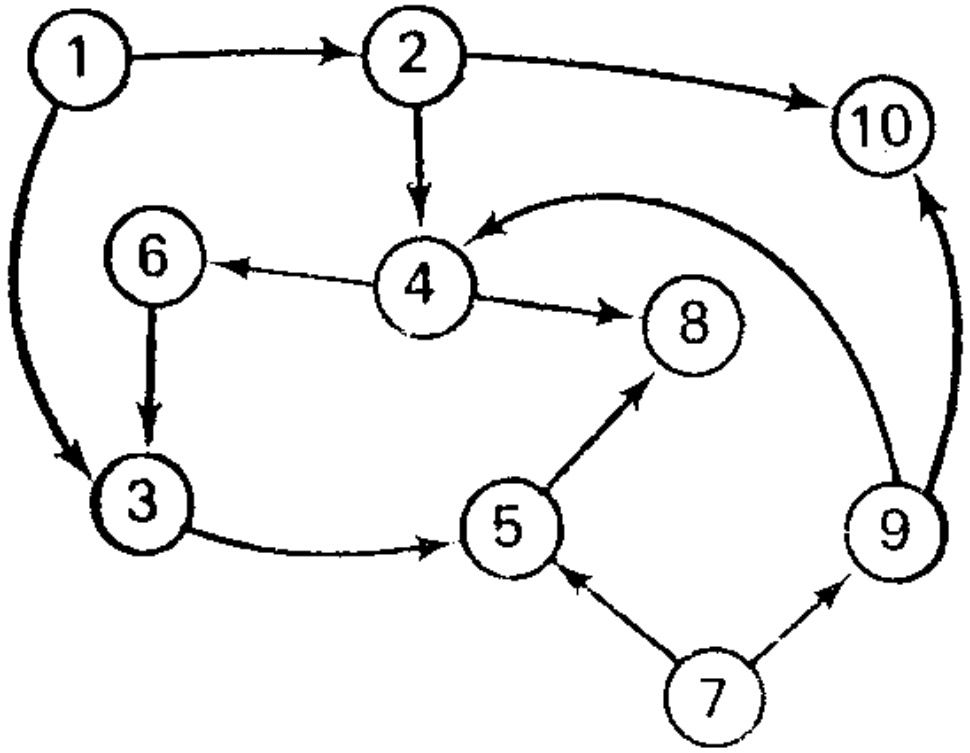
23. Топологично сортиране.

Нека S е крайно множество. Без ограничение S = { 1, 2, …, n}.

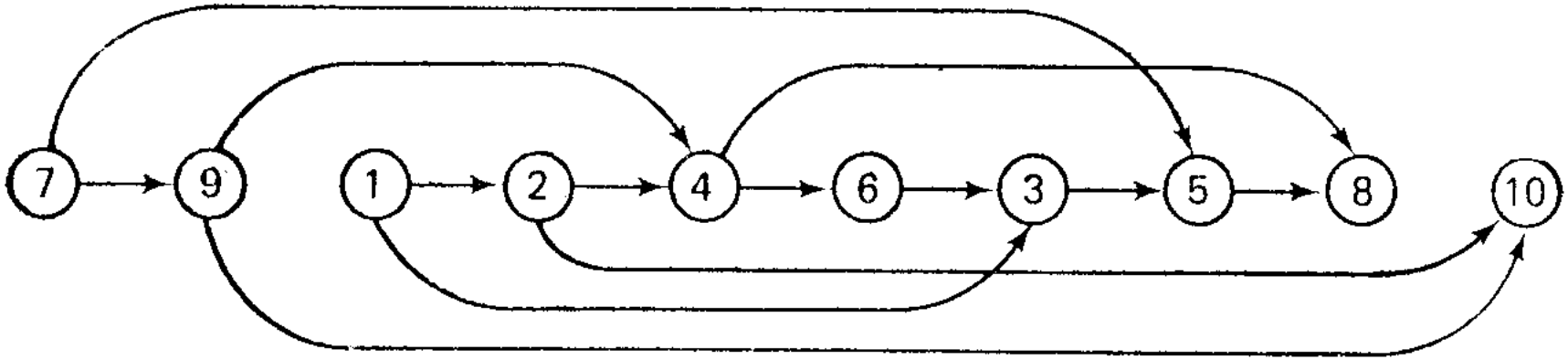
Нека в S е въведена релация $\prec$ (частична наредба). Ако a $\in$ S, b $\in$ S и a $\prec$ b, казваме че a предшества b. Частичната наредба е дефинирана за някои двойки елементи, а не за всички.

Предполагаме, че релацията $\prec$ няма контури, т.е. няма последователност от вида a $\prec$ b $\prec$ … $\prec$ a.
Пример за частична наредба е университетска учебна програма, в която някои курсове трябва да се проведат преди други, тъй като последните са свързани с материал, включен в предшестващите

ги курсове. Ако курсът v трябва да предшества курса w, пишем $v \prec w$. Топологичното сортиране означава подреждане на курсовете в такъв ред, че никой курс да не се позовава на материал, който ще се предава в следващи курсове.
Частичната наредба може да се илюстрира чрез граф, върховете на който означават елементите на S, а ребрата представляват отношенията на подреждане.



Целта на топологичното сортиране е да се преобразува частичната наредба в линейна. Графично това води до нареждане на върховете на графа в редица, така че стрелките да сочат надясно, както е показано:



При това положение, релацията $\prec$ може да се допълни до линейна наредба, т.е. да са изпълнени следните свойства:

- за всеки x, y, z $\in$ S: от x $\prec$ y и y $\prec$ z ➔ x $\prec$ z (транзитивност);
- за всеки x, y $\in$ S е изпълнено точно едно от x $\prec$ y и y $\prec$ x (силна антисиметричност, несиметричност);
- за всяко x $\in$ S имаме x $\not\prec$ x (антирефлексивност, нерефлексивност).

Алгоритъмът, който допълва релацията $\prec$ до линейна наредба, се нарича **топологично сортиране** и той е следния:

1. намираме елемент $s \in S$, който не се предшества от друг елемент и го извеждаме в края на опашката за извеждане;
2. изключваме избрания елемент s от множеството S, заедно с всички връзки на s с други елементи;
3. ако $S \neq \emptyset$ - премини към 1, иначе край.

В резултат получаваме пермутация $i_1, i_2, \ldots, i_n$ на 1, 2, …, n и линейната наредба се определя по следния начин: $a \prec b$, ако $a = i_s$, $b = i_t$ и $s < t$.

Ако на дадена стъпка не съществува елемент, който не се предшества от друг, то първоначалната релация $\prec$ е съдържала контури, което е недопустимо.

За реализацията ще използваме масив от |S| структури, по една за всеки елемент от S, като всяка структура съдържа поле count – броя на непосредствените предшественици на елемента и поле

stack – стек, съдържащ непосредствените наследници на елемента.
Елементите с count = 0 се обединяват в опашка, в края на която след всяка стъпка се добавят новополучените елементи с count = 0. Полетата за връзка в тази опашка се разполагат в полетата count, които са вече непотребни.
В програмата ще въведем ограничение за броя на елементите на S.

От входа постъпват броя на елементите и двойките a $\prec$ b на релацията $\prec$. Входните данни са следните:
10

1$\prec$2, 2$\prec$4, 4$\prec$6, 2$\prec$10, 4$\prec$8, 6$\prec$3, 1$\prec$3, 3$\prec$5, 5$\prec$8, 7$\prec$5, 7$\prec$9, 9$\prec$4, 9$\prec$10
Въвеждането на двойки се преустановява с въвеждане на фиктивна двойка 0 0.

```
#include <iostream.h >
```

```
#include “stack2.h”
#define max 100
void main ()
 { int n, j, k, f, r = 0;
   struct Elem {
     int count;
     Stack <int> stack;
   } x[max+1]; //номерацията започва от 1
  do {
   cout << “Въведете броя на елементите: “;
   cin >> n;
  } while (n < 0 && n > max);
  for (k = 0; k <=n; k++) x[k].count = 0;
  do {
    do {
      cout << “Въведете отношение или 0 0 за край: “;
      cin >> j >> k; if ( j==0 && k==0) break;
    } while ( j < 1 || k < 1 || j > max || k > max || j == k);
```

```
 if (j==0 && k==0) break;
 x[k].count++;
 x[j].stack.push (k);
} while (1);
for (k = 1; k <= n; k++)
 if (x[k].count==0)
  { x[r].count = k;
    r = k;
  }
f = x[0].count;
while (1) {
  if (f==0 && n!=0)
   { cout << “Релацията съдържа контури!” << endl;
     return;
   }
  if (f==0) break;
  n--;
  while ( x[f].stack.pop (k))
```

```
    { x[k].count--;
      if (x[k].count==0)
       { x[r].count = k;
         r = k;
       }
    }
   f = x[f].count;
 }
 cout << “Линейна наредба:“;
 f = x[0].count;
 while (f != 0)
   { cout << “  “ << f; f = x[f].count; }
 cout << endl;
}
```

Резултат:

