Глава 2

Основни елементи от програмирането на C++

C++ е език за обектно-ориентирано програмиране. Създаден е от Бярн Страуструп от AT&T Bell Laboratories в края на 1985 година. C++ е разширение на езика C в следните три направлнения:

- създаване и използване на абстрактни типове данни;
- обектно-ориентирано програмиране;
- подобрения на конструкции на езика C (производни типове, наследяване, полиморфизъм).

През първите шест месеца след описанието му се появиха над 20 търговски реализации на езика, предназначени за различни компютърни системи. От тогава до сега C++ се разраства чрез добавяне на много нови функции и затова процесът на стандартизацията му продължава и до момента. C++ е пример за език, който с времето расте и се развива. Всеки път, когато потребителите му са забелязвали някакви пропуски или недостатъци, те са го обогатявали със съответните нови възможности.

За разлика от C++, езикът Паскал е създаден планомерно главно за целите на обучението. Проф. Вирт добре е обмислил и доказал езика. Тъй като Паскал е създаден с ясна цел, отделните му компоненти са логически свързани и лесно могат да бъдат комбинирани. Разрастващите се езици, към които принадлежи C++ са доста объркани тъй като хора с различни вкусове правят различни нововъведения. Освен това, заради мобилността на програмите, не е възможно премахването на стари конструкции, даже да съществуват удобни техни подобрения. Така разрастващият се C++ събира в себе си голям брой възможности, които не винаги добре се съвместяват.

Езиците, създадени от компетентни хора, по принцип са лесни за научаване и използване. Разрастващите се езици обаче държат

монопола на пазара. Сега C++ е водещия език за програмиране с общо предназначение. Лошото е, че не е много лесен за усвояване, има си своите неудобства и капани. Но той има и огромни приложения – от програми на ниско, почти машинно ниво, до програми от най-висока степен на абстракция.

Целта на настоящия курс по програмиране е не да ви научи на всички възможности на C++, а на изкуството и науката програмиране.

При началното запознаване с езика, възникват два естествени въпроса:

- Какво е програма на C++ и как се пише тя?
- Как се изпълнява програма на C++?

Ще отговорим на тези въпроси чрез пример за програма на C++, след което ще дадем някои дефиниции и основни означения.

2.1. Пример за програма на C++

Задача 1. Да се напише програма, която намира периметъра и лицето на правоъгълник със страни 2,3 и 3,7.

Една програма,която решава задачата е следната:

```
Program Zad1.cpp
#include <iostream.h>
int main()
{double a = 2.3;
  double b = 3.7;
  double p, s;
  /* намиране на периметъра на правоъгълника */
  p = 2*(a+b);
  /* намиране на лицето на правоъгълника */
  s = a*b;
   /* извеждане на периметъра */
  cout << "p=  " << p << "\n";
/* извеждане на лицето */
  cout << "s=   " << s << "\n";
  return 0;
}
```

Първият ред

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

е **директива към компилатора на C++.** Чрез нея към файла, съдържащ програмата Zad1.cpp, се включва файлът с име iostream.h (При някои реализации на C++ разширението “.h” се пропуска). Този файл съдържа различни дефиниции и декларации, необходими за реализациите на операциите за поточен вход и изход. В програма Zad1.cpp се нуждаем от тази директива заради извеждането върху екрана на периметъра и лицето на правоъгълника.

Конструкцията

```
int main()
{ …
   return 0;
}
```

дефинира **функция**, наречена main (главна). Всяка програма на C++ трябва да има функция main. Повечето програми съдържат и други функции освен нея.

Дефиницията на main започва с **думата** int (съкращение от integer), показваща, че main връща цяло число, а не дроб или низ, например. Между фигурните скобки { и } е записана редица от дефиниции и оператори, която се нарича **тяло на функцията.** Компонентите на тялото се отделят със знака ; и се изпълняват последователно. С оператора return се означава краят на функцията. Стойността 0 означава, че тя се е изпълнила успешно. Ако програмата завърши изпълнението си и върне стойност различна от 0, това означава, че е възникнала грешка.

Конструкциите

```
double a = 2.3;
double b = 3.7;
double p, s;
```

дефинират **променливите** a, b, p и s от реалния тип double, като в първите два случая се дават начални стойности на a и b (2.3 и 3.7 съответно). Казва се още, че **a и b са инициализирани съответно с 2.3 и 3.7.**

Променливата е място за съхранение на данни, което може да съдържа различни стойности по време на изпълнение на програмата. Означава се чрез редица от букви, цифри и долна черта, започваща с буква или долна черта. Променливите имат три характеристики: **тип**, **име** и **стойност.** Преди да бъдат използвани, трябва да бъдат дефинирани.

C++ е строго типизиран език за програмиране. Всяка променлива има тип, който *явно* се указва при дефинирането й. Пропускането на типа на променливата води до сериозни грешки. Фиг. 1. илюстрира непълно дефинирането на променливи.

Дефиниране на променливи

Синтаксис

<име_на_тип> <променлива> [= <израз>]
{, <променлива> [= <израз>] };

където

<име_на_тип> е дума, означаваща име на тип като int, double и др.;

<израз> е правило за получаване на стойност – цяла, реална, знакова и др. тип, съвместим с <име_на_тип>.

Семантика

Дефиницията свързва променливата с множеството от допустимите стойности на типа, от който е променливата или с конкретна стойност от това множество. За целта се отделя определено количество оперативна памет (толкова, колкото да се запише най-голямата константа от множеството от допустимите стойности на съответния тип) и се именува с името на променливата. Тази памет е с неопределено съдържание или съдържа стойността на указания израз, ако е направена инициализация.

Не се допуска една и съща променлива да има няколко дефиниции в рамките на една и съща функция.

Фиг. 1.

В случая на програмата Zad1.cpp за a, b, p и s се отделят по 8 байта оперативна памет и

ОП

a	b	p	s
2.3	3.7	-	-
8 байта	8 байта	8 байта	8 байта

Неопределеността на p и s ще означаваме с -.

След дефинициите на променливите a, b, p и s е разположен коментарът

/* намиране на периметъра на правоъгълника */

Той е предназначен за програмиста и подсеща за смисъла на следващото действие.

Коментарът (Фиг. 2) е произволен текст, ограден със знаците /* и */. Игнорира се напълно от компилатора.

Коментар

Синтаксис

<коментар> ::= /* <редица_от_знаци> */

Семантика

Пояснява програмен фрагмент. Предназначен е за програмиста. Игнорира се от компилатора на езика.

Фиг. 2.

Конструкциите

```
p = 2*(a+b);
s = a*b;
```

са **оператори за присвояване на стойност** (Фиг. 3). Чрез тях променливите p и s получават текущи стойности. Операторът

```
p = 2*(a+b);
```

пресмята стойността на аритметичния израз 2*(a+b) и записва полученото реално число (в случая 12.0) в паметта, именувана с p. Аналогично, операторът

```
s = a*b;
```

пресмята стойността на аритметичния израз a*b и записва полученото реално число (в случая 8.51) в паметта, именувана със s.

По-подробно ще разгледаме този оператор в следващата глава. На този етап оставяме с интуитивната представа за <израз>. Ще се ограничим с лява страна от вида променлива. Ще отбележим само, че езикът C++ поддържа псевдоними, което дава възможност лявата страна на оператора за присвояване да бъде израз, чиято стойност е псевдоним на модифицируем обект.

Оператор за присвояване

Синтаксис

<променлива> = <израз>;

като <променлива> и <израз> са от един и същ тип.

Семантика

Пресмята стойността на <израз> и я записва в паметта, именувана с променливата от лявата страна на знака за присвояване =.

Фиг. 3.

Да се върнем към дефинициите на променливите a и b и операторите за присвояване и return на Zad1.cpp. Забелязваме, че в тях са използвани два вида числа: **цели** (2 и 0) и **реални** (2.3 и 3.7). Целите числа се записват като в математиката, а при реалните, знакът запетая се заменя с точка. Умножението е отбелязано със *, а събирането – с +. Забелязваме също, че изразите 2*(a+b) и a*b са реални, каквито са и променливите p и s от левите страни на знака = в операторите за присвояване.

Конструкциите

```
cout  << "p=  "  << p  <<  "\n";
cout  << "s=  "  << s   <<  "\n";
```

са оператори за извеждане. Наричат се още оператори за поточен изход. Еквивалентни са на редицата от оператори

```
cout  << "p=  ";
cout  << p;
cout  << "\n";
cout  << "s=  ";
cout  << s;
cout  << "\n";
```

Операторът << означава “изпрати към”. Обектът (променливата) cout (произнася се “си-аут”) е името на стандартния изходен поток, който обикновено е екрана или прозорез на екрана.

Редица от знаци, оградена в кавички, се нарича **знаков низ**, или **символен низ**, или само **низ.** В програмата Zad1.cpp “p= “ и “s= “ са низове. Низът “\n“ съдържа двойката знаци \ (backslash) и n, но те представляват един-единствен знак, който се нарича **знак за нов ред.** Операторът

```
   cout  << "p=  ";
```
извежда върху екрана низа
```
   p=
```
Операторът
```
   cout  << p;
```
извежда върху екрана стойността на p, а
```
   cout  << "\n";
```
премества курсора на следващия ред на екрана, т.е. указва следващото извеждане да бъде на нов ред.

Фиг. 7. показва по-детайлно синтаксиса и семантиката на оператора за извеждане.

Изпълнение на Zad1.cpp

След обработката на директивата
```
   #include <iostream.h>
```
файлът iostream.h е включен във файла, съдържащ функцията main на Zad1.cpp. Изпълнението на тялото на main започва с изпълнение на дефинициите
```
   double a = 2.3;
   double b = 3.7;
   double p, s;
```
в резултат, на което в ОП се отделят по 8 байта за променливите a, b, p и s, т.е.

ОП

a	b	p	s
2.3	3.7	-	-

Коментарите се пропускат. След изпълнението на операторите за присвояване:
```
   p = 2*(a+b);
   s = a*b;
```
променливите p и s се свързват с 12.0 и 8.51 съответно, т.е.

ОП

a	b	p	s
2.3	3.7	12.0	8.51

Операторите
```
   cout  << "p=  "  <<  p  <<   "\n";
   cout  << "s=  "  <<  s   <<   "\n";
```
извеждат върху екрана на монитора
```
   p= 12
```

```
s= 8.51
```

Изпълнението на оператора

```
return 0;
```

преустановява работата на програмата сигнализирайки, че тя е завършила успешно.

Забележка: Реалните числа се извеждат с възможно минималния брой знаци. Така реалното число 12.0 се извежда като 12.

В същност, описаните действия се извършват над машинния еквивалент на програмата Zad1.cpp. А как се достига до него?

Изпълнение на програма на езика C++

За целта се използва някаква среда за програмиране на C++. Ние ще изплзваме Visual C++ версия 6.0.

Изпълнението се осъществява чрез преминаване през следните стъпки:

1. Създаване на изходен код

Чрез текстовия редактор на средата, текстът на програмата се записва във файл. Неговото име се състои от две части – име и разширение. Разширението подсказва предназначението на файла. Различно е за отделните реализации на езика. Често срещано разширение за изходни файлове е “.cpp” или “.c”.

Примерната програма е записана във файла Zad1.cpp.

2. Компилиране

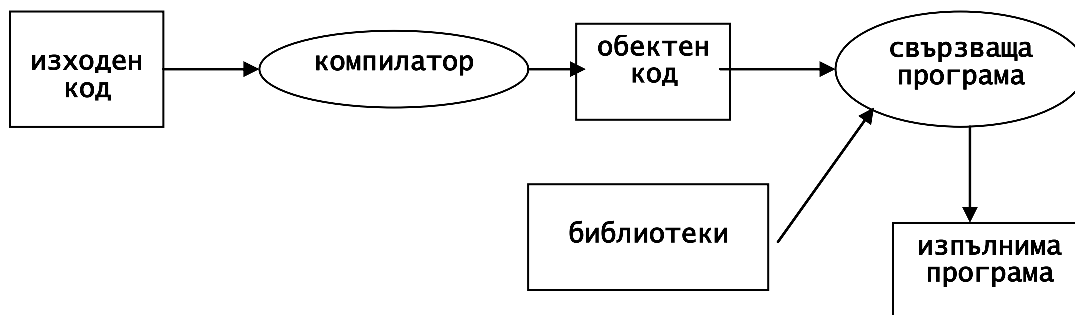
Тази стъпка се изпълнява от компилатора на езика. Първата част от работата на компилатора на C++ е откриването на грешки – синтактични и грешки, свързани с типа на данните. Съобщението за грешка съдържа номера на реда, където е открита грешка и кратко описание на предполагаемата причина за нея. Добре е грешките да се корегират в последователността, в която са обявени, защото една грешка може да доведе до т. нар. “каскаден ефект”, при който компилаторът открива повече грешки, отколкото реално съществуват. Коригираният текст на програмата трябва да се компилира отново. Втората част от работата на компилатора е превеждане (транслиране) на изходния (sourse) код на програмата в т. нар. **обектен код.** Обектният код се състои от машинни инструкции и информация за това, как да се зареди програмата в ОП, преди да започне изпълнението й. Обектният код се записва в отделен файл, обикновено със старото име, но с разширение “.obj” или “.o”.

Обектният файл съдържа само “превода” на програмата, а не и на библиотеките, които са декларирани в нея (в случая на програмата Zad1.cpp файлът Zad1.obj не съдържа обектния код на iostream.h). Авторите на пакета iostream.h са описали всички необходими действия и са записали нужния машинен код в библиотеката iostream.h.