Въведете броя на елементите: 10

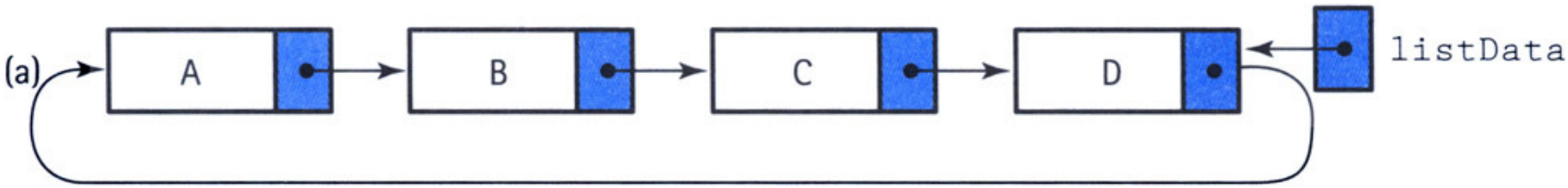
Въведете отношение или 0 0 за край: 1 2

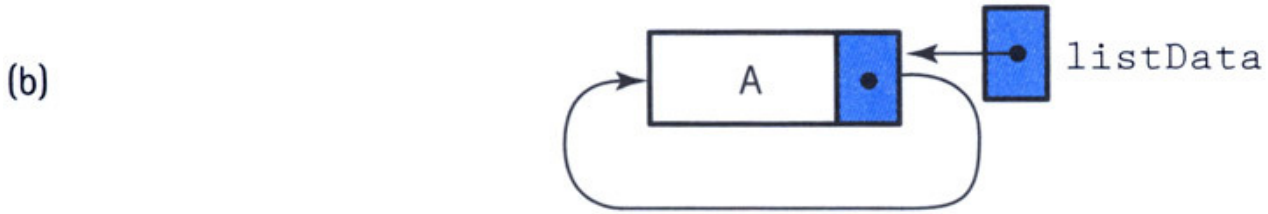
Въведете отношение или 0 0 за край: 2 4

Въведете отношение или 0 0 за край: 4 6
Въведете отношение или 0 0 за край: 2 10
Въведете отношение или 0 0 за край: 4 8
Въведете отношение или 0 0 за край: 6 3
Въведете отношение или 0 0 за край: 1 3
Въведете отношение или 0 0 за край: 3 5
Въведете отношение или 0 0 за край: 5 8
Въведете отношение или 0 0 за край: 7 5
Въведете отношение или 0 0 за край: 7 9
Въведете отношение или 0 0 за край: 9 4
Въведете отношение или 0 0 за край: 9 10
Въведете отношение или 0 0 за край: 0 0
Линейна наредба: 1 7 2 9 10 4 6 3 5 8

24. Циклични списъци.

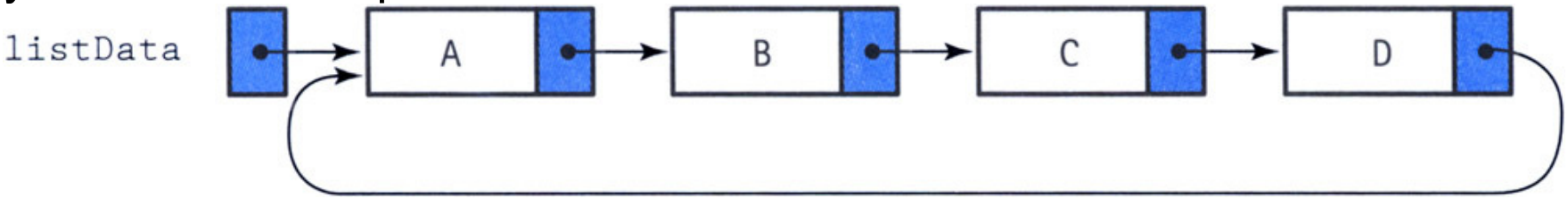
При цикличните списъци, за разлика от обикновените свързани списъци, в полето за връзка на последния елемент е записан адреса на първия елемент, т.е. последният елемент сочи към първия елемент на списъка, вместо да съдържа NULL. Така обикновеният списък става цикличен. Удобно е цикличните списъци да се задават с указател към последния елемент.


(a)
A
B
C
D
listData


(b)
A
listData


(c)
listData
(empty list)

Има вариант, при който цикличните списъци да се задават с указател към първия елемент.



Можем да започнем от всеки елемент и да обходим целия списък.
Ще разгледаме шаблон на класа CircleList, който реализира цикличен свързан списък с данни от тип T.
Функциите insert и remove реализират добавяне и изключване на елемент в началото на списъка.

файл circlelist.h

```
#ifndef CIRCLELIST_H
#define CIRCLELIST_H
template <class T>
struct Elem {
 T info;
 Elem <T> *link;
```

```
} ;
template <class T>
class CircleList {
 public:
   CircleList ();
   ~CircleList ();
   void insert (const T&);
   bool remove (T&);
 private:
   Elem <T> *ptr;
} ;
template <class T>
CircleList<T>::CircleList ()
 { ptr = NULL; }
template <class T>
CircleList<T>::~CircleList ()
 { if (ptr == NULL) return;
   Elem <T> *first = ptr -> link;
```

```
   ptr -> link = NULL;
   while (first != NULL)
    { ptr = first;
      first = first -> link;
      delete ptr;
    }
 }
template <class T>
void CircleList<T>::insert (const T& value)
 { Elem <T> *newel = new Elem <T>;
   newel -> info = value;
   if (ptr != NULL) newel -> link = ptr -> link;
   else ptr = newel;
   ptr -> link = newel;
 }
template <class T>
bool CircleList<T>::remove (T& value)
 { if (ptr == NULL) return false;
```

```
   value = ptr -> link -> info;
   Elem <T> *first = ptr -> link;
   if (ptr == first) ptr = NULL;
  else ptr -> link = first -> link;
  delete first;
  return true;
 }
#endif
```

Ще използваме циклични списъци за да реализираме алгоритъм за събиране на полиноми. Полиномите ще са на три променливи x, y, z и ще се представят с циклични списъци от едночлени с фиктивен елемент в края на списъка. Типът на елементите ще е структура със следните полета:

- поле C – коефициента на едночлена;
- поле S – ‘+’, ако елементът не е фиктивен и ‘-‘, ако е фиктивен елемент;
- поле X – степента на x;

- поле Y – степента на y;
- поле Z – степента на z;
- поле link – връзка към следващия елемент.

В алгоритъма ще предполагаме, че едночлените в един полином са подредени според стандартната лексикографска наредба – първо по нарастващите степени на x, после по нарастващите степени на y, после по нарастващите степени на z. Също ще предполагаме, че в полиномите е извършено приведение на подобните едночлени.

Алгоритъмът е следния:

(резултатът се получава в полинома, сочен от Q)

- създаваме списъците за двата полинома. Инициализираме указателите P и Q да сочат към началото съответно на първия и втория списък. Инициализираме указателите P_1 и Q_1 да сочат към фиктивния елемент съответно на първия и втория списък;
- докато P -> S != ‘-‘ или Q -> S != ‘-‘:
  - ако едночленът, сочен от P е по-нисък от едночлена, сочен от Q или P сочи към фиктивен елемент –

Q_1 = Q, Q = Q -> link;

- ако едночленът, сочен от P е по-висок от едночлена, сочен от Q или Q сочи към фиктивен елемент – добавяне на едночлена, сочен от P след едночлена, сочен от Q_1, Q_1 = Q_1 -> link, P_1 = P, P = P -> link;
- ако едночленът, сочен от P е подобен на едночлена, сочен от Q – актуализиране на коефициента на Q, P_1 = P, P = P -> link, ако коефициентът на едночлена, сочен от Q е станал 0 (и има поне два едночлена), изтриваме го, Q = Q_1 -> link, иначе Q_1 = Q, Q = Q -> link;

файл sumpoly.cpp

```
#include <iostream.h>
#include ”circlelist.h”
struct Term {
 double C;
 char S;
 int X, Y, Z;
```

```
};
struct Poly {
 Term info;
 Poly *link;
};
void add (Poly *P, Term &info)
 { Poly *newel = new Poly;
   newel -> info = info;
   newel -> link = P -> link;
   P -> link = newel;
 }
bool larger (Term &A, Term &B)
 { return ( A.X > B.X ||
          ( A.X == B.X && A.Y > B.Y ) ||
          ( A.X == B.X && A.Y == B.Y && A.Z > B.Z) );
 }
void remove (Poly *P)
 { Poly *temp = P -> link;
```

```
   P -> link = P -> link -> link;
   delete temp;
 }
void getPoly (Poly *&P, const char *name)
 { Term current;
   cout << name << " полином: " << endl;
   cout << "Въвеждането е в нарастващ лексикографски ред!" <<
endl;
   cout << "Въведете коефициент, степен на X, степен на Y," << endl
        << " степен на Z или -1 -1 -1 -1 за край:" << endl;
   do {
    cin >> current.C >> current.X >> current.Y >> current.Z;
    if (current.C == -1 && current.X == -1 &&
        current.Y == -1 && current.Z == -1) break;
    if (current.C == 0 || current.X < 0 ||
        current.Y < 0 || current.Z < 0) continue;
    current.S = '+';
    add (P, current);
```

```
  } while (1);
 }
void printPoly (Poly *P)
 { cout << "Сумата в същия формат е: " << endl;
   while (P -> info.S != '-')
    { if (P -> info.C == 0) cout << 0 << endl;
      cout << P -> info.C << ' ' << P -> info.X
           << ' ' << P -> info.Y << ' ' << P -> info.Z << endl;
      P = P -> link;
    }
 }
void main ()
 { const Term fict = { 0, '-', 0, 0, 0 };
   Poly *P = new Poly, *Q = new Poly, *Q1 = Q, *P1 = P;
   P -> info = fict; P -> link = P;
   Q -> info = fict; Q -> link = Q;
   getPoly (P, "Първи");
   getPoly (Q, "Втори");
```

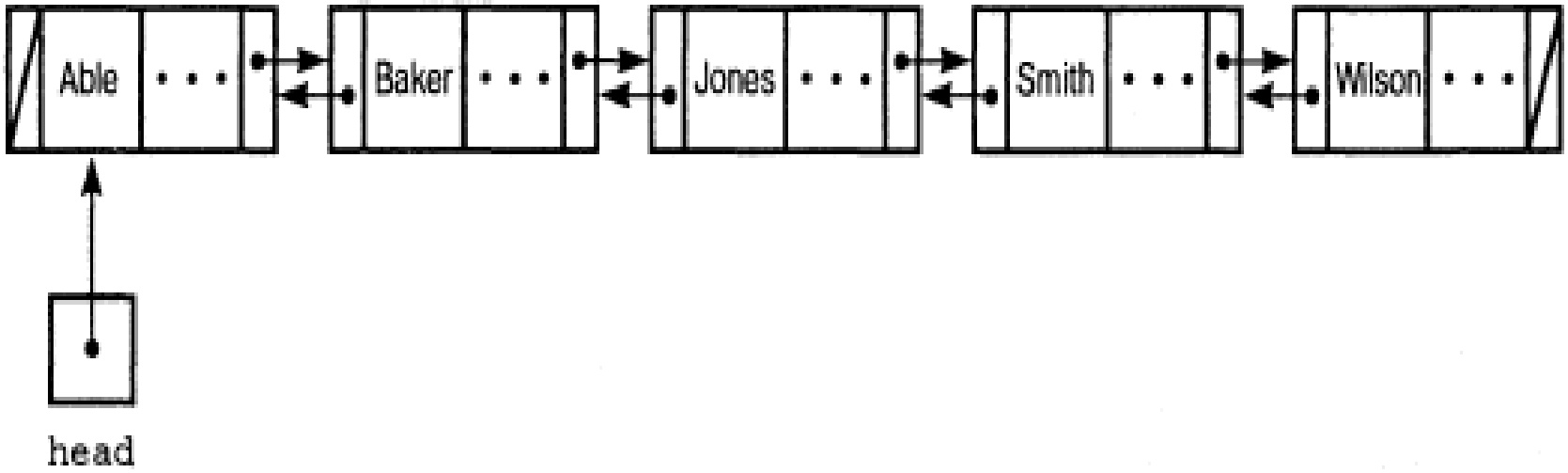
```
  P = P1 -> link; Q = Q1 -> link;
  while (P -> info.S != '-' || Q -> info.S != '-')
   { if (larger (Q -> info, P -> info) || P -> info.S == '-')
      { Q1 = Q; Q = Q -> link; }
     else if (larger (P -> info, Q -> info) || Q -> info.S == '-')
      { add (Q1, P -> info); Q1 = Q1 -> link;
        P1 = P; P = P -> link; }
     else {
        Q -> info.C += P -> info.C;
        P1 = P; P = P -> link;
        if (Q -> info.C == 0 && Q -> link != Q1)
            { remove (Q1); Q = Q1 -> link; }
        else { Q1 = Q; Q = Q -> link; }
     }
   }
  printPoly (Q->link);
}
```

Събираме двата полинома:
$2xy^2z + 4xy^2z^4 + 4xy^3z^4 + 2x^2y^2z^3 + x^3yz$
$8xy^3z^4 + 2x^2y^2z^3 + 4x^3yz$
Сумата на двата полинома е:
$2xy^2z + 4xy^2z^4 + 12xy^3z^4 + 4x^2y^2z^3 + 5x^3yz$
Резултат:
Първи полином:
Въвеждането е в нарастващ лексикографски ред!
Въведете коефициент, степен на X, степен на Y,
степен на Z или -1 -1 -1 -1 за край:
2 1 2 1
4 1 2 4
4 1 3 4
2 2 2 3
1 3 1 1
-1 -1 -1 -1
Втори полином:
Въвеждането е в нарастващ лексикографски ред!

Въведете коефициент, степен на X, степен на Y,
степен на Z или -1 -1 -1 -1 за край:
8 1 3 4
2 2 2 3
4 3 1 1
-1 -1 -1 -1
Сумата в същия формат е:
5 3 1 1
4 2 2 3
12 1 3 4
4 1 2 4
2 1 2 1

25. Двусвързани списъци.

При двусвързаните списъци елементите имат полета за връзка както към следващия елемент, така и към предишния елемент.



Възможно е двусвързаният списък да е цикличен – тогава полето за връзка към предишния елемент за първия елемент е към последния елемент и полето за връзка към следващия елемент на последния елемент е към първия елемент.
Елементите на списъка можем да представим като тип структура:

```
template <class InfoType>
struct Elem {
```

```
 InfoType info;
 Elem <InfoType> *next, *prev;
} ;
```

Ще опишем функция insert за включване на елемент след указан елемент и функция remove за изключване на указан елемент от двусвързан списък. Предполагаме, че списъкът има фиктивен елемент, така че при извикване на функцията insert ptr има стойност различна от NULL, а при функцията delete ptr не сочи към фиктивния елемент.

```
template <class InfoType>
void insert ( Elem <InfoType> *ptr, InfoType x )
 { Elem <InfoType> *p;
   if ( (p = new Elem <InfoType>) == NULL )
    { cout << “Няма достатъчно памет!” << endl;
      exit (1);
    }
```

```
  p -> info = x;
  p -> prev = ptr;
  p -> next = ptr -> next;
  ptr -> next -> prev = p;
  ptr -> next = p;
 }
template <class InfoType>
InfoType remove (Elem <InfoType> *ptr)
 { InfoType x = ptr -> info;
  ptr -> prev -> next = ptr -> next;
  ptr -> next -> prev = ptr -> prev;
  delete ptr;
  return x;
 }
```

26. Графи

Списъците са удобно средство при решаване на много проблеми при създаване на алгоритми, но понякога се налага използването на по-сложни от линейните структури от данни.

Графите са нелинейни структури от данни; делят се на два вида:

- **неориентиран граф** – разглежда се като G = < A, P >, където A е множество от **върхове** и P е множество от ненаредени двойки върхове, наречени **ребра**;
- **ориентиран граф** – разглежда се като D = < A, R >, където A е множество от върхове и R е множество от наредени двойки върхове, наречени **дъги**;

Ще разгледаме по-подробно крайните ориентирани графи, т.е. графите D = <A,R> при които A е крайно множество.

Всъщност теориите на ориентираните и неориентираните графи са независими една от друга и разглеждат различни обекти.

Нека е даден ориентиран граф D = < A, R >.

Нека $A_1 \subseteq A$, $R_1 \subseteq R$; графът $D_1 = < A_1, R_1 >$ се нарича **част** от D.

Ако $A_1 = A$, казваме че D_1 е **субграф** на D.

Нека $A_2 \subseteq A$, $R_2 \subseteq R$, така че R_2 съдържа всички дъги с краища върхове от A_2 и само тях. Графът $D_2 = < A_2, R_2 >$ се нарича **подграф** на D. Ако $D_2 \neq D$, казваме че D_2 е **истински подграф** на D.

Понякога ориентираният граф е такъв, че при него всички дъги имат присвоена величина, която наричаме **тегло.**

Пример.

Нека графът D се състои от всички кръстовища и свързващите ги улици в даден град.

Ако D_1 съдържа всички кръстовища и всички улици, които ги свързват в даден квартал на града, то D_1 е подграф на D.

Ако D_2 съдържа всички кръстовища в града и свързващите ги асфалтирани улици, то D_2 е субграф на D.

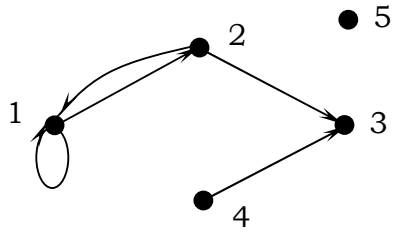
Ако D_3 съдържа всички кръстовища и улици по които се движи градския транспорт, то D_2 е част от D.

Ако $(a, b) \in R$ и $a = b$, дъгата (a, b) наричаме **примка.**

Нека $X \subseteq A$ и $(a, b) \in R$.

Ако $a \notin X$, $b \in X$, казваме че дъгата (a, b) **влиза в** X.

Ако $a \in X$, $b \notin X$, казваме че дъгата (a, b) **излиза от** X.
Дъгите, които влизат в X и излизат от X се наричат **инцидентни** с X.
Ще означаваме с $RX^+ \subseteq R$ множеството от всички дъги, които излизат от X. Броят на елементите на RX^+ се нарича **полустепен на изхода** на X, означаваме od (X).
Ще означаваме с $RX^- \subseteq R$ множеството от всички дъги, които излизат от X. Броят на елементите на RX^- се нарича **полустепен на входа** на X, означаваме id (X).
С td (X) означаваме броят на дъгите, инцидентни с X, т.е. td (X) = id (X) + od (X).
Пример.