3. Свързване

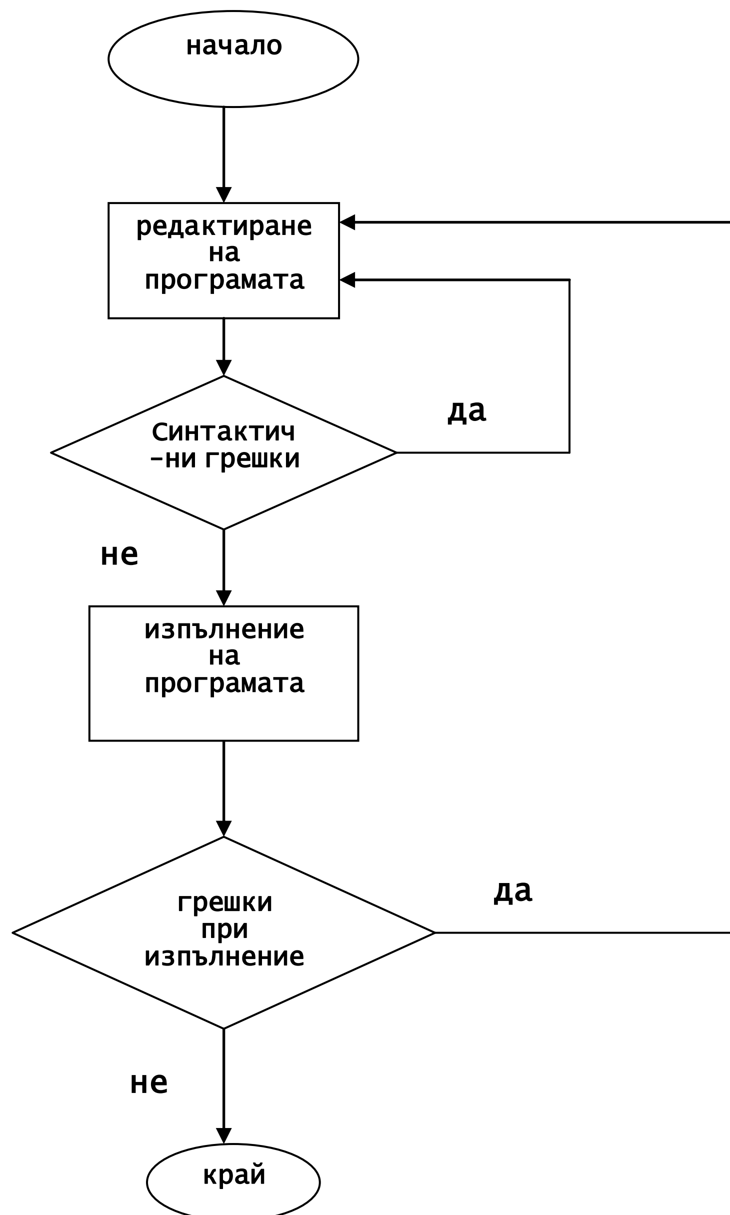
Обектният файл и необходимите части от библиотеки се свързват в т. нар. изпълним файл. Това се извършва от специална програма, наречена свързваща програма или свързващ редактор (linker). Изпълнимият файл има името на изходния файл, но разширението му обикновено е “.exe”. Той съдържа целия машинен код, необходим за изпълнението на програмата. Този файл може да се изпълни и извън средата за програмиране на езика C++.

Фиг. 4 илюстрира стъпките на изпълнение на програма на C++.



Фиг. 4

Програмистката дейност, свързана с изпълнението на програма на C++, преминава през три стъпки като реализира цикъла “редактиране-компилиране-настройка”. Започва се с редактора като се пише изходният файл. Компилира се програмата и ако има синтактични грешки, чрез редактора се поправят грешките. Когато програмата е “изчистена” от синтактичните грешки, започва изпълнението ѝ. Ако възникнат грешки по време на изпълнението, се осъществява връщане отново в редактора и се поправят предполагаемите грешки. После пак се компилира и стартира програмата. Цикълът “редактиране-компилиране-настройка” е илюстриран на Фиг. 5.



Фиг. 5

2.2. Основни означения

Всяка програма на C++ е записана като редица от знаци, които принадлежат на **азбуката на езика.**

Азбука на C++

Азбуката на езика включва:

- главните и малки букви на латинската азбука;
- цифрите;

- специалните символи

+ - * / = ( ) [ ] { } | : ; “ ‘ < > , . _

! @ # $ % ^ ~

Някой от тези знаци, по определени правила, са групирани в думи (лексеми) на езика.

Думи на езика

Думите на езика са идентификатори, запазени и стандартни думи, константи, оператори и препинателни знаци.

Идентификатори

Редица от букви, цифри и знака за подчертаване (долна черта), започваща с буква или знака за подчертаване, се нарича **идентификатор**.

Примери:

Редиците от знаци

A12 help Help double int15_12 rat_number INT1213 Int15_12

са идентификатори, а редиците

1ba ab+1 a(1) a’b

не са идентификатори. В първия случай редицата започва със цифра, а в останалите – редиците съдържат недопустими за идентификатор знаци.

Идентификаторите могат да са с произволна дължина. В съвременните компилатори максималният брой знаци на идентификаторите може да се задава, като подразбиращата се стойност е 32.

Забележка: При идентификаторите се прави разлика между малки и главни букви, така help, Help, HELP, HeLp и HElp са различни идентификатори.

Идентификаторите се използват за означаване на имена на променливи, константи, типове, функции, класове, обекти и други компоненти на програмите.

Препоръка: Не започвайте вашите идентификатори със знака за подчертаване. Такива идентификатори се използват от компилатора на C++ за вътрешно предназначение.

Допълнение: Чрез метаезика на Бекус-Наур, синтаксисът на променливите се определя по следния начин:

<променлива> ::= <идентификатор>

Някои идентификатори са резервирани в езика.

Запазени думи

Това са такива идентификатори, които се използват в програмите по стандартен, по предварително определен начин и които не могат да бъдат използвани по друг начин. Чрез тях се означават декларации, дефиниции, оператори, модификатори и други конструкции. Реализацията Visual C++ 6.0 съдържа около 70 такива думи.

В програмата Zad1.cpp са използвани запазените думи int, double, return.

Стандартни думи

Това са такива идентификатори, които се използват в програмите по стандартен, по предварително определен начин. Тези идентификатори **могат** да се използват и по други начини, например като обикновени идентификатори.

В програмата Zad1.cpp е използвана стандартната дума cout.

Например,

```
#include <iostream.h>
int main()
{int cout = 21;
 return 0;
}
```

е допустима програма на C++. В нея идентификаторът cout е използван като име на променлива. Правенето на опит за използване на cout по стандартния начин води до грешка. Така фрагментът

```
#include <iostream.h>
  int main()
  {int cout = 21;
   cout << cout << "\n";
   return 0;
   }
```

е недопустим.

Препоръка: Стандартните думи да се използват само по стандартния начин.

Константи

Данна, която не може да бъде променяна, се нарича **константа.** Има числови, знакови, низови и др. видове константи.

Целите и реалните числа са **числови константи.** Целите числа се записват както в математиката и могат да бъдат задавани в десетична, шестнадесетична или осмична бройна система. Реалните числа се записват по два начина: във формат с *фиксирана точка* (например, 2.34 -12345.098) и в *експоненциален формат* (например, 5.23e-3 и 5.23E-3 означават 5.23 умножено с 10^{-3}).

Низ, **знаков низ** или **символен низ** е крайна редица от знаци, оградени в кавички. Например, редиците: “Това е низ.”, “1+23-34”, “Hellow\n” са низове.

Забележка: Операторът

cout << “Hellow\n”;

извежда върху екрана поздрава Hellow и премества курсора на нов ред.

Оператори

В C++ има три групи оператори: аритметично - логически, управляващи и оператори за управление на динамичната памет.

- *аритметично-логически оператори*

Наричат се още **аритметично-логически операции.** Те реализират основните аритметични и логически операции като: събиране (+), изваждане (-), умножение (*), деление (/), логическо И (&&, and), логическо ИЛИ (||, or) и др. В програмата Zad1.cpp бяха използвани * и +.

- *управляващи оператори*

Това са конструкции, които управляват изчислителния процес. Такива са условния оператор, оператора за цикъл, за безусловен преход и др.

- *операторите за управление на динамичната памет*

Те позволяват по време на изпълнение на програмата да бъде заделяна и съответно освобождавана динамична памет.

Препинателни знаци

Използват се ; < > { } ( ) и др. знаци.

Разделяне на думите

В C++ разделителите на думите са интервалът, вертикалната и хоризонталната табулации и знакът за нов ред.

Коментари

Коментарите са текстове, които не се обработват от компилатора, а служат само като пояснения за програмистите. В C++ има два начина за означаване на коментари. Единият начин е, текстът да се огради с /* и */. Използвахме го вече в Zad1.cpp. Тези коментари не могат да бъдат влагани. Другият начин са коментарите, които започват с // и завършват с края на текущия ред.

Коментарите са допустими навсякъде, където е допустим разделител.

Забележка: Не се препоръчва използването на коментари от вида // в редовете на директивите на компилатора.

2.3. Вход и изход

Програма Zad1.cpp намира периметъра и лицето само на правоъгълник със страни 2.3 и 3.7. Нека решим тази задача в общия случай.

Задача 2. Да се напише програма, която въвежда размерите на правоъгълник и намира периметъра и лицето му.

Програмата Zad2.cpp решава тази задача.

Program Zad2.cpp

```
#include <iostream.h>
int main()
{// въвеждане на едната страна
  cout << "a= ";
  double a;
  cin >> a;
```

```
    // въвеждане на другата страна
    cout << "b= ";
    double b;
    cin >> b;
    // намиране на периметъра
    double p;
    p = 2*(a+b);
    // намиране на лицето
    double s;
    s = a*b;
     // извеждане на периметъра
    cout << "p=  " << p << "\n";
     // извеждане на лицето
    cout << "s=   " << s << "\n";
    return 0;
  }
```

Когато програмата бъде стартирана, върху екрана ще се появи подсещането

a=

което е покана за въвеждане размерите на едната страна на правоъгълника. Курсорът стои след знака =. Очаква се да бъде въведено число (цяло или реално), след което да бъде натиснат клавишът ENTER.

Следва покана за въвеждане на стойност за другата страна на правоъгълника, след което програмата ще изведе резултата и ще завърши изпълнението си.

Въвеждането на стойността на променливата a се осъществява с оператора за вход

```
cin >> a;
```

Обектът cin е името на стандартния входен поток, обикновено клавиатурата на компютъра. Изпълнението му води до пауза до въвеждане на число и натискане на клавиша ENTER. Нека за стойност на a е въведено 5.65, следвано от ENTER. В буфера на клавиатурата се записва

cin

5	.	6	5	\n	

След изпълнението на

```
cin >> a;
```

променливата a се свързва с 5.65, а в буфера на клавиатурата остава знакът \n, т.е.

cin

ОП

a

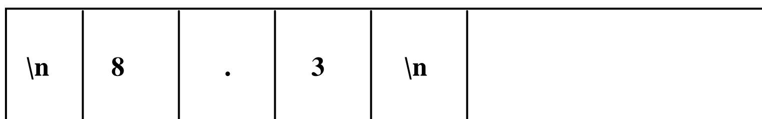
5.65

Въвеждането на стойността на променливата b се осъществява с оператора за вход

```
cin >> b;
```

Изпълнението му води до пауза до въвеждане на число и натискане на клавиша ENTER. Нека е въведено 8.3, следвано от ENTER. В буфера на клавиатурата имаме:

cin



Изпълнението на оператора

```
cin >> b;
```

прескача знака \n, свързва 8.3 с променливата b, а в буфера на клавиатурата отново остава знакът \n, т.е.

cin

ОП

a   b

5.65   8.3

Чрез оператора за вход могат да се въвеждат стойности на повече от една променлива. Фиг. 6 съдържа по-пълно негово описание.

Входът от клавиатурата е буфериран. Това означава, че всяка редица от натиснати клавиши се разглежда като пакет, който се обработва чак след като се натисне клавишът ENTER.

Оператор за вход >>

Синтаксис

cin >> <променлива>;

където

- cin е обект (променлива) от клас (тип) istream, свързан с клавиатурата,
- <променлива> е идентификатор, дефиниран, като променлива от "допустим тип", преди оператора за въвеждане. (Типовете int, long, double са допустими).

Семантика

Извлича (въвежда) от cin (клавиатурата) поредната дума и я прехвърля в аргумента-приемник <променлива>. Конструкцията

cin >> <променлива>

е израз от тип istream със стойност левия му аргумент, т.е. резултатът от изпълнението на оператора >> е cin. Това позволява няколко думи да бъдат извличани чрез верига от оператори >>.

Следователно, допустим е следният по-общ синтаксис на >>:

cin >> <променлива> { >> <променлива>};

Операторът >> се изпълнява отляво надясно. Такива оператори се наричат ляво асоциативни. Така операторът

cin >> $променлива_1$ >> $променлива_2$ >> … >> $променлива_n$;

е еквивалентен на редицата от оператори:

cin >> $променлива_1$;

cin >> $променлива_2$;

…

cin >> $променлива_n$;

Освен това, ако операцията въвеждане е завършила успешно, състоянието на cin е true, в противен случай състоянието на cin е false.

Фиг. 6.

В случая от Фиг. 6, настъпва пауза. Компилаторът очаква да бъдат въведени n *стойности* - за $променлива_1$, $променлива_{2,}$, …, $променлива_n$, съответно и бъде натиснат клавишът ENTER. Тези стойности трябва да бъдат въведени по подходящ начин (на един ред, на отделни редове

или по няколко данни на последователни редове, слепени или разделени с интервали, табулации или знака за нов ред), като *стойност*$_i$ трябва да бъде от тип, съвместим с типа на *променлива*$_i$ (i = 1, 2, …, n).

Пример: Да разгледаме програмния фрагмент:

```
double a, b, c;
cin >> a >> b >> c;
```

Операторът за вход изисква да бъдат въведени три реални числа за a, b и c съответно. Ако се въведат

```
1.1    2.2     3.3 ENTER
```

променливата a ще се свърже с 1.1, b – с 2.2 и c – със 3.3. Същият резултат ще се получи, ако се въведе

```
1.1   2.2 ENTER
3.3 ENTER
```

или

```
1.1 ENTER
2.2   3.3 ENTER
```

или

```
1.1 ENTER
2.2 ENTER
3.7 ENTER
```

или даже ако се въведе

```
1.1   2.2   3.3   4.4 ENTER
```

В последния случай, стойността 4.4 ще остане необработена в буфера на клавиатурата и ще обслужи следващо четене, ако има. Този начин на действие съвсем не е приемлив. Още по-лошо ще стане когато се въведат данни от неподходящ тип.

Пример: Да разгледаме фрагмента:

```
int a;
cin >> a;
```

Той дефинира целочислена променлива a (a е променлива от тип int), след което настъпва пауза в очакване да бъде въведено цяло число. Нека сме въвели 1.25, следвано от ENTER. Състоянието на буфера на клавиатурата е:

cin

1	**.**	**2**	**5**	**\n**	

Операторът

```
cin >> a;
```

свързва a с 1, но не прескача останалата информация от буфера и тя ще обслужи следващо четене, което води до непредсказуем резултат. Още по-неприятна е ситуацията, когато вместо цяло число за стойност на a се въведе някакъв низ, например one, следван от ENTER. В този случай, изпълнението на

```
cin >> a;
```

ще доведе до

cin

o	n	e	\n	**състояние** fail

и

ОП

a

-

т.е. стойността на променливата a не се променя (остава неопределена), а буферът на клавиатурата изпада в състояние fail. За съжаление системата не извежда съобщение за грешка, което да уведоми за възникналия проблем.

Засега препоръчваме въвеждането на коректни входни данни. Преудоляването на недостатъците, илюстрирани по-горе, ще разгледаме в следващите части на книгата.