Това е изображение на следния граф:
D = < { 1, 2, 3, 4, 5}, { (1, 2), (2, 1), (1, 1), (2, 3), (4, 3) } > .
Нека X = { 2, 4}.
Имаме RX^+ = { (2, 3), (4, 3), (2, 1) } и od (X) = 3,
RX^- = { (1, 2)} и id (X) = 1, td (X) = id (X) + od (X) = 3 + 1 = 4 .
Лесно се вижда, че сумата на полустепените на входа и на изхода на всички върхове на графа D дава броят на всички дъги в D, които не са примки.
Последователността $(a_1, a_2), (a_2, a_3), \ldots, (a_{n-1}, a_n)$ от дъги наричаме **път** от a_1 до a_n. Ако $a_1 = a_n$, казваме че пътят е **цикъл.**
Ако $a_i \neq a_j$ при $i \neq j$, казваме че пътят е **прост.**
Ако върховете $a_1, \ldots, a_{n-1}$ са два по два различни и $a_1 = a_n$, казваме че пътят е **прост цикъл.**
Прост път, който съдържа всички върхове на графа се нарича **Хамилтонов път.**
Прост цикъл, който съдържа всички върхове на графа се нарича **Хамилтонов цикъл.**

Например в горния граф (1, 2), (2, 3) е прост път, (1, 2), (2, 1) е прост цикъл. В него не съществуват Хамилтонови пътища и цикли. Граф, който съдържа поне един цикъл се нарича **цикличен**, ако не съдържа нито един цикъл, казваме че графът е **ацикличен.**
Върхът b е **достижим** от върха a, ако съществува път от a до b. Върхът a наричаме **изолиран** връх, ако той не е достижим от никой връх и никой връх не е достижим от него. В горния граф върхът 5 е изолиран връх, 3 е достижим от 1, 4 не е достижим от 3.
Нека $J \subseteq A$. J наричаме **минимално пораждащо подмножество** на A, ако:

- всеки връх на A е достижим от някой връх от J;
- J е минимално по включване с това свойство.

Например минималното пораждащо подмножество на ацикличен ориентиран граф се определя като множеството от всички върхове с полустепен на входа 0.
Ориентиран граф $D = < A, R >$ с върхове $a_1, a_2, \ldots, a_n$ може еднозначно да бъде определен с квадратна матрица от ред n. Тази матрица се нарича **матрица на съседство.** Ще я означаваме с X.

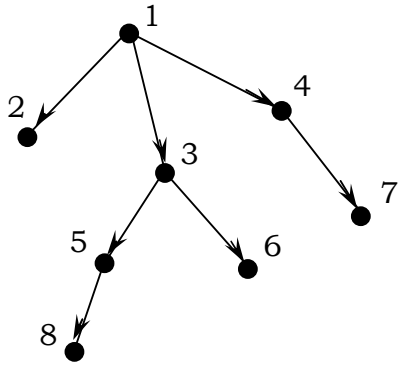
Тя се дефинира по следния начин: $X = (x_{ij})$, $x_{ij} = 1$, ако $(a_i, a_j) \in R$ и $x_{ij} = 0$, ако $(a_i, a_j) \notin R$.
За задаване на теглата се използва **матрица на теглата** $W = (w_{ij})$, w_{ij} = теглото на дъгата (a_i, a_j), ако $(a_i, a_j) \in R$ и $w_{ij} = 0$, ако $(a_i, a_j) \notin R$.
За графа D дефинираме **матрица на пътищата** P по следния начин: $P = (p_{ij})$, $p_{ij} = 1$, ако има път от a_i до a_j и $p_{ij} = 0$, ако няма път от a_i до a_j.
Ще отбележим, че матрицата P не определя еднозначно графа.
Теорема: Нека $Y = X^h$, $h \in N$; тогава y_{ij} дава броя на различните пътища с дължина h от a_i до a_j.
Следствие: Ако $X^h = O$ за някое h, то графът D е ацикличен.
Ориентираният граф $D = < A, R >$ може да се представи и чрез списъци по следния начин: списък от върховете и за всеки връх списък от дъгите с начало в този връх. Първият списък може да се замени с масив.
Основни операции с графи:

- създаване на граф;
- проверка за празен граф;

- търсене на връх или дъга в граф;
- добавяне на връх или дъга;
- премахване на връх или дъга;
- намиране на път между два върха с точно определена дължина, в частност с минимална или максимална дължина;
- намиране на циклите, в които участва даден връх на графа;
- обхождане на граф; под обхождане на граф разбираме посещаване на всеки връх точно веднъж и евентуално извършване на допълнителни действия свързани с конкретната задача; съществуват две основни стратегии за обхождане:
  - обхождане **в дълбочина** – ако се посети връх u, то непосредствено след това се посещава един от наследниците му v и останалите наследници на u се посещават след като бъдат посетени всички наследници на v;
  - обхождане **в широчина** – ако се посети връх v, то непосредствено след това се посещават всичките му наследници и тогава се преминава към техните наследници.

27. Дървета

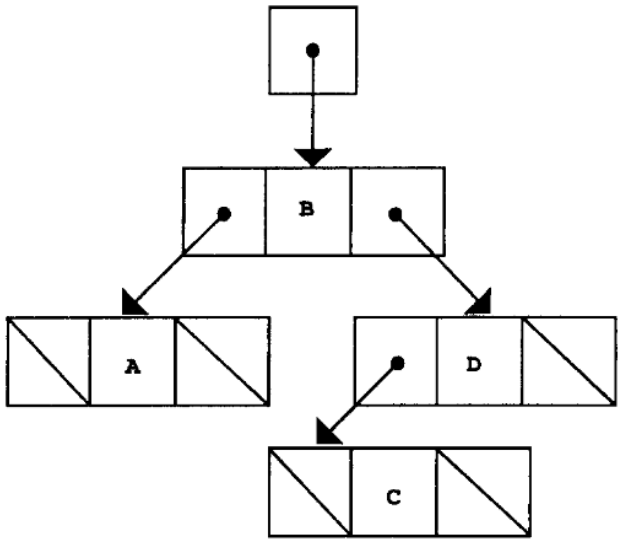
Ацикличен ориентиран граф, в който само един връх, наречен **корен,** има полустепен на входа 0 и всички останали върхове имат полустепен на входа 1 се нарича ориентирано **дърво.** Върховете с полустепен на изхода 0 се наричат **листа** на дървото. Дължината на пътя, измерена в брой дъги, от корена до даден връх се нарича **ниво** на този връх – този път при дърветата е единствен, тъй като всеки връх има единствен предшественик.
Два върха наричаме **братя**, ако са наследници на един и същи връх.
Пример.



В това дърво 1 е корен, на първо ниво са 2, 3, 4, на второ ниво са 5, 6, 7 и на трето ниво е 8 . Листата са 2, 6, 7, 8 . 2, 3, 4 са братя, 6 и 7 не са братя, въпреки че са на едно ниво.
Всеки вътрешен връх в едно дърво (който не е корен и не е лист) може да се разглежда като корен на дърво, което се състои от неговите наследници, от наследниците на техните наследници и т.н.
Например за горното дърво върхът 3 е корен на поддърво, което съдържа върховете 3, 5, 6, 8 .
Този факт дава основание да се даде алтернативна рекурсивна дефиниция за понятието дърво.
Дървото е крайно множество T от върхове, такива че:

- съществува точно един избран връх, наречен корен;
- всички останали върхове се съдържат в $k \geq 0$ на брой две по две непресичащи се подмножества $T_1, T_2, \ldots, T_k$, които се наричат поддървета на корена на T. Корените на тези поддървета наричаме наследници на корена на T.

Ако в дървото е въведена наредба на върховете по всяко ниво, дървото се нарича **наредено.**
Дърво в което всеки връх има полустепен на изхода 0, 1 или 2 се нарича **двоично (бинарно) дърво.**



Ще дадем аналогична на горната рекурсивна дефиниция за двоично дърво: двоично дърво е крайно множество от върхове, което е или празно, или се състои от един връх, наречен корен и

две непресичащи се двоични дървета, които се наричат ляво и дясно поддърво на корена на дървото.
Произволно дърво може да се представи чрез двоично по следния рекурсивен алгоритъм:

- коренът на двоичното дърво е коренът на изходното дърво;
- ляво поддърво на корена на двоичното дърво е преобразуваното най-ляво поддърво на корена на изходното дърво;
- дясно поддърво на корена на двоичното дърво е преобразуваното дърво на десния брат на корена.

Ще използваме рекурсивната дефиниция за двоично дърво, за да опишем структура за елемент на дърво.

```
template < class KeyType >
struct btree {
 KeyType key;
 btree < KeyType> *left, *right;
} ;
```

По горния алгоритъм елемент на произволно дърво може да се опише по следния начин:

```
template < class KeyType >
struct tree {
 KeyType key;
 tree < KeyType > *first_left, *next_right;
} ;
```

Рекурсивната дефиниция на двоично дърво предполага лесно използване на рекурсивни функции за операции с двоични дървета.