Вече използвахме оператора за изход. Фиг. 7 описва неговите синтаксис и семантика.

2.4. Структура на програмата на C++

Когато програмата е малка, естествено е целият й код да бъде записан в един файл. Когато програмите са по-големи или когато се работи в колектив, ситуацията е по-различна. Налага се да се раздели кодът в отделни изходни (source) файлове. Причините, поради които се налага разделянето, са следните. Компилирането на файл отнема време и е глупаво да се чака компилаторът да транслира

отново и отново код, който не е бил променян. Трябва да се компилират само файловете, които са били променяни.

Оператор за изход <<

Синтаксис

cout << <израз>;

където

- cout е обект (променлива) от клас (тип) ostream и предварително свързан с екрана на компютъра;
- <израз> е израз от допустим тип. Представата за израз продължава да бъде тази от математиката. Допустими типове са bool, int, short, long, double, float и др.

Семантика

Операторът << изпраща (извежда) към (върху) cout (екрана на компютъра) стойността на <израз>. Конструкцията

cout << <израз>

е израз от тип ostream и има стойност първия му аргумент, т.е. резултатът от изпълнението на оператора << в горния случай е cout. Това позволява чрез верига от оператори << да бъдат изведени стойностите на повече от един <израз>, т.е. допустим е следният по-общ синтаксис:

cout << <израз> { << <израз>};

Операторът << се изпълнява отляво надясно (ляво асоциативен е). Така операторът

cout << израз$_1$ << израз$_2$ << … << израз$_n$

е еквивалентен на редицата от оператори:

cout << израз$_1$;

cout << израз$_2$;

…

cout << израз$_n$;

Фиг. 7.

Друга причина е работата в колектив. Би било трудно много програмисти да редактират едновременно един файл. Затова кодът на програмата се разделя така, че всеки програмист да отговаря за един или няколко файлове.

Ако програмата се състои от няколко файла, трябва да се каже на компилатора как да компилира и изгради цялата програма. Това ще направим в следващите раздели. Сега ще дадем най-обща представа за структурата на изходните файлове. Ще ги наричаме още модули.

Изходните файлове се организират по следния начин:

<изходен_файл> ::= <заглавен_блок_с_коментари>
<заглавни_файлове>
<константи>
<класове>
<глобални_променливи>
<функции>

Заглавен блок с коментари

Всеки модул започва със заглавен блок с коментари, даващи информация за целта му, използвания компилатор и операционна среда, за името на програмиста и датата на създаването му. Заглавният коментар може да съдържа забележки, свързани с описания на структури от данни, аргументи, формат на файлове, правила, уговорки.

Заглавни файлове

В тази част на модула са изброени всички необходими заглавни файлове. Например

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

Забелязваме, че за разделител е използван знакът за нов ред, а не ;.

Константи

В тази част се описват константите, необходими за модула. Вече имаме някаква минимална представа за тях. По-подробно описание на синтаксиса и семантиката им е дадена на Фиг. 8. За да бъде програмата по-лесна за четене и модифициране, е полезно да се дават символични имена не само на променливите, а и на константите. Това става чрез дефинирането на константи.

Задача 3. Да се напише програма, която въвежда радиуса на окръжност и намира и извежда дължината на окръжността и лицето на кръга с дадения радиус.

Една програма, която решава задачата е следната:

```
Program Zad3.cpp
#include <iostream.h>
const double PI = 3.1415926535898;
int main()
{ double r;
  cout << "r= ";
  cin >> r;
  double p = 2 * PI * r;
  double s = PI * r * r;
  cout << "p=" << p << "\n";
  cout << "s=" << s << "\n;
  return 0;
}
```

В тази програма е дефинирана реална константа с име PI и стойност 3.1415926535898, след което е използвано името PI.

Дефиниране на константи

Синтаксис

const <име_на_тип> <име_на_константа> = <израз>;

където

const е запазена дума (съкращение от constant);

<име_на_тип> е идентификатор, означаващ име на тип;

<име_на_константа> е идентификатор, обикновено състоящ се от главни букви, за да се различава визуално от променливите.

<израз> е израз от тип, съвместим с <име_на_тип>.

Семантика

Свързва <име_на_константа> със стойността на <израз>. Правенето на опит да бъде променяна стойността на константата предизвиква грешка.

Фиг. 8.

Примери:

```
const int MAXINT = 32767;
const double RI = 2.5 * MAXINT;
```

Предимства на декларирането на константите:

- Програмите стават по-ясни и четливи.
- Лесно (само на едно място) се променят.
- Вероятността за грешки, възможни при многократното изписване на стойността на константата, намалява.

Забележка: Тъй като в програмата Zad3.cpp е използвана само една функция (main), декларацията на константата PI може да се постави във функцията main, преди първото нейно използване.

Класове

Тази част съдържа дефинициите на класовете, използвани в модула.

В езика C++ има стандартен набор от типове данни като int, double, float, char, string и др. Този набор може да бъде разширен чрез дефинирането на класове.

Дефинирането на клас въвежда нов тип, който може да бъде интегриран в езика. Класовете са в основата на обектно-ориетираното програмиране, за което е предназначен езика C++.

Дефинирането и използването на класове ще бъде разгледано по-късно.

Глобални променливи

Езикът поддържа глобални променливи. Те са променливи, които се дефинират извън функциите и които са “видими” за всички функции, дефинирани след тях. Декларират се както се дефинират другите (локалните) променливи. Използването на много глобални променливи е лош стил за програмиране и не се препоръчва. Всяка глобална променлива трябва да е съпроводена с коментар, обясняващ предназначението ѝ.

Функции

Всеки модул задължително съдържа функция main. Възможно е да съдържа и други функции. Тогава те се изброяват в тази част на

модула. Ако функциите са подредени така, че всяка от тях е дефинирана преди да бъде извикана, тогава main трябва да бъде последна. В противен случай, в началото на тази част на модула, трябва да се **декларират** всички функции.

Задачи

Задача 1. Кои от следните редици от знаци са идентификатори, кои не и защо?

а) a	б) x1	в) x_1	г) x’	д) x1x2
е) sin	ж) sin x	з) cos(x)	и) x-1	к) 2a
л) min 1	м) Beta	н) a1+a2	о) k”m	п) sin’x

Задача 2. Намерете синтактичните грешки в следващата програма:

```
include <iostream>
int Main()
{ cout >> “a, b =  “;
  cin << a, b;
  cout << “The product of “ << a << “and” << b << “is: “
       << a*b < “\n”
  return 0;
}
```

Задача 3. Напишете програма, която разменя стойностите на две числови променливи.

Задача 4. Напишете програма, която намира минималното (максималното) от две цели числа.

Задача 5. Напишете програма, която изписва с главни букви текста Hellow World.

Упътване: Голямата буква H може да се представи така:

```
o   o
o   o
o   o
ooooo
o   o
o   o
o   o
```

и да се реализира по следния начин:

char* let_H = “o   o\no   o\no   o\nooooo\no   o\no   o\no   o\n”;

където char* означава тип низ.

Допълнителна литература

1. Г. Симов, Програмиране на С++, С., СИМ, 1993.
2. К. Хорстман, Принципи на програмирането със С++, С., СОФТЕХ, 2000.
3. П. Лукас, Наръчник на програмиста, С., Техника, 1994.

Глава 3

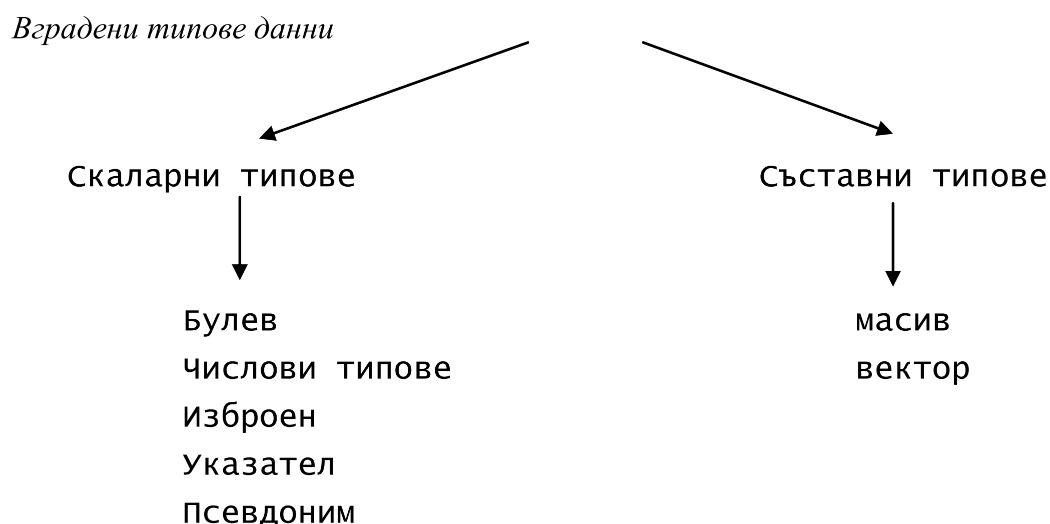
Скаларни типове данни

Езикът C++ е изключително мощен по отношение на типовете данни, които притежава. Най-общо, типовете му могат да бъдат разделени на: **вградени** и **абстрактни.**

Вградените типове са предварително дефинирани и се поддържат от неговото ядро.

Абстрактните типове се дефинират от програмиста. За целта се определят съответни класове.

Една непълна класификация на вградените типове данни е дадена на Фиг. 1.



Фиг. 1.

Скаларни са типовете данни, които се състоят от една компонента (число, знак и др.).

Съставни типове са онези типове данни, компонентите на които са редици от елементи.

Типът указател дава средства за динамично разпределение на паметта.

В тази глава ще разгледаме само някои скаларни типове данни.

Всеки тип се определя с **множество от допустими стойности** (множество от стойности) и **операции и вградени функции**, които могат да се прилагат над елементите от множеството от стойностите му.

3.1. Логически тип

Нарича се още булев тип в чест на Дж. Бул, английски логик, поставил основите на математическата логика.

Типът е стандартен, вграден в реализацията. За означаването му се използва запазената дума bool (съкращение от boolean).

Множество от стойности

Състои се от два елемента – стойностите true (истина) и false (лъжа). Тези стойности се наричат още **булеви константи**.

<булева_константа> ::= true | false.

Променлива величина, множеството от допустимите стойности, на която съвпада с множеството от стойности на типа булев, се нарича **булева** или **логическа променлива** или **променлива от тип булев**. Дефинира се по обичайния начин.

Примери:

bool b1, b2;

bool b3 = false;

Дефиницията свързва булевите променливи с множеството от стойности на типа булев или с конкретна стойност от това множество като отделя по 1 байт оперативна памет за всяка от тях. Съдържанието на тази памет е неопределено или е булевата константа, свързана с дефинираната прменлива, в случай, че тя е инициализирана.

След дефиницията от примера по-горе, имаме:

```
ОП
b1        b2        b3
-         -         false
```

1 байт 1 байт 1 байт

Съдържанието на паметта, именувана с b1 и b2 е неопределено, а това на именуваната с b3 е false. В същност, вместо false в паметта е записан кодът (вътрешното представяне) на false - 0.

Вътрешните представяния на булевите константи са:

false 0

true 1

Операции и вградени функции

Логически операции

Конюнкция (логическо умножение)

Тя е двуаргументна (бинарна) операция. Означава се с and или && (за Visual C++, 6.0) и се дефинира по следния начин:

A	B	A and B
false	false	false
false	rue	false
true	false	false
true	true	true

Операцията се поставя между двата си аргумента. Такива операции се наричат **инфиксни**.

Дизюнкция (логическо събиране)

Тя е бинарна, инфиксна операция. Означава се с or или || (за Visual C++, 6.0) и се дефинира по следния начин:

A	B	A or B
false	false	false
false	true	true
true	false	true
true	true	true

Логическо отрицание

Тя е едноаргументна (унарна) операция. Означава се с not или ! (за Visual C++, 6.0) и се дефинира по следния начин:

A	not A
false	true
true	false

Поставя се пред единствения си аргумент. Такива оператори се наричат **префиксни.**

Забележка: Може да няма разделител между оператора ! и константите true и false, т.е. записите !true и !false са допустими.

Допълнение: Смисълът на операторите and, or и not е разширен чрез разширяване смисъла на булевите константи. Прието е, че true е всяка стойност, различна от 0 и че false е стойността 0.

Операции за сравнение

Над булевите данни могат да се извършват следните инфиксни операции за сравнение:

```
==    - за равно
!=    - за различно
>     - за по-голямо
>=    - за по-голямо или равно
<     - за по-малко
<=    - за по-малко или равно
```

Сравняват се кодовете.

Примери:

```
false < true   e true
false > false  e false
true >= false  e true
```

Въвеждане

Не е възможно въвеждане на стойност на булева променлива чрез оператора >>, т.е. операторът

```
cin >> b1;
```

е недопустим, където b1 е булевата променлива, дефинирана по-горе.

Извеждане

Осъществява се чрез оператора

```
cout << <булева_константа>;
```

или по-общо

```
cout << <булев_израз>;
```

където синтактичната категория <булев_израз> е определена по-долу.

Извежда се кодът на булевата константа или кодът на булевата константа, която е стойност на <булев_израз>.

Булеви изрази

Булевите изрази са правила за получаване на булева стойност. Дефинират се *рекурсивно* по следния начин:

- Булевите константи са булеви изрази.
- Булевите променливи са булеви изрази.
- Прилагането на булевите оператори not (!), and (&&), or (||) над булеви изрази е булев израз.
- Прилагането на операциите за сравнение ==, !=, >, >=, <, <= към булеви изрази е булев израз.

Примери: Нека имаме дефиницията

```
bool b, b1, b2;
```

Следните изрази са булеви:

```
true    b    b1    b2    !false    !!b    !b1 || b2
!!!b && !!!!!b2    b < !b2  false >= b  b1 == b2 > b   b != b1
```

Тази дефиниция е непълна. Ще отбележим само, че сравнението на аритметични изрази чрез изброените по-горе операции за сравнение, е булев израз. Освен това, аритметичен израз, поставен на място, където синтаксисът изисква булев израз, изпълнява ролята на булев израз. Това е резултат от разширяването смисъла на булевите

константи true и false, чрез приемането всяка стойност, различна от 0 да се интерпретира като true и 0 да се интерпретира като false.

Засега отлагаме разглеждането на семантиката на булевите изрази. Това ще направим след разглеждане на аритметичните изрази.

3.2. Числени типове

3.2.1. Целочислени типове

Ще разгледаме целочисления тип int.

Типът е стандартен, вграден в реализацията на езика. За означаването му се използва запазената дума int.