Пример е обхождането на дърво, за което съществуват три основни метода:

- **смесен ред** на обхождане (ляво – корен - дясно):
  - обхожда се лявото поддърво;
  - посещава се корена, т.е. извършват се действия свързани с конкретната задача;
  - обхожда се дясното поддърво;
- **низходящ ред** на обхождане (корен – ляво - дясно):

- посещава се корена;
- обхожда се лявото поддърво;
- обхожда се дясното поддърво;

- **възходящ ред** на обхождане (ляво – дясно - корен):
  - обхожда се лявото поддърво;
  - обхожда се дясното поддърво;
  - посещава се корена.

Ще реализираме три функции, които реализират трите вида обхождане. Функцията p (btree < KeyType > *t) реализира посещаване на корена на дървото, сочено от t.
Функция, която реализира смесен ред на обхождане на двоично дърво.

```
template < class KeyType >
void inorder (btree <KeyType> *t)
 { if (t != NULL)
    {  inorder (t -> left);
       p (t);
       inorder (t -> right);
```

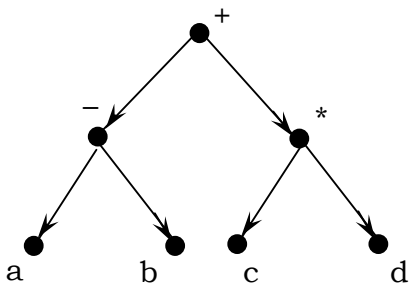
```
   }
 }
```
Функция, която реализира низходящ ред на обхождане на двоично дърво.
```
template < class KeyType >
void preorder (btree <KeyType> *t)
 { if (t != NULL)
   {  p (t);
      preorder (t -> left);
      preorder (t -> right);
   }
 }
```
Функция, която реализира възходящ ред на обхождане на двоично дърво.
```
template < class KeyType >
void postorder (btree <KeyType> *t)
 { if (t != NULL)
   {  postorder (t -> left);
```

```
        postorder (t -> right);
        p (t);
    }
}
```

Аритметичните изрази могат да се представят по следния начин:

- операциите са в нелиста;
- операндите са в листа;
- ако една операция се съдържа във връх v, то v има толкова наследника, колкото са аргументите на операцията и тя се прилага върху изразите, които отговарят на поддърветата на v.

Например изразът a – b + c*d се представя по следния начин:



Нека p (t) извежда ключа в съответния връх на дървото:

- при смесено обхождане получаваме изразът във възприетия инфиксен запис: a – b + c*d;
- при низходящо обхождане получаваме изразът в така наречения префиксен запис: + - a b * c d;
- при възходящо обхождане получаваме изразът в постфиксен запис или още **обратен полски запис**: a b – c d * +

Обратният полски запис се използва от компилаторите за пресмятане на изрази. Ще опишем алгоритъм за пресмятане на израз, който е в обратен полски запис: (разполагаме със стек, който в началото е празен)

1. i = 1;
2. Прочитаме i-тия символ от израза.
   Ако той е операнд, го включваме в стека.

Ако той е операция, изключваме двата операнда във върха на стека, прилагаме за тях операцията и полученото го включваме в стека. i = i + 1 Ако има непрочетени символи, премини към 2, в противен случай премини към 3 .

3. Резултатът е във върха на стека. Край.

Основни операции с дърво:

- създаване на дърво;
- проверка за празно дърво;
- достъп до корена на дървото;
- намиране на броя на поддърветата на даден връх;
- включване на поддърво в дадено дърво;
- изключване на поддърво от дърво;
- унищожаване на дърво;
- обхождане на дърво;
- включване на връх в двоично дърво за търсене;
- изключване на връх от двоично дърво за търсене.

28. Създаване на двоично дърво и включване на връх в двоично дърво за търсене

Ще използваме следната структура за елемент на дървото:

```
struct btree {
 int key;
 btree *left, *right;
} ;
```

Функция за създаване:

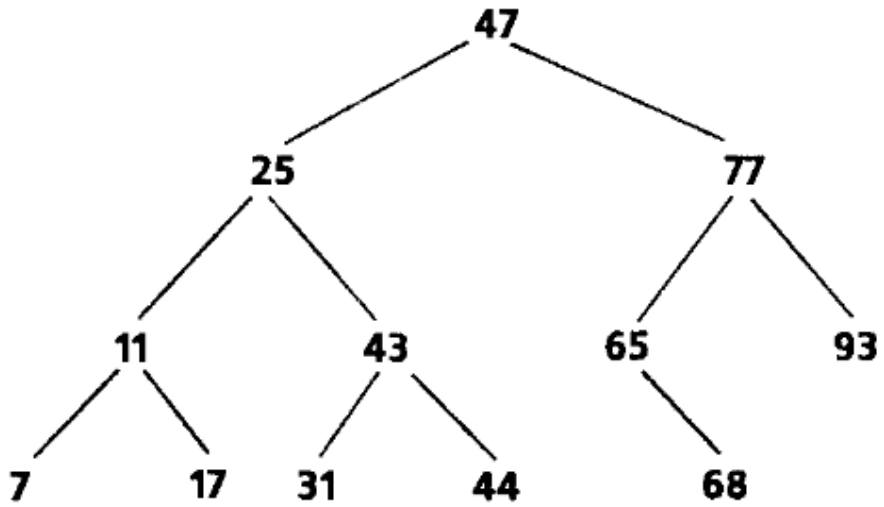
```
void create (btree *&t)
 { int x; char ch;
   cout << “Ключ на корен: “;
   cin >> x;
   if ( (t = new btree) == NULL)
    {  cout << “Няма достатъчно памет!\n” << endl;
       exit (1);
    }
   t -> left = t -> right = NULL; t -> key = x;
```

```
  cout << “Ляво поддърво с ключ на корена “ << x
       << “ Y/N? “;
  ch = cin.get ();
  if ( (ch = cin.get ()) == ‘Y’ || ch == ‘y’ )
     create (t -> left);
  cout << “Дясно поддърво с ключ на корена “ << x
       << “ Y/N? “;
  ch = cin.get ();
  if ( (ch = cin.get ()) == ‘Y’ || ch == ‘y’ )
     create (t -> right);
}
```

За реализиране на конкретно дърво може да се използва следната главна функция:

```
void main ()
 { btree *tree;
   create (tree);
   …
 }
```

Под **двоично дърво за търсене** разбираме двоично дърво, за всеки връх на което стойностите на всички ключове в лявото поддърво са по-малки от стойността на ключа във върха x и стойностите на всички ключове в дясното поддърво са не по-малки от стойността на ключа във върха x. Такова дърво още наричаме двоично дърво за търсене с повторения, тъй като в него се допускат върхове с еднакви ключове. Ако изискването за стойностите на ключовете на върховете в дясното поддърво на върха x е да са по-големи от стойността на ключа във върха x, тогава двоичното дърво за търсене е без повторения.



Ще опишем функция за добавяне на връх в двоично дърво за търсене с повторения.

```
template < class KeyType >
struct btree {
 KeyType key;
 btree < KeyType > *left, *right;
} ;
template < class KeyType >
void insert ( KeyType x, btree < KeyType > *t)
```

```
{ if (t == NULL)
   { if ( ( t = new btree < KeyType >) == NULL)
      { cout << “Няма достатъчно памет!” << endl;
        exit (1);
      }
    t -> left = t -> right = NULL; t -> key = x;
    return;
   }
  if ( x < t -> key ) insert (x, t -> left);
  else insert (x, t -> right);
}
```

За реализиране на конкретно дърво за търсене с повторения може да използваме следната главна функция:

```
void main ()
 { btree <int> *tree = NULL;
   insert (5, tree);
   insert (3, tree);
   insert (4, tree);
```

```
  …
}
```
Ще отбележим, че смесеното обхождане на дърво за търсене (с или без повторение) посещава върховете, сортирани във възходящ ред.

29. Обектно-ориентирана реализация на двоично дърво за търсене

Ще опишем клас за двоично дърво за търсене без повторения. Първо ще опишем клас за елемент от дървото.

файл treenode.h
```
#ifndef TREENODE_H
#define TREENODE_H
template < class NodeType > class Tree;
template < class NodeType >
class TreeNode {
```

```
 friend class Tree < NodeType >;
 public:
      TreeNode (const NodeType &d) :
        leftPtr (0), rightPtr (0), data (d) { }
      NodeType getData () const
         { return data; }
 private:
     NodeType data;
     TreeNode < NodeType > *leftPtr, *rightPtr;
} ;
#endif
```

Сега ще опишем клас за самото дърво. Той ще съдържа три функции за обхождане – в смесен, низходящ и възходящ ред. При обхождането стойностите на ключовете във върховете се извеждат на екрана.

Двоичното търсещо дърво, което реализираме, може да се използва за сортиране на данни – достатъчно е да се добавят всички елементи и след това да се извърши смесено обхождане.

файл tree.h

```
#ifndef TREE_H
#define TREE_H
#include <assert.h>
#include “treenode.h”
template < class NodeType >
class Tree {
 public:
    Tree ();
    void insertNode (const NodeType &);
    void preOrderTraversal () const;
    void inOrderTraversal () const;
    void postOrderTraversal () const;
 private:
    TreeNode < NodeType > *rootPtr;
    void insertNodeHelper (TreeNode <NodeType >*&, const NodeType
&);
```

```
    void preOrderHelper (TreeNode <NodeType >*) const;
    void inOrderHelper (TreeNode <NodeType >*) const;
    void postOrderHelper (TreeNode <NodeType >*) const;
} ;
template < class NodeType >
Tree < NodeType > :: Tree ()
 { rootPtr = 0; }
template < class NodeType >
void Tree < NodeType > :: insertNode ( const NodeType &value )
 { insertNodeHelper (rootPtr, value); }
template < class NodeType >
void Tree < NodeType > :: insertNodeHelper
      ( TreeNode < NodeType > *& ptr, const NodeType &value )
 { if (ptr == NULL)
    {  ptr = new TreeNode <NodeType> (value);
      assert (ptr != NULL);
    }
  else if (value < ptr -> data)
```

```
    insertNodeHelper ( ptr -> leftPtr, value );
  else if (value > ptr -> data)
    insertNodeHelper ( ptr -> rightPtr, value );
  else cout << value << " е дубликат!" << endl;
 }
template < class NodeType >
void Tree < NodeType > :: preOrderTraversal () const
 { preOrderHelper (rootPtr); }
template < class NodeType >
void Tree < NodeType > :: preOrderHelper
     (TreeNode <NodeType> *ptr) const
 { if (ptr != 0)
   {  cout << ptr -> data << ' ';
      preOrderHelper (ptr -> leftPtr);
      preOrderHelper (ptr -> rightPtr);
   }
 }
template < class NodeType >
```

```
void Tree < NodeType > :: inOrderTraversal () const
 { inOrderHelper (rootPtr); }
template < class NodeType >
void Tree < NodeType > :: inOrderHelper
      (TreeNode <NodeType> *ptr) const
 { if (ptr != 0)
    {  inOrderHelper (ptr -> leftPtr);
       cout << ptr -> data << ' ';
       inOrderHelper (ptr -> rightPtr);
    }
 }
template < class NodeType >
void Tree < NodeType > :: postOrderTraversal () const
 { postOrderHelper (rootPtr); }
template < class NodeType >
void Tree < NodeType > :: postOrderHelper
      (TreeNode <NodeType> *ptr) const
 { if (ptr != 0)
```

```
    {  postOrderHelper (ptr -> leftPtr);
       postOrderHelper (ptr -> rightPtr);
       cout << ptr -> data << ‘ ‘;
    }
 }
#endif
```

Ще опишем програма, която използва гореописаните класове.

<u>файл treetest.cpp</u>

```
#include <iostream.h>
#include <iomanip.h>
#include “tree.h”
void main ()
 { Tree < int > intTree;
   int intValue, i;
   cout << “Въведете 10 int – стойности:\n”;
   for (i = 0; i < 10; i++)
    { cin >> intValue;
```

```
   intTree.insertNode (intValue);
 }
cout << “Низходящо обхождане:\n“;
intTree.preOrderTraversal ();
cout << endl;
cout << “Смесено обхождане:\n“;
intTree.inOrderTraversal ();
cout << endl;
cout << “Възходящо обхождане:\n“;
intTree.postOrderTraversal ();
cout << endl;
Tree < double > doubleTree;
double doubleValue;
cout << “Въведете 10 double стойности:\n”
        << setiosflags (ios::fixed | ios::showpoint)
        << setprecision (1);
for (i = 0; i < 10; i++)
 { cin >> doubleValue;
```

```
    doubleTree.insertNode (doubleValue);
   }
  cout << “Низходящо обхождане:\n“;
  doubleTree.preOrderTraversal ();
  cout << endl;
  cout << “Смесено обхождане:\n“;
  doubleTree.inOrderTraversal ();
  cout << endl;
  cout << “Възходящо обхождане:\n“;
  doubleTree.postOrderTraversal ();
  cout << endl;
}
```

Резултат:
Въведете 10 int – стойности:
2 1 3 4 7 5 8 6 9 10
Низходящо обхождане:
2 1 3 4 7 5 6 8 9 10

Смесено обхождане:
1 2 3 4 5 6 7 8 9 10
Възходящо обхождане:
1 6 5 10 9 8 7 4 3 2
Въведете 10 double стойности:
12 45 22 74 43 98 77 55 33 66
Низходящо обхождане:
12.0 45.0 22.0 43.0 33.0 74.0 55.0 66.0 98.0 77.0
Смесено обхождане:
12.0 22.0 33.0 43.0 45.0 55.0 66.0 74.0 77.0 98.0
Възходящо обхождане:
33.0 43.0 22.0 66.0 55.0 77.0 98.0 74.0 45.0 12.00

Ще отбележим, че конкретният вид на двоичното дърво за търсене съществено зависи от реда по който се подават данните. Търсенето на елемент в дървото в средния случай се осъществява с $\log_2 n$ сравнения. Това означава, че в дърво с 1000 върха съществуването на елемент с даден ключ може да се

провери средно с около 10 сравнения, при дърво с 1000000 върха този брой е 20.
Съществува, обаче, най-лош случай, в който дървото се изражда в списък – когато елементите се подават в сортиран ред. Тогава сложността за търсене е линейна, т.е. от порядъка на броя на върховете на дървото.

30. Обработка на файлове с последователен достъп

В C++ всеки файл се разглежда като последователност от n байта с номера от 0 до n-1, като всеки файл завършва с маркер за край на файл EOF (end of file).
За да се извърши вход и изход с файлове в C++ в програмата трябва да се включат заглавните файлове ifstream.h и ofstream.h. Заглавният файл fstream.h включва и двата файла и някои други

допълнителни функции. Връзката между файловете и програмата се осъществява от потоците (обекти от посочените класове). Файловете се отварят чрез създаване на обекти от класовете ifstream, ofstream, fstream или чрез обръщение към вече създадени обекти.
За всеки файл, който предстои да бъде отворен, се задава режим за отваряне, като някои от възможностите са следните:

- ios::app - за добавяне на данни в края на файла без да се променят данните, които се намират вече в него;
- ios::in – отваряне на файл за въвеждане;
- ios::out – отваряне на файл за извеждане;
- ios::binary – отваряне на файл за двоичен вход или изход. Използва се с другите режими чрез използване на операция побитово или (‘|’).

Отварянето на файл за извеждане става чрез създаване на обект от класа ofstream, като при създаването се предават два параметъра на съответния конструктор на класа – името на файла и режима за отваряне.

За обект от класа ofstream режима за отваряне може да бъде ios::out или ios::app. В режима ios::out съществуващите файлове се унищожават и създават наново, а несъществуващите файлове се създават. Аргументът режим за отваряне по премълчаване е ios::out. Обект от класа ofstream може да бъде създаден без да се отваря файл – за целта не се подават аргументи на конструктора. Чрез този обект по-късно може да бъде отворен файл чрез функцията-елемент open, която има същите параметри като конструктора за отваряне.
Операцията отрицание (‘!’) е предефинирана в класа ios и се използва върху обекти от този клас. Операцията връща ненулева стойност (true) когато при отварянето на файла с помощта на обекта са се установили флаговете failbit или badbit. Някои възможни причини за това са опит за отваряне на несъществуващ файл за четене, опит за създаване на файл, когато на диска няма свободно място и т.н.
Операцията за преобразуване към тип void* също е предефинирана в класа ios. Тя преобразува обект от клас ios към

указател, който получава стойност 0, ако за обекта са установени флаговете failbit или badbit. Например в условието на цикъла while (cin >> c) { …} , обектът cin автоматично се преобразува към void*. Това условие е истина докато за cin не е установен някой от флаговете failbit или badbit. Въвеждането на признак за край на файла (в DOS и в Windows с клавишна комбинация Ctrl+Z) води до установяване на failbit за cin. В този случай условието не е истина и цикълът ще приключи.
При излизане от областта на действие на обект, свързан с файл, автоматично се извиква неговия деструктор, който освен че унищожава обекта, затваря файла свързан с него (ако има такъв отворен файл). Затварянето на файл може да става и явно чрез функцията-елемент close без параметри. Препоръчва се явно затваряне, тъй като това води до икономия на ресурси и до по-голяма яснота в програмата.
Ще разгледаме програма за създаване на файл с последователен достъп, който съдържа записи, съставени от номер на сметка,

баланс и име на клиента. Предполага се, че файлът се създава и обработва в сортиран ред на номерата на сметките.