Множество от стойности

Множеството от стойности на типа int зависи от хардуера и реализацията и не се дефинира от ANSI (American National Standarts Institute). Състои се от целите числа от някакъв интервал. За реализацията Visual C++ 6.0, това е интервалът [-2147483648, 2147483647].

Целите числа се записват като в математиката, т.е.

<цяло_число> ::= [+|-]<цяло_без_знак>
<цяло_без_знак> ::= <цифра>|
<цифра><цяло_без_знак>
<цифра> ::= 0| 1| … |9.

Обикновено знакът + се пропуска.

Допълнение: Целите числа могат да са в десетична, осмична и шестнадесетична позиционна система. Осмичните числа започват с 0 (нула), а шестнадесетичните - с 0x (нула, x).

Елементите от множеството от стойности на типа int се наричат **константи от тип int.**

Променлива величина, множеството от допустимите стойности, на която съвпада с множеството от стойности на типа int, се **нарича цяла променлива** или **променлива от тип int.**

Дефинира се по обичайния начин. Дефиницията свързва променливите от тип int с множеството от стойности на типа int или с конкретна стойност от това множество като отделя по 4 байта (1 дума)

оперативна памет за всяка от тях. Ако променливата не е била инициализирана, съдържанието на свързната с нея памет е неопределено, а в противен случай – съдържа указаната при инициализацията стойност.

Примери:

int i;

int j = 56;

След тези дефиниции, имаме:

```
ОП
i           j
-           56
4 байта   4 байта
```

Операции и вградени фунции

Аритметични операции

Унарни операции

Записват се пред или след единствения си аргумент.

+, - са префиксни операции. Потвърждават или променят знака на аргумента си.

Примери: Нека

```
int i = 12, j = -7;
```

Следните означения съдържат унарна операция + или -:

```
-i    +j    -j    +i    -567
```

Бинарни операции

Имат два аргумента. Следните аритметични операции са инфиксни:

```
+      - събиране
-      - изваждане
*      - умножение
/      - целочислено деление
%      - остатък от целочислено деление.
```

Примери:

```
15 – 1235 = -1220    13 / 5 = 2
```

15 + 1235 = 1250   13 % 5 = 3
-15 * 123 = -1845   23 % 3 = 2

Забележка: Допустимо е използването на два знака за аритметични операции, но единият трябва да е унарен. Например, допустими са 5-+4, 5+-4, имащи стойност 1, а също 5*-4, равно на –20.

Логически операции

Логическите операции конюнкция, дизюнкция и отрицание могат да се прилагат над целочислени константи. Дефинират се по същия начин, като целите числа, които са различни от 0 се интерпретират true, а 0 – като false.

Примери:

123 and 0   е false
0 or 15   е true
not 67   е false

Операции за сравнение

Над целочислени константи могат да се извършват следните инфиксни операции за сравнение:

==	- за равно	!=	- за различно
>	- за по-голямо	>=	- за по-голямо или равно
<	- за по-малко	<=	- за по-малко или равно.

Наредбата на целите числа е като в математиката.

Примери:

123 < 234   е true
-123456 > 324   е false
23451 >= 0   е true

Вградени функции

В езика C++ има набор от вградени функции. Обръщението към такива функции има следния синтаксис:

<име_на_функция>(<израз>, <израз>, …, <израз>)

и връща стойност от типа на функцията.

Тук ще разгледаме само едноаргументната целочислена функция abs.

abs(x) – намира |x|, където x е цял израз
(в частност цяла константа).

Примери:
abs(-1587) = 1587   abs(0) = 0   abs(23) = 23

За използването на тази функция е необходимо в частта на заглавните файлове да се включи директивата:

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

Библиотеката math.h съдържа богат набор от функции. В някой реализации тя има име math или cmath.

Въвеждане

Реализира се по стандартния и разгледан вече начин.

Пример: Ако

```
int i, j;
```

операторът

```
cin >> i >> j;
```

въвежда стойности на целите променливи i и j. Очаква се въвеждане от стандартния входен поток на две цели константи от тип int, разделени с интервал, знаците за хоризонтална или вертикална табулация или знака за преминаване на нов ред.

Извеждане

Реализира се чрез оператора

```
cout << <цяла_константа>;
```

или по-общо

```
cout << <цял_израз>;
```

В текущата позиция на курсора се извежда константата или стойността на целия израз. Използва се минималното количество позиции, необходими за записване на цялото число.

Пример: Нека имаме дефиницията

```
int i = 1234, j = 9876;
```

Операторът

```
cout << i << j << “\n”;
```

извежда върху екрана стойностите на i и j, но слепени
12349876

Този изход не е ясен. Налага се да се извърши **форматиране** на изхода. То се осъществява чрез подходящи **манипулатори.**

Манипулатор setw

Setw е вградена функция.

Синтаксис

```
setw(<цял_израз>)
```

Семантика

Стойността на <цял_израз> задава широчината на полето на **следващия** изход.

Пример: Операторът

```
cout << setw(10);
```

не извежда нищо. Той "манипулира" следващото извеждане като указва, че в поле с широчина 10 отдясно приравнена, ще бъде записана следващата извеждана цяла константа.

Забележка: Този манипулатор важи само за първото след него извеждане.

Пример: Нека

ОП

i	j
1234	9876

Операторът

```
cout << setw(10) << i << j << "\n";
```

извежда отново стойностите на i и j слепени, като 1234 се предшества от 6 интервала, т.е.

******12349876

където интервалът е означен със знака *.

Операторът

```
cout << setw(10) << i << setw(10) << j << "\n";
```

извежда

******1234******9876

Манипулатори dec, oct и hex

Целите числа се извеждат в десетична позиционна система. Ако се налага изходът им да е в осмична или шестнадесетична позиционна система, се използват манипулаторите oct и hex съответно. Всеки от тях е в сила, докато друг манипулатор за позиционна система не е

указан за следващ извод. Връщането към десетична позиционна система се осъществява чрез манипулатора dec.

dec – манипулатор, задаващ всички следващи изходи (докато не е указан друг манипулатор, променящ позиционната система) на цели числа да са в десетична позиционна система;

oct – манипулатор, задаващ всички следващи изходи (докато не е указан друг манипулатор, променящ позиционната система) на цели числа да са в осмиична позиционна система;

hex - манипулатор, задаващ всички следващи изходи (докато не е указан друг манипулатор, променящ позиционната система) на цели числа да са в шестнадесетична позиционна система.

Пример: Нека имаме

ОП

i	j
12	23

След изпълнението на операторите

```
cout << setw(10) << dec << i << setw(10) << j << "\n";
cout << setw(10) << oct << i << setw(10) << j << "\n";
cout << setw(10) << hex << i << setw(10) << j << "\n";
```

имаме:

```
********12********23
********14********27
*********c********17
```

Забележка: Преди използване на манипулаторите е необходимо да се включи заглавният файл iomanip.h, т.е. в частта за заглавни файлове да се запише директивата

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

Други целочислени типове

Други цели типове се получават от int като се използват модификаторите short, long, signed и unsigned. Тези модификатори доопределят някои аспекти на типа int.

За реализацията Visual C++ 6.0 са в сила:

Тип	Диапазон	Необходима памет
short int	-32768 до 32767	2 байта
unsigned short int	0 до 65535	2 байта
long int	-2147483648 до 2147483647	4 байта
unsigned long int	0 до 4294967295	4 байта
unsigned int	0 до 4294967295	4 байта

Запазената дума int при тези типове се подразбира и може да бъде пропусната. Типовете short int (или само short) и long int (или само long) са съкратен запис на signed short int и signed long int.

3.2.2. Реални типове

Ще разгледаме реалния тип double.

Типът е стандартен, вграден във всички реализации на езика.

Множество от стойности

Множеството от стойности на типа double се състои от реалните числа от $-1.74*10^{308}$ до $1.7*10^{308}$. Записват се във два формата – като числа с фиксирана и като числа с плаваща запетая (експоненциален формат).

<реално_число> ::= <цяло_число>.<цяло_число_без_знак>|
<цяло_число>E<порядък> |
<цяло_число>.<цяло_число_без_знак>E<порядък>
|

<порядък> ::= <цяло_число>

При експоненциалния формат може да се използва и малката латинска буква e.

Примери: Следните числа

123.3454   -10343.034   123E13   -1.23e-4

са коректно записани реални числа.

Смисълът на експоненциалния формат е реално число, получено след умножаване на числото пред E (e) с 10 на степен числото след E (e).

Примери: 12.5E4 е реалното число 125000.0, а –1234.025e-3 е реалното число –1.234025.

Елементите от множеството от стойности на типа double се наричат **реални константи** или по-точно **константи от реалния тип double.**

Променлива величина, множеството от допустимите стойности, на която съвпада с множеството от стойности на типа double, се нарича **реална променлива** или **променлива от тип double.**

Дефинира се по обичайния начин. Дефиницията свързва реалните променливи с множеството от стойности на типа double или с конкретна стойност от това множество, като отделя по 8 байта оперативна памет за всяка от тях. Ако променливата не е била инициализирана, съдържанието на свързаната с нея памет е неопределено, а в противен случай – съдържа указаната при инициализацията стойност.

Примери:

```
double i;
double j = 5699.876;
```

След тази дефиниция, имаме:

```
ОП
   i              j
   -           5699.876
8 байта        8 байта
```

Операции и вградени функции

Аритметични операции

Унарни операции

+, - Префиксни са. Потвърждават или променят знака на аргумента си.

Примери: Нека

```
double i = 1.2, j = -7.5;
```

Следните конструкции съдържат унарна операция + или -:

```
-i    +j    -j    +i    -56.7
```

Бинарни операции

Имат два аргумента. Следните аритметичните оператори са инфиксни:

```
+     - събиране
-     - изваждане
*     - умножение
/     - деление (поне единият аргумент е реален)
```

Примери:

```
15.3 - 12.2 = 3.1     13.0 / 5 = 2.6
15 + 12.35 =  27.35   13 / 5.0 = 2.6
-1.5 * 12.3 = -18.45
```

Логически операции

Логическите операции конюнкция, дизюнкция и отрицание могат да се прилагат над реални константи. Дефинират се по същия начин, като реалните числа, които са различни от 0.0 се интерпретират като true, а 0.0 – като false.

Примери:

```
123.6 and 0.0      е false
0.0 or 15.67       е true
not 67.7           е false
```

Операции за сравнение

Над реални данни могат да се извършват следните инфиксни операции за сравнение:

```
==    - за равно                 !=    - за различно
>     - за по-голямо             >=    - за по-голямо или равно
<     - за по-малко              <=    - за по-малко или равно.
```

Наредбата на реалните числа е като в математиката.

Примери:

```
123.56 < 234.09              е true
-123456.9888 > 324.0098      е false
23451.6 >= 0                 е true
```

Допълнение: Сравнението за равно на две реални числа x и y се реализира обикновено чрез релацията: $|x - y| < \varepsilon$, където $\varepsilon = 10^{-14}$ за тип double. По-добър начин е да се използва релацията:

$$\frac{|x - y|}{\max\{|x|, |y|\}} \leq \varepsilon$$

Вградени функции

При цял или реален аргумент, следните функции връщат реален резултат от тип double:

sin(x) - синус, sin x, x е в радиани

cos(x) - косинус, cos x, x е в радиани

tan(x) - тангенс, tg x, x е в радиани

asin(x) - аркуссинус, arcsin $x \in [-\pi/2, \pi/2]$, $x \in [-1, 1]$

acos(x) - аркускосинус, arccos $x \in [0, \pi]$, $x \in [-1, 1]$

atan(x) - аркустангенс, arctg $x \in (-\pi/2, \pi/2)$

exp(x) - експонента, e^x

log(x) - натурален логаритъм, ln x, $x > 0$

log10(x) - десетичен логаритъм, lg x, $x > 0$

sinh(x) - хиперболичен синус, sh x

cosh(x) - хиперболичен косинус, ch x

tanh(x) - хиперболичен тангенс, th x

ceil(x) - ([x]+1), преобразувано в тип double

floor(x) - [x], преобразувано в тип double

fabs(x) - абсолютна стойност на x, |x|

sqrt(x) - корен квадратен от x, $x \geq 0$

pow(x, n) - степенуване, x^n (x и n са реални от тип double).

Примери: ceil(12.345) = 13.0 ceil(-12.345) = -12.0

ceil(1234) = 1234.0 ceil(-1234) = -1234.0

floor(12.345) = 12.0 floor(-12.345) = -13.0

floor(123) = 123.0 floor(-123) = -123.0

fabs(123) = 123.0 fabs(-1234) = 1234.0

sin(PI/6) намира sin(30°), където PI = 3.141592

Тези функции се намират в библиотеката math.h и за да бъдат използвани е необходимо в частта на заглавните файлове да бъде включена директивата:

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

Задача 4. Да се напише програма, която намира функцията скобка от дадено реално число.

Следната програма решава задачата:

```
#include <iostream.h>
#include <math.h>
int main()
{cout << “x= “;
 double x;
 cin >> x;
 int y;
 y = floor(x);
 cout << y << “\n”;
 return 0;
}
```

Компилаторът издава предупреждение, че на линия 8 се извършва **преобразуване** на типа double в тип int и това може да доведе до загубване на информация.

Въвеждане

Реализира се по стандартния начин.

Пример: Ако

```
double x, y;
```

операторът

```
cin >> x >> y;
```

въвежда стойности на x и y.

За разделител се използват: интервалът, знаците за вертикална и хоризонтална табулация и знакът за преминаване на нов ред. Ако за някоя реална променлива е въведена цяла константа, извършва се конвертиране в реален тип, след което полученото реално число се записва в отделената за променливата памет.

Извеждане

Реализира се чрез оператора

```
cout << <реална_константа>;
```

или по-общо

```
cout << <реален_израз>;
```

В текущата позиция на курсора се извежда реалната константа или стойността на реалния израз. Използва се минималното количество позиции, необходими за записване на реалното число.

Пример: Нека имаме дефиницията

```
double x = 12.34, y = 9.876;
```

Операторът

```
cout << x << y << "\n";
```

извежда върху екрана стойностите на x и y слепени

12.349.876

Този изход не е ясен. Налага се да се извърши форматиране на изхода. То се осъществява чрез подходящи манипулатори.

Манипулатор setw

Setw е функция.

Синтаксис

setw(<цял_израз>)

Семантика

Задава широчината на **следващото** изходно поле.

Пример: Операторът

```
cout << setw(12);
```

не извежда нищо. Той "манипулира" следващото извеждане като указва, че в поле с широчина 12 отдясно приравнена, ще бъде записана следващата извеждана реална константа.

Този манипулатор важи само за първото след него извеждане.

Пример: Нека

ОП

x y

1.56 -2.36

Операторът

```
cout << setw(10) << x << y << "\n";
```

извежда отново стойностите на x и y слепени, като 1.56 се предшества от 6 интервала, т.е.

******1.56-2.36

където интервалът е означен със знака *.

Операторът

```
cout << setw(10) << x << setw(10) << y << "\n";
```

извежда

******1.56*****-2.36

При реалните данни се налага използването и на манипулатор за задаване на броя на цифрите след десетичната точка.

Манипулатор setprecision

Setprecision е функция.

Синтаксис

```
setprecision(<цял_израз>)
```

Семантика

Стойността на аргумента на тази функция задава броя на цифрите, използвани при извеждане на следващите реални числа, а в съчетание със загадъчното обръщение setiosflags(ios::fixed), задава броя на цифрите след десетичната точка на извежданите реални числа.

Пример 1: Операторът

```
cout << setprecision(3);
```

не извежда нищо. Той указва, че следващите реални константи ще се извеждат с 3 значещи цифри.

Нека в ОП имаме:

ОП

X

21.5632

Операторът

```
cout << setprecision(3) << setw(10) << x << “\n”;
```

извежда

******21.6

като извършва закръгляване.

А оператора

```
cout << setprecision(2) << setw(10) << x << “\n”;
```

извежда

********22

Пример 2: Операторът

```
cout << setiosflags(ios::fixed) << setprecision(3);
```

не извежда нищо. Той указва, че следващите реални константи ще се извеждат с точно 3 цифри след десетичната точка.

Нека в ОП имаме:

ОП

```
X
21.5632
```

Операторът

```
cout << setiosflags(ios::fixed)
     << setprecision(3)
     << setw(10) << x << "\n";
```

извежда

```
****21.563
```

А операторът

```
cout << setiosflags(ios::fixed)
     << setprecision(1)
     << setw(10) << x << "\n";
```

извежда

```
******21.6
```

като извършва закръгляване.

Забележка: За щастие, манипулаторите setprecision() и setiosflags () са в сила не само за първата извеждана константа, но и за всички следващи, докато с нов setprecition() не се промени точността и с
resetiosflags(ios::fixed) не се отмени setiosflags(ios::fixed).

Пример: Изпълнението на програмата

```
#include <iostream.h>
#include <iomanip.h>
int main()
{double a = 209.5, b = 63.75658;
 cout  << setprecision(3)
           << setiosflags(ios::fixed);
 cout << setw(10) << a << "\n";
 cout << setw(10) << b << "\n";
 cout << resetiosflags(ios::fixed);
 cout << setw(20) << a << "\n";
 cout << setw(20) << b << "\n";
 return 0;
}
```

извежда

```
***209.500
****63.757
****************210
***************63.8
```

Допълнение: Точността на типа double е около 15 значещи цифри. За да се убедите в това, изпълнете следната програма:

```
#include <iostream.h>
int main()
{ double a = 5e14;
  double b = a – 0.1;
  double c = a – b;
  cout << c << "\n";
  return 0;
}
```

Лесно се вижда, че стойността на променливата c трябва да бъде 0.1, а програмата намира за такава 0.125. Причината е в броя на значещите цифри на b.

Други реални типове

В езика C++ има и друг реален тип, наречен float. Различава се от типа double по множеството от стойностите си и заеманата памет.

Множеството от стойности на типа float се състои от реалните числа от диапазона от $-3.4*10^{38}$ до $3.4*10^{38}$. За записване на константите от този диапазон са необходими 4 байта ОП.

Броят на значещите цифри при този тип е около 7. За да се убедите в това, изпълнете следната програма:

```
#include <iostream.h>
int main()
{ float a = 5e6;
  float b = a – 0.1;
  float c = a – b;
  cout << c << "\n";
  return 0;
```

}

Лесно се вижда, че стойността на c трябва да е 0.1, а след изпълнение на горната програма се получава 0. Компилаторът предупреждава, че на линия 4 става преобразуване от тип float в тип double.

Реална константа от диапазона на тип float, но с повече от 7 значещи цифри се приема от компилатора за реално число от тип double и присвояването му на променлива от тип float издава предупреждението, че се извършва преобразуване от тип double в тип float, което може да доведе до загуба на точност.

Точността е причината заради, която препоръчваме използването на типа double.

3.2.3. Аритметични изрази

Аритметичните изрази са правила за получаване на числови константи. Има два вида аритметични изрази: цели и реални.

<аритметичен_израз> ::= <цял_израз> | <реален_израз>

Цели аритметични изрази

Целите аритметични изрази са правила за получаване на константи от тип int или разновидностите му. Дефинират се рекурсивно по следния начин:

- Целите константи са цели аритметични изрази.

Примери: 123 –2345 –32767
0 22233345 –87

са цели аритметични изрази.

- Целите променливи са цели аритметични изрази.

Примери: Ако имаме дефиницията:

```
int i, j;
short p, q, r;
i   j
```

са цели аритметични изрази от тип int, а

p q r

са цели аритметични изрази от тип short.

- Прилагането на унарните операции + и – към цели аритметични изрази е цял аритметичен израз.

Примери: -i +j -j

са цели аритметични изрази от тип int, а

+p -p +r -q

са цели аритметични изрази от тип short.

- Прилагането на бинарните аритметични операции +, -, *, / и % към цели аритметични изрази, е цял аритметичен израз.

Пример: i % 10 + j * i - p -i + j / 5

са цели аритметични изрази от тип int,

r – p / 12 – q r % q – p r + p – q

са цели аритметични изрази от тип short.

- Цели функции, приложени над цели аритметични изрази, са цели аритметични изрази.

Примери: abs(i+j) е цял аритметичен израз от тил int, а abs(p-r) е цял аритметичен израз от тип short.

Реални аритметични изрази

Реалните аритметични изрази са правила за получаване на константа от тип double или float. Дефинират се рекурсивно по следния начин:

- Реалните константи са реални аритметични изрази.

Примери: 1.23e-3 –2345e2 –3.2767 0.0

222.33345 –8.7009

са реални аритметични изрази.

Забележка: Реална константа от диапазона на тип float, но с повече от 7 значещи цифри се приема от компилатора за реално число от тип double.

- Реалните променливи са реални аритметични изрази.

Примери: Ако имаме дефинициите:

double i, j;

float p, q, r;

i j

са реални аритметични изрази от тип double, а

p q r

са реални аритметични изрази от тип float.

- Прилагането на унарните операции + и – към реални аритметични изрази е реален аритметичен израз.

Примери: -i +j -j

са реални аритметични изрази от тип double, а

+p -p +r -q

са реални аритметични изрази от тип float.

- Прилагането на бинарните аритметични операции +, -, * и / към аритметични изрази, поне един от които е реален, е реален аритметичен израз.

Пример: i % 10 + j*i - p -i + j/5

са реални аритметични изрази от тип double, а

–p/12 – q r%q – p r + p – q

са реални аритметични изрази от тип float.

- Реални функции, приложени над реални или цели аритметични изрази, са реални аритметични изрази.

Примери: fabs(i+j) sin(i-p) cos(p/r-q) floor(p)

ceil(r-p+i) exp(p) log(r-p*q)

са реални аритметични изрази от тип double.

Семантика на аритметичните изрази

За пресмятане на стойностите на аритметичните изрази се използват следният приоритет на операциите и вградените функции:

1. Вградени функции

2. Действията в скобите

3. Операции в следния приоритет

- +, - (унарни) най-висок
- *, /, %
- +, - (бинарни)
- <<, >> най-нисък

Забележка 1: Операторите >> и << са за побитови измествания надясно и наляво съответно. Те са предефинирани за входно/изходни операции. В случая имаме предвид тази тяхна употреба.

Забележка 2: Инфиксните операции, които са разположени на един и същ ред са с еднакъв приоритет. Тези оператори се изпълняват отляво надясно. Унарните операции се изпълняват отдясно наляво.

Пример: Нека имаме дефиницията

```
double x = 23.56, y = -123.5;
```

Изпълнението на оператора

```
cout << sin(x) + ceil(y) * x – cos(y);
```

ще се извърши по следния начин: отначало ще се пресметнат стойностите на sin(x), ceil(y) и cos(y), след това ще изпълни операцията * над стойността на ceil(y) и x, полученото ще събере със стойността на sin(x), след което от полученото реално число ще се извади пресметнатата вече стойност на cos(y). Накрая върху екрана ще се изведе получената реална константа.

Аритметичните изрази могат да съдържат операнди от различни типове. За да се пресметне стойността на такъв израз, автоматично се извършва преобразуване на типовете на операндите му. Без загуба на точността се осъществяват следните преобразувания:

Тип	Преобразува се до тип
bool	всички числови типове
short	int
unsigned short	unsigned int
float	double

т.е.конвертира се от “по-малък” тип (в байтове) към “по-голям” тип.

За да се пресметне стойността на аритметичен израз с операнди от различни типове, последователно се прилагат правилата по-долу, докато се уеднаквят типовете (ако е възможно).

Ако има операнд от тип:	Другите операнди се преобразуват до:

double	double
float	float
unsigned int	unsigned int
int	int
unsigned short	unsigned short
short	short

Семантика на булевите изрази

Булевите изрази са правила за получаване на булева стойност. За пресмятане на стойностите им се използва следният приоритет на операциите и вградените функции:

1. Вградени функции
2. Действията в скобите
3. Операции в следния приоритет

- !, not, +, - (унарни) най-висок
- *, /, %
- +, - (бинарни)
- >> << (вход/изход)
- <, <=, >, >=
- ==, !=
- &&
- || най-нисък

Забележка: Инфиксните операции, които са разположени на един и същ ред са с еднакъв приоритет. Тези оператори се изпълняват отляво надясно, т.е. лявоасоциативни са. Унарните оператори се изпълняват отдясно наляво, т.е. дясноасоциативни са.

Примери: а) Нека имаме дефинициите:

```
double x = 23.56, y = -123.5;
bool b1, b2, b3;
b1 = true;
b2 = !b1;
b3 = b1||b2;
```

Изпълнението на оператора

```
cout << sin(x) + ceil(y) * x > 12398;
```

ще сигнализира грешка – некоректни аргументи на <<. Това е така, заради нарушения приоритет. Операторът << е с по-висок приоритет от този на операторите за сравнение. Налага се вторият аргумент на << да бъде ограден в скоби, т.е.операторът

```
cout << (sin(x) + ceil(y) * x > 12398);
```

вече работи добре.

б) Изпълнението на оператора

```
cout << b1 && b2 || b3 << "\n";
```

също съобщава грешка – некоректни аргументи на <<. Отново е нарушен приоритетът на операциите. Налага се аргументът b1 && b2 || b3 на << да се огради в скоби, т.е.

```
cout << (b1 && b2 || b3) << "\n";
```

вече работи добре.

Задачи върху типовете булев, цял и реален

Задача 5. Кои от следните редици от знаци са числа в C++?

а) 061 б) –31 в) 1/5 г) +910.009

д) VII е) 0.(3) ж) sin(0) з) 134+12

Решение: а), б), г).

Задача 6. Да се запишат на C++ следните числа:

а) 6! б) LXXIV в) –0,4(6) г) 138,2(38)

д) 11/4 е) π ж) $1,2 .10^{-1}$ з) $–23,(1) .10^{2}$

В дробната част да се укажат до 4 цифри.

Решение:

а) 120 б) 74 в) –0.4667 г) 138.2384

д) 2.7500 е) 3.1416 ж) 0.1200 з) –2311.1111

Задача 7. Да се запишат на езика C++ следните математически формули:

а) $a + b.c – a^2b^3c^4$

б) $\frac{a.b}{c} + \frac{c}{a.b}$

в) $(1 + \frac{x}{1!} + \frac{x^2}{2!}).(1 + \frac{x^3}{3!} + \frac{x^5}{5!})$

Решение:

г) $\sqrt{1 + \sqrt{2 + \sqrt{3 + \sqrt{4}}}}$

а) a + b * c – a * a * b * b * b * c * c * c * c

б) (a * b) / c + c / (a * b)

в) (1 + x + x*x/2)*(1 + x*x*x/6 + x*x*x*x*x/120)

или

(1 + x + pow(x,2))/(1 + pow(x, 3)/6 + pow(x, 5)/120)

г) sqrt(1 + sqrt(2 + sqrt(3 + sqrt(4))))

Задача 8. Какво ще бъде изведено след изпълнението на следната програма:

```
#include <iostream.h>
#include <math.h>
int main()
{ cout << "x=";
  double x;
  cin >>x;
  bool b;
  b = x < ceil(x);
  cout << "x=";
  cin >> x;
  b = b && (x < floor(x));
  cout << "b= " << b << "\n";
  return 0;
 }
```

ако като вход бъдат зададени числата

а) 2.7 и 0.8 б) 2.7 и –0.8 в) –2.7 и –0.8.

Решение: а) ТЪй като булевият израз 2.7 < ceil(2.7) има стойност true, а true && (0.8 < floor(0.8)) - е false, ще бъде изведено 0 (false).

Задача 9. Какъв ще е резултатът от изпълнението на програмата

```
#include <iostream.h>
int main()
{int a, b;
 cin >> a >> b >> a >> b >> a;
```

```
 cout << a << " " << b << " "
          << a << " " << b << " "
          << a << "\n";
 return 0;
}
```

ако като вход са зададени числата 1, 2, 3, 4 и 5?

Решение: След обработката на дефиницията int a, b; за променливите a и b са отделени по 4 байта ОП, т.е.

ОП

a	b
-	-

а след изпълнението на оператора за четене >>, a и b получават отначало стойностите 1 и 2. След това стойностите им се променят на 3 и 4 съответно. Най-накрая a става 5, т,е,

a	b
5	4

Тогава програмата извежда

5 4 5 4 5

Задача 10. Да се запише булев израз, който има стойност true, ако посоченото условие е в сила, и стойност false, ако условието не е в сила.

а) цялото число а се дели на 5;

б) точката x принадлежи на отсечката [2, 6];

в) точката x не принадлежи на отсечката [2, 6];

г) точката x принадлежи на отсечката [2, 6] или на отсечката [-4, -2];

д) поне едно от числата a, b и c е отрицателно;

е) числата a, b и c са равни помежду си.