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
void main ()
 { ofstream outclientFile ("clients.dat", ios::out);
   if (!outclientFile)
    { cerr << "Файлът не може да бъде отворен!" << endl;
      exit (1);
    }
   cout << "Въведете записи от номер " <<
      "на сметката, име и баланс; Ctrl+Z за край:\n? ";
   int account; char name[30]; double balance;
   while (cin >> account >> name >> balance)
    {
      outclientFile << account << ' ' << name << ' ' << balance << endl;
      cout << "? ";
```

```
    }
   outclientFile.close ();
 }
```

Отварянето на файл за въвеждане става чрез създаване на обект от класа ifstream, като при създаването се предават два параметъра на съответния конструктор на класа – името на файла и режима за отваряне. По премълчаване режимът за отваряне е ios::in, т.е. режим за въвеждане. Подобно на случая с обекти от класа ofstream, обекти от класа ifstream могат да бъдат създавани без да се отваря файл едновременно с това. По-късно в програмата чрез тези обекти могат да се отварят файлове с помощта на функцията-елемент open.

Обекти от класа ifstream могат да се използват като условие в цикъл, например: while (обект_от_клас_ifstream >> c) { …} . При компилиране на условието се използва операцията за преобразуване на обект от клас ios към указател от тип void *, който приема нулева стойност, ако за обекта са установени флагове failbit или badbit. Достигането до края на файла включва

флага failbit, така че цикълът ще приключи, ако се достигне до края на файла.
При последователно търсене на данни във файл често се налага последователна обработка на файла, като на всяка стъпка се започва от началото на файла. За целта всеки обект от класа istream има така наречения указател get на позицията за въвеждане от файла и всеки обект от класа ostream има указател put на позицията за извеждане във файла. Стойността на тези указатели е поредният номер на байт във файла. Класовете istream и ostream съдържат функции-елементи за задаване на стойност на указателя на позицията във файла.
Тези функции са следните:

- seekg – за позициониране при въвеждане от файл (от класа istream);
- seekp – за позициониране при извеждане във файл (от класа ostream).

Всяка от тези функции има по два аргумента. Първият аргумент е от тип long int и задава новата стойност за указателя на позицията

във файла, а вторият аргумент може да приема една от следните стойности:

- ios::beg – позициониране относно началото на файла;
- ios::end – позициониране относно края на файла;
- ios::cur – позициониране относно текущата позиция.

По премълчаване стойността на този аргумент е ios::beg.
Примери:
обект_от_клас_istream.seekg (0) задава стойност 0 на указателя на позицията във файла, който е свързан с обекта, т.е. това води до позициониране в началото на файла.
обект_от_клас_istream.seekg (n) води до позициониране на байт с номер n.
обект_от_клас_istream.seekg (n, ios::cur) води до позициониране с n байта напред, т.е. спрямо текущата позиция.
обект_от_клас_istream.seekg (n, ios::end) води до позициониране с n байта преди края на файла.
обект_от_клас_istream.seekg (0, ios::end) води до позициониране в края на файла.

Аналогични примери могат да се направят с обект от клас ostream и функцията-елемент seekp.
Функциите-елементи tellg от класа istream и tellp от класа ostream връщат текущите стойности на указателите get и put. Те връщат стойност от тип long int.
Текстовите файлове с последователен достъп съдържат записи с неограничена дължина. Затова обновяване на място не може да се извършва, т.е. възможно е единствено презаписване на целия файл или присъединяване на данни в края на файла.
Примерна програма:

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include <iomanip.h>
void main ()
 { ifstream inclientFile (“clients.dat”, ios::in);
   if (!inclientFile)
    { cerr << “Файлът не може да бъде отворен!” << endl;
```

```
    exit (1);
  }
  int account;
  char name[30];
  double balance;
  cout << setiosflags (ios::left) << setw (10) << "Сметка" << setw (30)
       << "Име" << "Баланс" << endl;
  while (inclientFile >> account >> name >> balance)
    cout << setiosflags (ios::left) << setw (10) << account << setw (30)
         << name << setprecision (2) << balance << endl;
}
```

Резултат:

Сметка	Име	Баланс
123	Ivanov	3.4e+002
342	Petrov	3.3e+002
655	Georgiev	5.5e+002

31. Обработка на файлове с пряк достъп

Основният метод при обработване на файлове с пряк достъп е използването на записи с фиксирана дължина. В този случай, програмата може бързо да определи точното местоположение на даден запис спрямо началото на файла.
При обработка на файлове с пряк достъп записването, обновяването и изключването на данни може да стават без да се разрушават другите данни или да се презаписва целия файл.
Функцията-елемент write от класа ostream извежда фиксиран брой байтове в зададен поток, започвайки от дадено място в оперативната памет. Когато потокът е свързан с файл, данните се извеждат започвайки от позицията, определена с помощта на указателя за позиция put.
Първият аргумент на write е от тип const char * и определя мястото в оперативната памет, откъдето ще се извежда. Вторият аргумент е от цял тип и определя броя на байтовете за извеждане.

Функцията-елемент read от класа istream въвежда фиксиран брой байтове от зададен поток в област от оперативната памет.
Ако потокът е свързан с файл, байтовете се въвеждат започвайки от позицията, определена с помощта на указателя get.
Първият аргумент на read е от тип char * и определя мястото в оперативната памет, където ще се въвеждат байтовете. Вторият аргумент е от цял тип и определя броя на байтовете за въвеждане.
Ако се планира програма, която чете неформатирани данни, записани с write, то е необходимо тя да бъде компилирана и изпълнена в операционна система, съвместима с програмата, която е записала данните.
Ще опишем програма за обработка на сметки, която може да съхрани до 100 записа с фиксирана дължина за компания с до 100 клиенти.
Програмата ще обновява сметки, ще унищожава сметки, ще вмъква нови сметки и ще извежда списък на всички сметки във форматиран текстов файл.

Ще напишем заглавен файл, който определя типа на структурата.
Файл clientdata.h

```
#ifndef CLIENTDATA_H
#define CLIENTDATA_H
struct clientData {
 int account;
 char firstName[15], lastName[15];
 double balance;
} ;
#endif
```

Ще напишем програма за инициализиране на файла със записите.
Файл clientinit.cpp

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
```

```
#include “clientdata.h”
void main ()
 { ofstream outCredit ( “credit.dat”, ios::binary);
   if (!outCredit)
    { cerr << “Файлът не може да бъде отворен!\n”;
      exit (1);
    }
   clientData blankClient = { 0, “”, “”, 0.0 };
   for (int i = 0; i < 100; i++)
     outCredit.write (reinterpret_cast < const char *> (&blankClient),
                          sizeof (clientData));
 }
```

Всеки незапълнен запис от типа на структурата съдържа номер 0, празни низове за име и фамилия и баланс 0. Във файла първоначално се задават 100 незапълнени записа и по-нататък програмата лесно ще може да определи дали някой от записите е запълнен или не.

В извикването на функцията-елемент write се извършва преобразуване с операцията reinterpret_cast, тъй като аргументът (&blankClient) е от тип clientData* и така извикването се компилира без синтактична грешка.
Ще напишем програма за директно записване във файла с пряк достъп.
Файл clientadd.cpp

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include “clientdata.h”
void main ()
 { ofstream outCredit ( “credit.dat”, ios::binary);
   if (!outCredit)
    { cerr << “Файлът не може да бъде отворен!\n”;
      exit (1);
    }
```

```
   cout << “Въведете номер на сметка от 1 до 100, 0 за край: “;
   clientData client;
   cin >> client.account;
   while (client.account > 0 && client.account <= 100)
    { cout << “Въведете името и баланса: “;
      cin >> client.firstName >> client.lastName >> client.balance;
      outCredit.seekp ( (client.account-1)*sizeof (clientData));
      outCredit.write (reinterpret_cast < const char *> (&client),
                              sizeof (clientData));
      cout << “Въведете номер на сметка от 1 до 100, 0 за край: “;
      cin >> client.account;
    }
 }
```

Програмата записва данни във файла чрез комбинация от функциите seekp и write – функцията seekp установява указателя put на зададена позиция във файла и след това данните се извеждат с write.

От номера на сметката се изважда 1, тъй като номерацията на байтовете във файла започва от 0 за запис 1.
Накрая ще напишем програма, която извежда непразните записи.
файл clientprint.cpp

```
#include <iostream.h>
#include <iomanip.h>
#include <fstream.h>
#include <stdlib.h>
#include “clientdata.h”
void outputLine (ostream &, const clientData &);
void main ()
 {  ifstream inCredit (“credit.dat”, ios::in);
    if (!inCredit)
     { cerr << “Файлът не може да бъде отворен!”;
       exit (1);
     }
    cout << setw (10) << “Сметка:“
```

```
        << setw (16) << “Име:“ << setw (16) << “Фамилия:”
        << setw (10) << “Баланс:” << endl;
   clientData client;
   inCredit.read (reinterpret_cast < char * > (&client), sizeof
(clientData));
   while (inCredit && !inCredit.eof ())
    { if (client.account != 0)
      outputLine (cout, client);
     inCredit.read (reinterpret_cast <char * > (&client), sizeof
(clientData));
    }
}
void outputLine (ostream &output, const clientData &c)
{ output << setw (10) << c.account
        << setw (16) << c.firstName << setw (16) << c.lastName
        << setprecision (2) << setiosflags (ios::fixed | ios::showpoint)
        << setw (10) << c.balance << endl;
}
```

Програмата последователно чете записи от файла и извежда тези от тях, които са непразни. Четенето с функцията read приключва, ако се достигне края на файла (тогава !inCredit.eof () става лъжа) или ако възникне грешка по време на четене (тогава inCredit става лъжа).
Записите се извеждат във възходящ ред по номера на сметката – това е следствие от съхраняването на записите във файл с пряк достъп.
Обработката на файлове с пряк достъп е по-бърза, но изисква повече дискова памет, тъй като се налага да се съхраняват незапълнени записи.

32. Макроопределения. Включване на файлове.

Препроцесорът е програма за подготовка за процеса на компилация. Тази възможност е характерна за езиците за системно програмиране. Обикновено в средите за програмиране ние извикваме компилатора, а той препроцесора.
Препроцесорът заменя всички коментари с интервали. Те не влияят на ефективността на програмата, тъй като компилатора ги пропуска.
В C и C++ ние можем да задаваме специални инструкции към препроцесора. Това става с помощта на така наречените директиви на препроцесора. Общият вид на една директива е следния:
#ключова_дума вид_на_заместването
Ключовата дума се записва с малки букви, а вида на заместването определя какви промени ще се извършват. В края на директивата не се поставя ';'. Директивите се записват самостоятелно на един ред на произволно място в програмата.