Решение:

а) a % 5 == 0

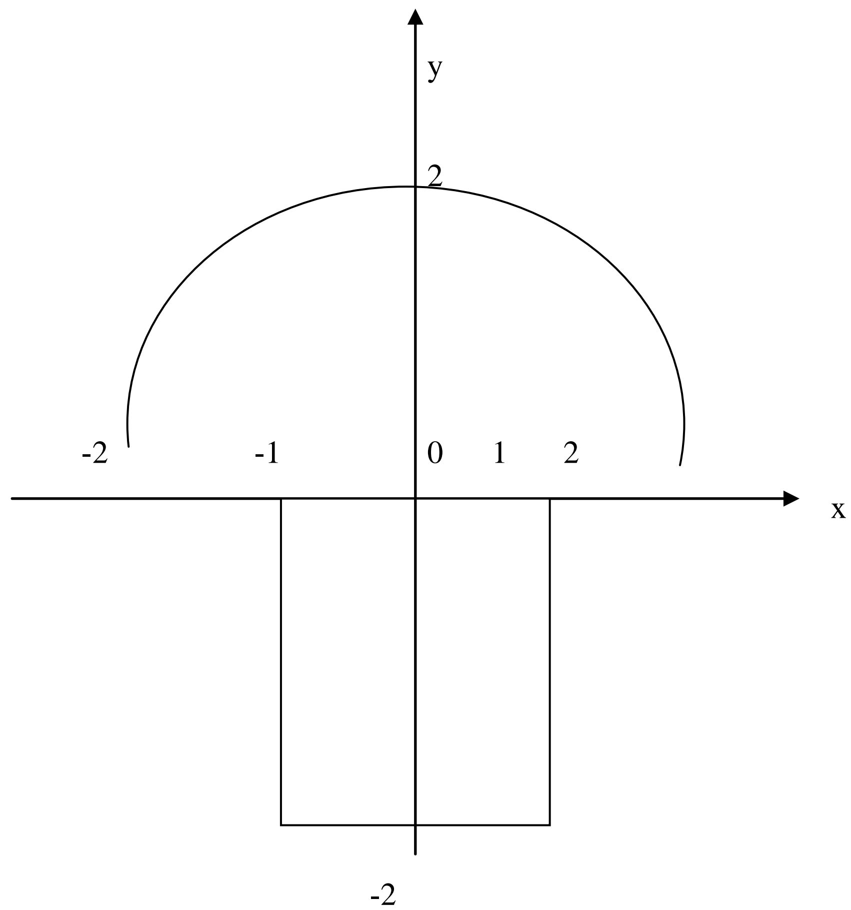
б) x >= 2 && x <= 6

в) x < 2 || x > 6 или !(x >= 2 && x <= 6)

г) x >= 2 && x <= 6 || x >= -4 && x <= -2

е) a == b && a == c

Задача 11. Да се напише програма, която въвежда координатите на точка от равнината и извежда 1, ако точката принадлежи на фигурата по-долу и – 0, в противен случай.



Решение:

```
#include <iostream.h>
#include <math.h>
int main()
{cout << "x= ";
 double x;
 cin >> x;
 cout << "y= ";
 double y;
 cin >> y;
 bool b1 = x*x + y*y <= 4 && y >= 0;
 bool b2 = fabs(x) <= 1 && y < 0 && y >= -2;
 cout << (b1 || b2) << "\n";
 return 0;
}
```

Задачи

Задача 1. Да се запишат на езика C++ следните математически формули:

а) $\dfrac{x^2 + y^2}{(|\sin x| + \lg|y|)^{\frac{1}{5}}}$

б) $\dfrac{a - b.c}{(\text{arctg } a - \text{sh } b + \text{ch } c)^4}$

в) $(\text{acos } x - [x] - 1)^3 * e^{x+1,23}$

г) $\ln(x^4 + e^x + 10)$

д) $\log_5(x^2 + x^4 + 8)$

Задача 2. Кои от следните редици от символи са правилно записани изрази на езика C++:

а) 1 + |y|

б) -abs(x) + sin z

в) abs(x) + cos(abs(y - 1.7))

г) 1 + sqrt(sin((u+v)/10))

д) -6 + xy

е) 1/-2 + Beta.

Задача 3. Да се запишат в традиционна (математическа) форма следните изрази, записани в синтаксиса на езика C++:

а) sqrt(a+b) - sqrt(a-b)

б) a + b/(c+d)-(a+b)/c+d

в) x*y/(u+v)-(u-v)/y*(a+b)

г) 1+exp(cos((x+y)/2)).

Задача 4. Да се пресметне стойността на израза:

а) cos(0) + abs(1/(1/3-1))

б) abs(a-10) + sin(a-1), за a = 1;

в) cos(-2+2*x) +sqrt(fabs(x-5)), за x = 1;

г) sin(sin(x*x-1)*sin(x*x-1)) + cos(x*x*x-1)*abs(x-2), за x = 1;

д) sin(sin(x*x-1))+cos(x*x*x-1)*cos(abs(x-2)-1)/y*a + sqrt(abs(y)-x), за x = 1, y = -2, a = 2.

Задача 5. В аритметичния израз

а) a/b*c/d*e/f*h

б) a+b/x-2*y

в) a+b/x-2*y

да се поставят скоби така, че полученият израз да съответствува на математическата формула:

а) $\dfrac{a}{b\,.\,\dfrac{c}{d\,.\,\dfrac{e}{f\,.\,h}}}$ б) $\dfrac{a+b}{x-2.y}$ в) $a+\dfrac{b}{x-2}\,.\,y$

Задача 6. Да се напише израз на езика C++, който да изразява:

а) периметъра на квадрат с лице, равно на a;

б) лицето на равностранен триъгълник с периметър, равен на p.

Задача 7. Да се напише програма, която пресмята стойността на v1, където

а) $V1 = m + \dfrac{n}{p + \dfrac{q}{r + \dfrac{s}{t}}}$ б) $V1 = \dfrac{\sin(\sin(\sin x)) + \cos(\cos(\cos x))}{|\ln x| + |\cos x| + e^x}$

Задача 8. Да се пресметне стойността на израза:

а) pow(x, 2) + pow(y, 2) <= 4	при x = 0.6, y = -1.2
б) p % 7 == p / 5 - 2	при p = 15
в) floor(10*k+16.3)/2 == 0	при k = 0.185
г) !((k+325)%2 == 1)	при k = 28
д) u*v != 0 && v > u	при u = 2, v = 1
е) x || !y	при x = false, y = true.

Задача 9. Да се запише булев израз, който да има стойност истина, ако посоченото условие е вярно и стойност - лъжа, ако условието не е вярно:

а) цялото число p се дели на 4 или на 7;

б) уравнението $a.x^2 + b.x + c = 0$ $(a \neq 0)$ няма реални корени;

в) точка с координати (a, b) лежи във вътрешността на кръг с радиус 5 и център (0, 1).

г) точка с координати (a, b) лежи извън кръга с център (c, d) и радиус f;

д) точка принадлежи на частта от кръга с център (0, 0) и радиус 5 в трети квадрант;

е) точка принадлежи на венеца с център (0, 0) и радиуси 5 и 10;

ж) x принадлежи на отсечката [0, 1];

з) x = max{a, b, c}

и) x != max{a, b, c} (операцията ! да не се използва);

к) поне една от булевите променливи x и y има стойност true;

л) и двете булеви променливи x и y имат стойност true;

м) нито едно от числата a, b и c е положително;

н) цифрата 7 влиза в записа на положителното трицифрено число p;

о) цифрите на трицифреното число m са различни;

п) поне две от цифрите на трицифреното число m са равни помежду си.

Допълнителна литература

4. К. Хорстман, Принципи на програмирането със C++, С., СОФТЕХ, 2000.
5. П. Лукас, Наръчник на програмиста, С., Техника, 1994.
6. Ст. Липман, Езикът C++ в примери, “КОЛХИДА ТРЕЙД” КООП, С. 1993.

Глава 4

Основни структури за управление на изчислителния процес

В тази глава ще разгледаме управляващите оператори в езика. Ще ги наричаме само оператори.

4.1. Оператор за присвояване

Това е един от най-важните оператори на езика. Вече многократно го използвахме, а също в глава 2 описахме неговите синтаксис и семантика. В тази глава ще го разгледаме по-подробно. Ще напомним неговите синтаксис и семантика.

Синтаксис

<променлива> = <израз>;

- където <променлива> е идентификатор, дефиниран вече като променлива,
- <израз> е израз от тип, съвместим с типа на <променлива>.

Семантика

Намира се стойността на <израз>. Ако тя е от тип, различен от типа на <променлива>, конвертира се ако е възможно до него и се записва в именуваната с <променлива> памет.

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

Ще отбележим, че в рамките на една функция не са възможни две дефиниции на една и съща променлива, но на една и съща променлива може да й бъдат присвоявани многократно различни стойности.

Пример: Не са допустими

```
…
double a = 1.5;
…
double a = a + 5.1;
…
```

но са допустими присвояванията:

```
…
double a = 1.5;
…
a = a + 34.5;
…
a = 0.5 + sin(a);
…
```

В езика C++ са въведени някои съкратени форми на оператора за присвояване. Например, заради честото използване на оператора:

```
a = a + 1;
```

той съкратено се означава с

```
a++;
```

Въведено е също и съкращението a-- на оператора a = a-1;

В същност ++ и -- са реализирани като постфиксни унарни оператори увеличаващи съответно намаляващи аргумента си с 1. Приоритетът им е един и същ с този на унарните оператори +, - и !.

Забележка: От оператора ++, за добавяне на 1, идва името на езика C++ - вариант на езика C, към който са добавени много подобрения и нови черти.

Допълнение: Операторът за присвояване = е претоварен и с функцията на дясноасоциативна инфиксна аритметично-логическа операция с приоритет по-нисък от този на дизюнкцията ||. Това позволява на оператора за присвояване

```
X = y;
```

където x е променлива, a y – израз, да се гледа като на израз от тип – типа на x и стойност – стойността на y, ако е от типа на x или стойността на y, но преобразувана до типа на x.

Пример: Програмата

```
#include <iostream.h>
int main()
{int a;
```

```
 double  b;
 b = 3.2342;
 cout << (a = b) << "\n";
 return 0;
}
```

е допустима. Резултът от изпълнението й е 3, като компилаторът издава предупреждение за загуба на информация при преобразуването от тип double в тип int. Изразът a = b е от тип int и има стойност 3. Ограждането му в скоби е необходимо заради по-ниския приоритет на = от този на <<.

Допълнение: Допустим е операторът:

```
X = y = 5;
```

Тъй като = е дясноасоциативен, отначало променливата y се свързва със 5, което е стойността на израза y = 5. След това x се свързва с 5, което е стойността на целия израз.

Някои компилатори издават предупреждение при тази употреба на оператора =. Затова не препоръчваме да се използва = като аритметичен оператор.

Задачи върху оператора за присвояване

Задача 12. Нека са дадени дефинициите

```
double x, y, z;
int m, n, p;
```

Кои от следните редици от символи са оператори за присвояване:

а) -x = y;	б) x = -y;	в) m + n = p;
г) p = x + y;	д) z = x - y	е) z = m + n;
ж) sin(0) = 0;	з) x n + sin(z)	к) 4 = sin(p + 5)?

В случаите а), в), ж) и к) редиците не са оператори за присвояване, тъй като на израз се присвоява израз. В случай г) редицата от символи е оператор за присвояване, но тъй като на цяла променлива се присвоява стойността на реален израз, компилаторът ще направи предупреждение за загуба на точност, а в случай з) е пропуснат символът '=' от знака за присвояване.

Задача 13. Да се напише програма, която въвежда стойности на реалните променливи a и b, след което разменя и извежда стойностите

им (Например, ако a = 5.6, а b = -3.4, след изпълнението на програмата a да става -3.4 , а b да получава стойността 5.6).

Програма Zad13.cpp решава задачата.

```
Program Zad13.cpp
#include <iostream.h>
int main()
{cout << "a= ";
 double a;
 cin >> a;
 cout << "b= ";
 double b;
 cin >> b;
 double x;
 x = a;
 a = b;
 b = x;
 cout << "a= " << a << "\n";
 cout << "b =" << b << "\n";
 return 0;
}
```

В тази програма се използва работна променлива x, която съхранява първоначалната стойност на променливата a.

Задача 14*.* Да се напише програма, която въвежда положително трицифрено число и извежда на отделни редове цифрите на стотиците, на десетиците и на единиците на числото.

Програма Zad14.cpp решава задачата.

```
Program Zad14.cpp
#include <iostream.h>
#include <iomanip.h>
int main()
{ cout << "a - three-digit, integer and positive? ";
  int a ;
  cin >> a;
  short s, d, e;
  s = a / 100;
  d = a / 10 % 10;
```

```
  e = a % 10;
  cout << setw(10) << "stotici: " << setw(5) << s << "\n";
  cout << setw(10) << "desetici:" << setw(5) << d << "\n";
  cout << setw(10) << "edinici: " << setw(5) << e << "\n";
  return 0;
 }
```

Задача 15. На цялата променлива b да се присвои първата цифра на дробната част на положителното реално число x (Например, ако x = 52.467, то b = 4).

Програма Zad15.cpp решава задачата

```
Program Zad15.cpp
#include<iostream.h>
#include <math.h>
int main()
{ cout << "x>0? ";
  double x;
  cin >> x;
  int i = floor(x * 10);
  int b = i % 10;
  cout << x << "\n";
  cout << b << "\n";
  return 0;
}
```

Задача 16. Да се напише програма, която извежда 1, ако в записа на положителното четирицифрено число a, всички цифри са различни и 0 - в противен случай.

Програма Zad16.cpp решава задачата.

```
Program Zad20.cpp
#include <iostream.h>
int main()
{ cout << "a - four-digit, integer and positive? ";
  int a ;
  cin >> a;
  short h, s, d, e;
  h = a / 1000;
```

```
  s = a / 100 % 10;
  d = a / 10 % 10;
  e = a % 10;
  cout << (h != s && h != d && h != e &&
          s != d && s != e && d != e) << "\n";
  return 0;
}
```

4.2. Празен оператор

Това е най-простия оператор на C++. Описанието му е дадено на Фиг. 1.

Синтаксис

;

Операторът не съдържа никакви символи. Завършва със знака ;.

Семантика

Не извършва никакви действия. Използва се когато синтаксисът на някакъв оператор изисква присъствието на поне един оператор, а логиката на програмата не изисква такъв.

Фиг. 1.

Забележка: Излишни празни оператори не предизвикват грешка при компилация. Например, редицата от оператори

```
a = 150;;;
b = 50;;;;
c = a + b;;
```

се състои от: оператора за присвояване a = 150, 2 празни оператора, оператора за присвояване b = 50, 3 празни оператора, оператора за присвояване c = a + b и 1 празен оператор и е напълно допустим програмен фрагмент.

Други примери ще дадем по-късно.

4.3. Блок

Често синтаксисът на някакъв оператор на езика изисква използването на един оператор, а логиката на задачата – редица от оператори. В този случай се налага оформянето на блок (Фиг. 2.).

Синтаксис

{ <оператор$_1$>

<оператор$_2$>

. . .

<оператор$_n$>

}

Семантика

Обединява няколко оператора в един, наречен блок. Може да бъде поставен навсякъде, където по синтаксис стои оператор.

Дефинициите в блока, се отнасят само за него, т.е. не могат да се използват извън него.

Фиг. 2.

Пример: Операторът

```
{cout << “a= “;
 double a;
 cin >> a;
 cout << “b= “;
 double b;
 cin >> b;
 double c = (a+b)/2;
 cout << “average{a, b} = “ << c << “\n”;
}
```

е блок. Опитът за използване на променливите a, b и c след блока, предизвиква грешка.

Препоръка: Двойката фигурни скобки, отварящи и затварящи блока да се поставят една под друга.

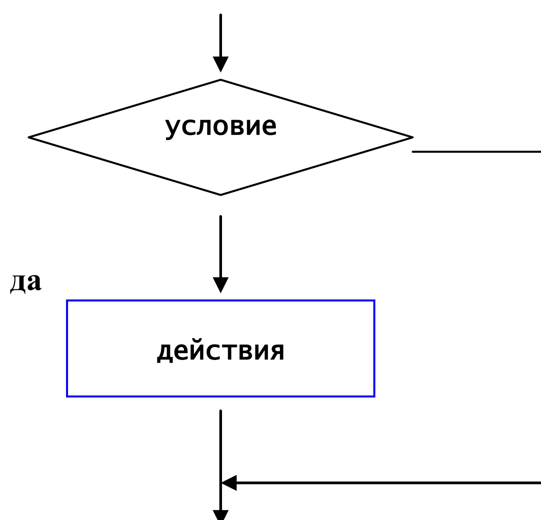
Забележка: За разлика от другите оператори, блокът не завършва със знака ;.

4.4. Условни оператори

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

4.4.1. Условен оператор if

Чрез този условен оператор се реализира разклоняващ се изчислителен процес от вид, илюстриран на Фиг. 3.



Фиг. 3.

Ако указаното условие е в сила, изпълняват се определени действия, а ако не – тези действия се прескачат. И в двата случая след това се изпълняват общи действия.

Условието се задава чрез някакъв булев израз, а действията – чрез оператор. Фиг. 4 описва подробно синтаксиса и семантиката на този оператор.

Синтаксис

if (<условие>) <оператор>

където

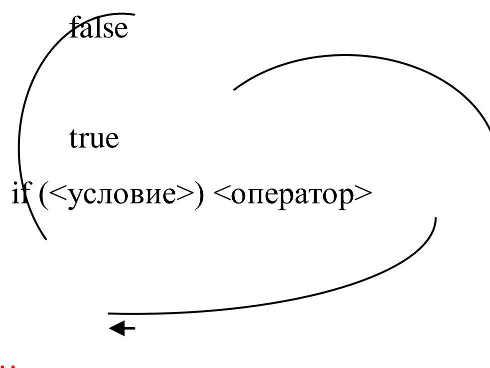
- if (ако) е запазена дума

- <условие> е булев израз;

- <оператор> е произволен оператор.

Семантика

Пресмята се стойността на булевия израз, представящ условието. Ако резултатът е true, изпълнява се <оператор>. В противен случай <оператор> не се изпълнява, т.е.



Фиг. 4.

Забележки:

1. Булевият израз, определящ <условие>, трябва да бъде определен. Огражда се в кръгли скобки.

2. Операторът след условието е точно един. Ако е необходимо няколко оператора да се изпълнят, трябва да се обединят в блок.

Задача 17. Да се напише програма, която намира май-малкото от три дадени реални числа.

Ще реализираме следните стъпки:

а) Въвеждане на стойности на реалните променливи a, b и c.

б) Инициализиране със стойността на a на работна реална променлива min, която ще играе и ролята на изходна променлива.

в) Сравняване на b с min. Ако стойността на b е по-малка от запомнения в min текущ минимум, запомня се b в min. В противен случай, min не се променя. Така min съдържа min{a, b}.

г) Сравняване на c с min. Ако стойността на c е по-малка от запомнения в min текущ минимум, c се запомня в min. В противен случай, min не се променя. Така min съдържа min{a, b, c}.

д) Извеждане на резултата – стойността на min.

Програма Zad17.cpp реализира този алгоритъм.

Program Zad17.cpp

```
#include <iostream.h>
int main()
{cout << "a= ";
 double a;
 cin >> a;
 cout << "b= ";
 double b;
 cin >> b;
 cout << "c= ";
 double c;
 cin >> c;
 double min = a;
 if (b < min) min = b;
 if (c < min) min = c;
 cout << "min{" << a << ", " << b << ", "
    << c << "}= " << min << "\n";
 return 0;
}
```

Ако вместо очаквано реално число, при въвеждане на стойности за променливите a, b и c, се въведе произволен низ, не представляващ число, буферът на клавиатурата, свързан със cin ще изпадне в състояние fail, а обектът cin ще има стойност false. Добрият стил за програмиране изисква в такъв случай програмата да прекъсне изпълнението си с подходящо съобщение за грешка. Програмата от задача 18 реализира този стил.

Задача 18. Да се напише програма, която намира най-малкото от три дадени реални числа. Програмата да извършва проверка за коректност на входните данни.

Програма Zad18.cpp решава задачата.

Program Zad18.cpp

```
#include <iostream.h>
int main()
{cout << "a= ";
 double a;
 cin >> a;
 if (!cin)
 {cout << "Error, bad input \n";
  return 1;
 }
 cout << "b= ";
 double b;
 cin >> b;
 if (!cin)
 {cout << "Error, bad input \n";
  return 1;
 }
 cout << "c= ";
 double c;
 cin >> c;
 if (!cin)
 {cout << "Error, bad input \n";
  return 1;
 }
 double min = a;
 if (b < min) min = b;
 if (c < min) min = c;
 cout << "min{" << a << ", " << b << ", "
    << c << "}= " << min << "\n";
 return 0;
}
```

Ще напомним, че операторът return предизвиква преустановяване работата на програмата, а стойността 1 – че е възникнала грешка.

Задача 19. Да се сортира във възходящ ред редица от три реални числа, запомнени в променливите a, b и c.

Ще реализираме следните стъпки:

а) Въвеждане на стойности за a, b и c.

б) Сравняване на стойностите на a и b. Ако е в сила релацията b<a, извършва се размяна на стойностите на a и b. В противен случай – размяната не се извършва.

в) Сравняване на стойностите на a и c. Ако е в сила релацията c<a, извършва се размяна на стойностите на a и c. В противен случай – размяната не се извършва. След това действие, променливата a съдържа най-малката стойност на редицата.

г) Сравняване на стойностите на b и c. Ако е в сила релацията c<b, извършва се размяна на стойностите на b и c. В противен случай – размяната не се извършва.

е) Извеждане на стойностите на a, b, и c.

Програма Zad19.cpp реализира това описание.

Program Zad19.cpp

```
#include <iostream.h>
#include <iomanip.h>
int main()
{ cout << "a= ";
  double a;
  cin >> a;
  if (!cin)
  {cout << "Error, bad input \n";
   return 1;
  }
  cout << "b= ";
  double b;
  cin >> b;
  if (!cin)
  {cout << "Error, bad input \n";
   return 1;
  }
```

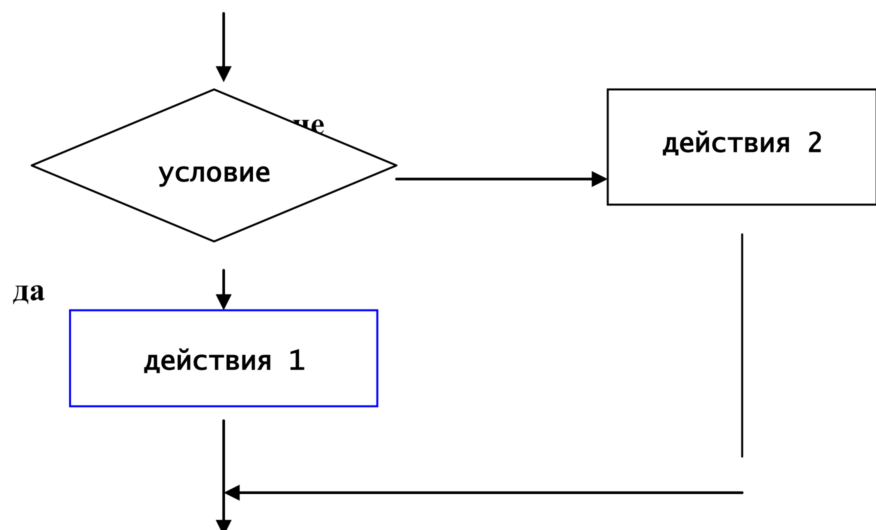
```
cout << "c= ";
double c;
cin >> c;
if (!cin)
{cout << "Error, bad input \n";
 return 1;
}
if (b < a) {double x = a; a = b; b = x;}
if (c < a) {double x = c; c = a; a = x;}
if (c < b) {double x = c; c = b; b = x;}
cout << setprecision(2) << setiosflags(ios :: fixed);
cout << setw(10) << a << setw(10) << b << setw(10)
   << c << "\n";
return 0;
}
```

Забележка: Променливата x е видима (може да се използва) само в блоковете, където е дефинирана.

4.4.2. Оператор if/else

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

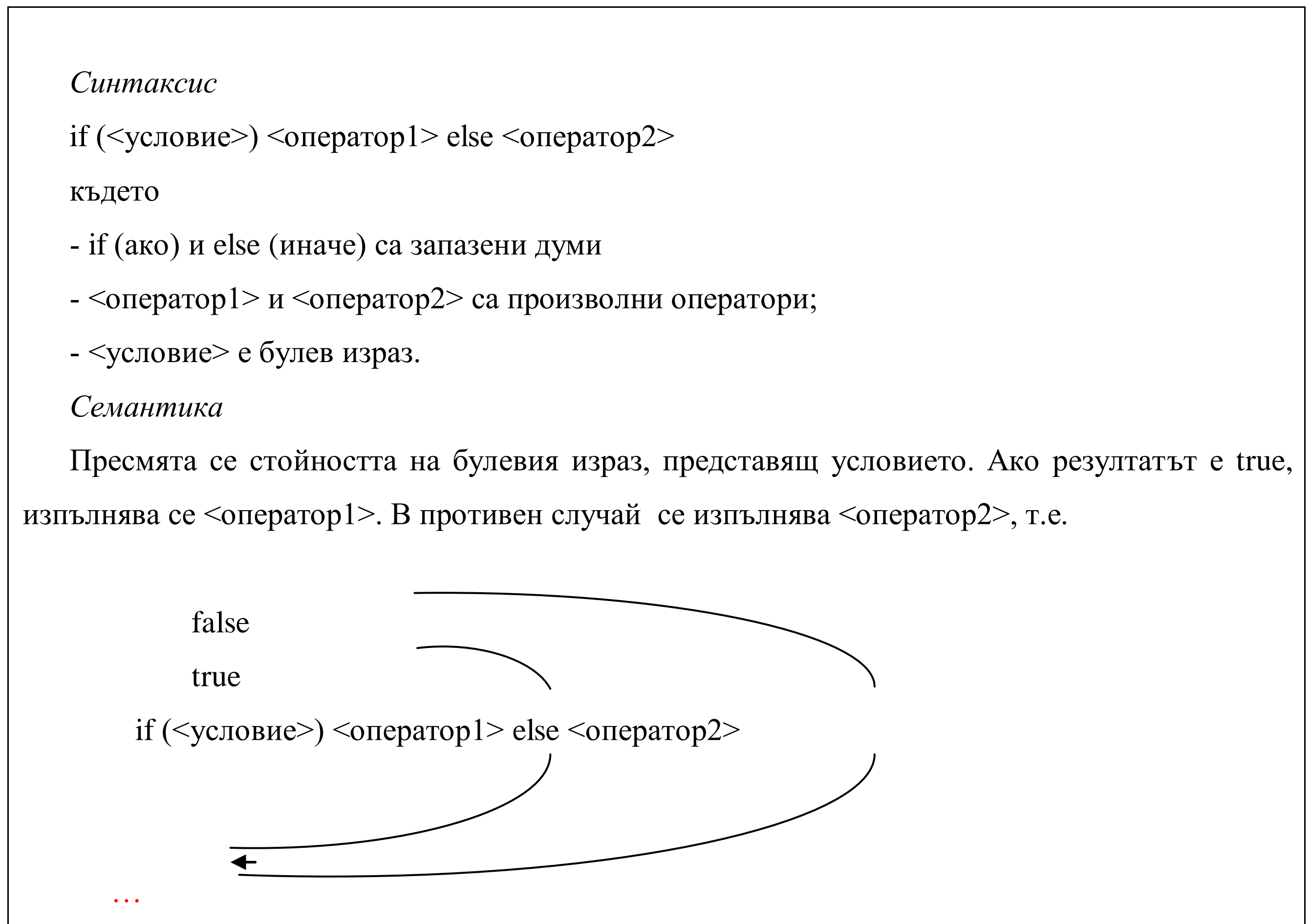
Чрез него се реализира разклоняващ се изчислителен процес от вид, илюстриран на Фиг. 5.



Фиг. 5.

Ако указаното условие е в сила, се изпълняват се едни действия, а ако не – други действия. И в двата случая след това се изпълняват общи действия.

Условието се задава чрез някакъв булев израз, а действия 1 и действия 2 – чрез оператори. Фиг. 6 описва подробно синтаксиса и семантиката на този оператор.



Синтаксис

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

където

- if (ако) и else (иначе) са запазени думи
- <оператор1> и <оператор2> са произволни оператори;
- <условие> е булев израз.

Семантика

Пресмята се стойността на булевия израз, представящ условието. Ако резултатът е true, изпълнява се <оператор1>. В противен случай се изпълнява <оператор2>, т.е.

Фиг. 6.

Забележки:

1. Булевият израз, определящ <условие>, трябва да бъде напълно определен. Задължително се огражда в кръгли скобки.

2. Операторът след условието е точно един. Ако е необходимо няколко оператора да се изпълнят, трябва да се обединят в блок.

3. Операторът след else е точно един. Ако е необходимо няколко оператора да се изпълнят, трябва да се обединят в блок.

Задача 20. Променливата y зависи от променливата x. Зависимостта е следната:
Да се напише програма, която по дадено x намира съответната стойност на y.

$$y = \begin{cases} lg(x) + 1.82, & \text{ако} \quad x \geq 1 \\ x^2 + 7.x + 8.82, & \text{ако} \quad x < 1 \end{cases}$$

Програма Zad20.cpp решава задачата.