Възможно е директивите да се записват на няколко реда, но в края на всеки ред (с изключение на последния) трябва да поставим символа ‘\’.
Например:
#ключова_дума\
вид_на_заместването
Директивите не са част от C и C++ - те задават определени действия, които се извършват върху първичния файл в ASCII код. Макроопределения се извършват с помощта на директивата define. Синтаксисът е следния:
#define име_на_макрос низ
Името е произволен идентификатор. След като срещне директивата define, препроцесорът замества всяко по-нататъшно срещане на името на макроса с указания низ. Този процес наричаме разширяване на макроса. Прието е имената на макросите да се записват с главни букви за да се отличават от останалите елементи в програмата. Примери:
#define PI 3.14159

```
#define LIMIT 100
int masiv[LIMIT];
for (int i = 0; i < LIMIT; i++) { masiv[i] = …; }
```

Възможно е низът в директивата define сам по себе си да съдържа макрос. Например:

```
#define EXPR (PI*r*r)
```

Това е възможно, тъй като след като извърши разширяването на целия текст, препроцесорът преглежда наново файла дали не са възможни още разширявания.

Разширяването на EXPR в програмата ще се извършва на две стъпки: EXPR → (PI*r*r) → (3.14159*r*r).

Ако името на макроса се среща в символна или низова константа, то при изпълнение на define препроцесорът не разширява това срещане.

Едно и също име на макрос може да бъде дефинирано многократно в програмата. Новото определение отменя действието на последното въведено определение.

Действието на define може да бъде отменено с помощта на директивата undef със следния синтаксис:
#undef име_на_макрос
Името на макроса определя макроса, който искаме да отменим. След тази директива този макрос престава да се разширява от препроцесора. Пример:
#define GLOB 10
…
#define GLOB 500
…
#undef GLOB
…
Между двете директиви define GLOB ще се замества с 10, а след втората директива define GLOB ще се замества с 500 до директивата undef, а след нея макросът GLOB вече не е дефиниран.
Възможно е да се дефинира макрос без да се задава низ.

Например:
#define TEST
Такива макроси се наричат празни и те се използват при условна компилация. Празните макроси не се разширяват, т.е. ако в програмата препроцесорът срещне TEST, той няма да го разшири и компилаторът ще даде синтактична грешка.
При разработване на програми много често възниква необходимостта от обединяването на няколко първични файла в един. Това се постига с директивата include със следния синтаксис:
#include “име_на_файл”
#include <име_на_файл>
Когато препроцесорът срещне директива include, той открива посочения първичен файл (в ASCII код) и след това на мястото на директивата записва текста на файла. Самият файл може също да съдържа директиви на препроцесора – те ще бъдат обработени при следващото разширение.

Разликата в синтаксиса на include е в начина, по който се търси файла. Ако файлът е записан с “”, той се търси първо в текущата директория и след това в include-директорията, т.е. директорията със стандартните файлове за включване. Ако файлът е записан с <>, той директно се търси в include-директорията.

33. Макроопределения с аргументи.

Макроопределения с аргументи се дефинират по следния начин:
#define име_на_макрос(списък_от_формални_аргументи) низ
Между името на макроса и отварящата скоба на списъка от аргументи не трябва да има интервали – ако има интервал препроцесорът ще интепретира директивата като макроопределение без аргументи. Имената на формалните аргументи са валидни само за дефиницията и те могат да съвпадат с други идентификатори в програмата без това да води до грешка.

Когато използваме името на макроса в програмния текст, фактическите аргументи могат да бъдат произволна последователност от символи. Когато препроцесорът срещне в програмата макрос с аргументи, той го обработва по следния начин:

1. в низа на директивата define формалните аргументи се заместват с фактическите аргументи;
2. името на макроса, следвано от списъка с фактическите аргументи се замества с променения в 1. низ.

Примери:

```
#define PI 3.14159
#define CIRCLE(r) (PI*(r)*(r))
…
double x, y, z;
…
z = CIRCLE (x);
```

Заместването се извършва на две стъпки:

CIRCLE (x) → (PI*(x)*(x)) → (3.14159*(x)*(x)).

Скобите, които заграждат формалния аргумент r, са необходими. Например, ако запишем CIRCLE (x+1) и те липсваха, препроцесорът би извършил следното заместване:
CIRCLE (x+1) → (PI*x+1*x+1) → (3.14159*x+1*x+1), което очевидно не е желания резултат.
Скобите, които заграждат целия низ за заместване, също са необходими. Например, ако запишем 10/CIRCLE (x) и те липсваха, препроцесорът би извършил следното заместване:
10/CIRCLE (x) → 10/PI*(x)*(x) → 10/3.14159*(x)*(x), което очевидно не е желания резултат.
Една функция може да се реализира като макрос. Например:
#define sqr(x) ((x)*(x))
Възможно е да възникнат странични ефекти във връзка с реализирането на функция като макрос.
Например, ако запишем sqr (y++) ние очакваме функцията да върне y*y и след това y да се увеличи с 1. Препроцесорът обаче извършва следното заместване:

sqr (y++) → ((y++)*(y++)) – тук функцията отново връща y*y, но y се увеличава с 2.

Макросът с аргументи е алтернатива на използването на функции. Когато се прави избор каква реализация да се използва, трябва да се имат предвид следните предимства и недостатъци:

- макросът се изпълнява по-бързо от функцията, тъй като при него няма предаване на аргументи по време на изпълнение на програмата;
- всяко обръщение към макроса се замества с програмен текст, докато описанието на функцията като програмен текст е единствено за всички обръщения;
- макросите могат да породят странични ефекти;
- при обръщение към макрос липсва какъвто и да е контрол върху съответствието на типовете на формалните и фактическите аргументи. Това се прави при функциите с помощта на прототипа;

- макросът води до разширение в първичния файл преди компилацията, докато функцията е постоянна програмна единица обработвана от компилатора – това води до трудно откриване на грешки при работа с големи макроси.

34. Условна компилация.

Препроцесорът на C и C++ има набор от директиви, с помощта на които може да се определи алтернативно дали даден фрагмент от програмния текст да се компилира или не. Тези директиви се наричат директиви за условна компилация. Те реализират механизъм за алтернативен избор от вида if…else…endif, подобен на този при условните оператори.
Частта if има вида:

```
#ifdef име_на_макрос
  текст на програма на C или C++
#ifndef име_на_макрос
  текст на програма на C или C++
```

#if константен_израз
  текст на програма на C или C++
Името на макроса в директивите ifdef, ifndef и константния израз в директивата if са условието, което определя дали ще се компилира текста след if или текста след else.
Частта else има вида:
#else
  текст на програма на C или C++
Частта endif има вида:
#endif
С помощта на частта endif се определя края на конструкцията if…else….
Ако преди частта if в програмата е дефиниран макрос с име, съвпадащо с името в директивата ifdef, условието е изпълнено. Ако няма такъв макрос, условието не е изпълнено.
Ако преди частта if в програмата е дефиниран макрос с име, съвпадащо с името в директивата ifndef, условието не е изпълнено. Ако няма такъв макрос, условието е изпълнено.

Тъй като в тези случаи е важен само факта дали макросът е дефиниран или не, за целите на условната компилация могат да се използват празни макроси.
За директивата if условието е константен израз, който се пресмята по общоприетите правила. Ако стойността му е различна от 0, то условието е изпълнено, в противен случай условието не е изпълнено.
Препроцесорът проверява дали условието в директивите ifdef, ifndef или if е изпълнено. Ако то е изпълнено, програмният текст в частта if се включва в програмата, а този в частта else се пропуска. Ако условието не е изпълнено, програмният текст в частта if се пропуска, а този в частта else се включва в програмата.
Да разгледаме един пример. Един от възможните начини за тестване на програмата е да се извеждат междинни резултати за контрол. След като завърши тестването, извеждането на междинни резултати вече не е нужно, така че самото извеждане може да се включи в директиви за условна компилация по следния начин:

```
#ifdef DEBUG
#define PRINT(x) x
#else
#define PRINT(x)
#endif
```

На етапа тестване на програмата, в началото се дефинира макросът DEBUG чрез директивата #define DEBUG. Това води до включване в програмата на всички междинни резултати, реализирани чрез макроса PRINT.

Например: PRINT (cout << x << y << z;)

След като се изчистят грешките в програмата, директивата #define DEBUG се изтрива или се отменя чрез директивата #undef DEBUG

При повторно компилиране ще се изпълни директивата #else и всички макроси PRINT ще се заменят с празен низ.

Предимството на този подход е, че само с една директива #ifdef се управляват извежданията на всички междинни резултати.

Друг начин за тестване чрез използване на условна компилация е следния: при всяко извеждане да използваме ifdef.

```
#ifdef DEBUG
cout << x << y << z;
#endif
```

Директивите за условна компилация могат да се използват и без else, също така те могат да бъдат вложени.

Директивите за условна компилация също могат да се използват, ако в програмата има машинно зависими части. Например:

```
#define PC_16 //шестнадесетбитова реализация
#ifdef PC_16
#define INT 16
#else
#define INT 32
#endif
```