Program Zad20.cpp

```
#include <iostream.h>
#include <iomanip.h>
#include <math.h>
int main()
{ cout << "x= ";
  double x;
  cin >> x;
  if (!cin)
  {cout << "Error, bad input!!! \n";
   return 1;
  }
  double y;
  if (x >= 1) y = log10(x) + 1.82;
  else y = x*x - 7*x + 8.82;
  cout << setprecision(3) << setiosflags(ios :: fixed);
  cout << setw(10) << x << setw(10) << y << "\n";
  return 0;
}
```

След въвеждането на стойността на променливата x, програмата извършва проверка за валидност на въведената стойност. Изпълнението на оператора if/else води до пресмятане на стойността на булевия израз x >= 1. Ако тя е true, се изпълнява операторът за присвояване y = log10(x) + 1.82;. В противен случай се изпълнява операторът за присвояване y = x*x - 7*x + 8.82;, след което се извежда резултатът.

Вложени условни оператори

В условните оператори:

if (<условие>) <оператор>

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

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

При влагането е възможно да възникнат двусмислици. Ако в един условен оператор има повече запазени думи if отколкото else, възниква въпросът, за кой от операторите if се отнася съответното else. Например, нека разгледаме оператора

```
if (x >= 0) if ( x >= 5) x = 1/x; else x = -x;
```

Възможни са следните две различни тълкувания на този оператор:

а) if оператор, тялото на който е if/else оператор, т.е.

```
if (x >= 0)
  if (x >= 5) x = 1/x; else x = -x;
```

При това тълкување, ако преди изпълнението на if оператора x има стойност –5, след изпълнението му, стойността на x остава непроменена.

б) if/else оператор, с if оператор след <условие>, т.е.

```
if (x >= 0) if ( x >= 5) x = 1/x;
else x = -x;
```

При това тълкување, ако преди изпълнението на if/else оператора x има стойност –5, след изпълнението му, стойността на x става 5.

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

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

Според това правило, компилаторът на C++ ще приеме първото тълкување за горните вложени условни оператори.

Препоръка: Условен оператор да се влага в друг условен оператор само след else. Ако се налага да се вложи след условието, вложеният условен оператор да се направи блок.

Задачи върху операторите if и if/else

Задача 21. Ако променливата a има стойност 8, определете каква стойност ще има променливата b след изпълнението на оператора

```
if (a > 4) b = 5; else
  if (a < 4) b = -5; else
   if (a == 8) b = 8; else b = 3;
```

Тъй като е в сила условието a > 4, променливата b ще получи стойността 5.

Задача 22. Стойността на y зависи от x. Зависимостта е следната:

$$y = \begin{cases} x, & \text{ако } x \leq 2 \\ 2, & \text{ако } x \in (2, 3] \\ x - 1, & \text{ако } x > 3 \end{cases}$$

Да се напише програма, която по дадено x, намира стойността на y.

Програма Zad22.cpp решава задачата.

Program Zad22.cpp

```
#include <iostream.h>
#include <iomanip.h>
int main()
{ cout << "x= ";
  double x;
  cin >> x;
  if (!cin)
  { cout << "Error, Bad input\n";
    return 1;
  }
  double y;
  if (x <= 2) y = x; else
   if (x <= 3) y = 2; else y = x-1;
```

```
  cout << setprecision(3) << setiosflags(ios :: fixed);
  cout << setw(10) << x << setw(10) << y << "\n";
  return 0;
 }
```

Забележка: В програма Zad22.cpp след първото else е в сила условието x > 2. Затова не е нужно то да се проверява.

Задача 23. Да се напише програма, която въвежда три реални числа a, b и c и извежда 0, ако не съществува триъгълник със страни a, b и c. Ако такъв триъгълник съществува, да извежда 3, 2 или 1 в зависимост от това какъв е триъгълникът - равностранен, равнобедрен или разностранен съответно.

Програма Zad23.cpp решава задачата.

Program Zad23.cpp

```
#include <iostream.h>
int main()
{ cout << "a= ";
  double a;
  cin >> a;
  if (!cin)
  {cout << "Error, bad input \n";
   return 1;
  }
  cout << "b= ";
  double b;
  cin >> b;
  if (!cin)
   {cout << "Error, bad input \n";
    return 1;
   }
  cout << "c= ";
  double c;
  cin >> c;
  if (!cin)
   {cout << "Error, bad input \n";
    return 1;
   }
```

```
  bool x = a <= 0 || b <= 0 || c <= 0 ||
           a+b <= c || a+c <= b || b+c <= a;
   if (x) cout << 0 << "\n"; else
       if (a == b && b == c) cout << 3 << "\n"; else
        if (a == b || a == c || b == c) cout << 2 << "\n"; else
           cout << 1 << "\n";
  return 0;
 }
```

Булевата променлива x е помощна. Тя има стойност true, ако a, b и c не са страни на триъгълник. Получена е след прилагане отрицание на условието a, b и c да са страни на триъгълник, т.е. на условието

a > 0 && b > 0 && c > 0 && a + b > c && a + c > b && b + c > a

като са използвани законите на де Морган.

Закони на де Морган:
!!A е еквивалентно на A
!(A || B) е еквивалентно на !A && !B
!(A && B) е еквивалентно на !A || !B

Задача 24*.* Да се напише програма, която на цялата променлива k присвоява номера на квадранта, в който се намира точка с координати (x, y). Точката не лежи на координатните оси, т.е. x.y ≠ 0.

Програмата Zad24.cpp решава задачата.

Program Zad24.cpp

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

```
if (!cin)
{cout << "Error, Bad input! \n";
 return 1;
}
if (x*y == 0)
{cout << "The input is incorrect! \n";
 return 1;
}
int k;
if (x*y>0) {if (x>0) k = 1; else k= 3;}
else
 if (x>0) k = 4; else k = 2;
cout << "The point is in: " << k << "\n";
return 0;
}
```

4.4.3. Оператор switch

Често се налага да се избере за изпълнение един измежду множество от варианти. Пример за това дава следната задача.

Задача 25. Да се напише програма, която въвежда цифра, след което я извежда с думи.

За решаването на задачата трябва да се реализира следната неелементарна функция:

$$f = \begin{cases} \text{изведи zero,} & \text{ако въведената цифра е 0} \\ \text{изведи one,} & \text{ако въведената цифра е 1} \\ \quad \ldots & \quad \ldots \\ \text{изведи nine,} & \text{ако въведената цифра е 9} \end{cases}$$

Последното може да стане чрез следната програма:

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

```
int main()
{cout << "i= ";
 int i;
 cin >> i;
 if (!cin)
 {cout << "Error, bad input!\n";
  return 1;
 }
 if (i < 0 || i > 9)
 {cout << "Bad input \n";
  return 1;
 }
 else
 if (i == 0) cout << "zero \n";
 else if (i == 1) cout << "one \n";
 else if (i == 2) cout << "two \n";
 else if (i == 3) cout << "three \n";
 else if (i == 4) cout << "four \n";
 else if (i == 5) cout << "five \n";
 else if (i == 6) cout << "six \n";
 else if (i == 7) cout << "seven \n";
 else if (i == 8) cout << "eight \n";
 else if (i == 9) cout << "nine \n";
 return 0;
}
```

В нея са използвани вложени if и if/else оператори, условията на които сравняват променливата i с цифрите 0, 1, 2, …, 9.

Има по-удобна форма за реализиране на това влагане. Постига се чрез оператора за избор на вариант switch.

Програма Zad25.cpp е друго решение на задачата.

Program Zad25.cpp

```
#include <iostream.h>
int main()
```

```
{cout << "i= ";
 int i;
 cin >> i;
 if (!cin)
 {cout << "Error, bad input!\n";
  return 1;
 }
 if (i < 0 || i > 9)
 {cout << "Bad input \n";
  return 1;
 }
 else
 switch (i)
 {case 0 : cout << "zero \n"; break;
  case 1 : cout << "one \n"; break;
  case 2 : cout << "two \n"; break;
  case 3 : cout << "three \n"; break;
  case 4 : cout << "four \n"; break;
  case 5 : cout << "five \n"; break;
  case 6 : cout << "six \n"; break;
  case 7 : cout << "seven \n"; break;
  case 8 : cout << "eight \n"; break;
  case 9 : cout << "nine \n"; break;
 }
 return 0;
}
```

Операторът switch започва със запазената дума switch (ключ), следван от, ограден в кръгли скобки, цял израз. Между фигурните скобки са изброени вариантите на оператора. Описанието им започва със запазената дума case (случай, вариант), следвана в случая от цифра, наречена етикет, двоеточие и редица от оператори.

Изпълнение на програмата

След въвеждането на стойност на променливата i се извършва проверка за коректност на въведеното. Нека въведената стойност е 7. Изпълнението на оператора switch причинява да бъде пресметната стойността на израза i – в случая 7. След това последователно сравнява тази

стойност със стойностите на етикетите до намиране на етикета 7 и изпълнява редицата от оператори след него. В резултат върху екрана се извежда

seven

курсурът се премества на нов ред и се прекъсва изпълнението на оператора switch. Последното е причинено от оператора break в края на редицата от оператори за варианта с етикет 7.

Операторът switch реализира избор на вариант от множество варианти (възможности). Синтаксисът и семантиката му са дадени на Фиг. 7.

Синтаксис

switch (<израз>)
{ case <израз$_1$> : <редица_от_оператори$_1$>
case <израз$_2$> : <редица_от_оператори$_2$>
…
case <израз$_{n-1}$> : <редица_от_оператори$_{n-1}$>
[default : <редица_от_оператори$_n$>]
}

където

- switch (ключ), case (случай, избор или вариант) и default (по премълчаване) са запазени думи на езика;
- <израз> е израз от допустим тип (Типовете bool, int и char са допустими, реалните типове double и float не са допустими). Ще го наричаме още switch-израз.
- <израз$_1$>, <израз$_2$>, …, <израз$_{n-1}$> са константни изрази, задължително с различни стойности.
- <редица_от_оператори$_i$> (i = 1, 2, …, n) се дефинира по следния начин:

<редица_от_оператори> ::= <празно>|
<оператор>|
<оператор><редица_от_оператори>

Семантика

Намира се стойността на switch-израза. Получената константа се сравнява последователно със стойностите на етикетите <израз$_1$>, <израз$_2$>, … При съвпадение, се изпълняват

операторите на съответния вариант и операторите на всички варианти, разположени след него, до срещане на оператор break. В противен случай, ако участва default-вариант, се изпълнява редицата от оператори, която му съответства и в случай, че не участва такъв – не следват никакви действия от оператора switch.

Фиг. 7.

Между фигурните скобки са изброени вариантите на оператора. Всеки вариант (без евентуално един) започва със запазената дума case, следвана от израз (нарича се още case–израз или етикет), който се пресмята по време на компилация. Такива изрази се наричат **константни**. Те не зависят от входните данни. След константния израз се поставя знакът двоеточие, следван от редица от оператори (оператори на варианта), която може да е празна. Сред вариантите може да има един (не е задължителен), който няма case-израз и започва със запазената дума default. Той се изпълнява в случай, че никой от останалите варианти не е бил изпълнен.

Съществува възможност програмистът да съобщи на компилатора, че желае да се изпълни само редицата от оператори на варианта с етикет, съвпадащ със стойността на switch-израза, а не и всички следващи го. Това се реализира чрез използване на оператор break в края на редицата от оператори на варианта. Този оператор предизвиква прекъсване на изпълнението на оператора switch и предаване на управлението на първия оператор след него (Фиг. 8.).

Операторът break принадлежи към групата на т. нар. **оператори за преход**. Тези оператори предават управлението безусловно в някаква точка на програмата.

Синтаксис

break;

Семантика

Прекратява изпълнеието на най-вътрешния съдържащ го оператор switch или оператор за цикъл. Изпълнението на програмата продължава от оператора, следващ (съдържащ) прекъснатия.

Фиг. 8.

Програмистът съзнателно пропуска оператора break, когато за няколко различни стойности от множеството от стойности, трябва да се извършат еднакви действия.

Забележка: Ако в оператора switch не е използван операторът break, ще бъде изпълнена редицата от оператори на варианта, чийто case-израз съвпада със стойността на switch-израза и също всички след него.

Използването на оператора switch има едно единствено предимство пред операторите if и if/else – прави реализацията по-ясна. Основен негов недостатък е, че може да се прилага при много специални обстоятелства, произтичащи от наложените ограничения на типа на switch-израза, а именно, той трябва да е цял, булев или символен. Освен това, използването на оператора break, затруднява доказването на важни математически свойства на програмите, използващи break.

Задачи върху оператора switch

Задача 26. Да се напише програма, която по зададено реално число x намира стойността на един от следните изрази:

y = x - 5
y = sin(x)
y = cos(x)
y = exp(x).

Изборът на желания израз да става по следния начин: при въвеждане на цифрата 1 се избира първият, на 2 – вторият, на 3 – третият и на 4 – четвъртия израз.

Програма Zad25.cpp решава задачата.

```
#include <iostream.h>
#include <iomanip.h>
#include <math.h>
int main()
{cout << "=================================\n";
 cout << "|    y = x-5              -> 1    |\n";
 cout << "|    y = sin(x)           -> 2    |\n ";
 cout << "|    y = cos(x)           -> 3    |\n";
```

```
  cout << "|     y = exp(x)          -> 4    |\n";
  cout << "==================================\n";
  cout << " 1, 2, 3 or 4? \n";
  int i;
  cin >> i;
  if (!cin)
  {cout << "Error, Bad input!!! \n";
   return 1;
  }
  if (i == 1 || i == 2 || i == 3 || i == 4)
  {cout << "x= ";
   double x;
   cin >> x;
   if (!cin)
   {cout << "Error, Bad input!! \n";
    return 1;
   }
   double y;
   switch (i)
   {case 1: y = x - 5;  break;
    case 2: y = sin(x); break;
    case 3: y = cos(x); break;
    case 4: y = exp(x); break;
   }
   cout << "y= " << y << "\n";
  }
  else
  {cout << "Error, Bad choise!!\n";
   return 1;
  }
  return 0;
}
```

Задачи

Задача 1. Явява ли се условен оператор редицата от символи:

   а) `if (x < y) x = 0; else y = 0;`

```
б) if (x > y) x = 0; else cin >> y;
в) if (x >= y) x = 0; y = 0; else cout << z;
г) if (x < y) ; else z = 5;
д) if (x < y < z) then z = z + 1;
е) if (x != y) z = z+1; x = x + y;
```

Задача 2. Кое условие е в сила след запазената дума else на условния оператор:

```
а) if (a > 1 && a < 5) b = 5; else b = 10;
б) if (a < 1 || a > 5) b = a; else a = b;
в) if (a = b || a = c || b = c) c = a + b; else c = a - b;
```

Задача 3. Да се намерят грешките в следните оператори:

```
а) if (1 < x < 2) x = x + 1; y = 0;
   else x = 0; y = y + 1;
б) if (1 < x) && (x < 2)
   {x = x + 1;
    y = 0;
    };
   else
   {x = 0;
    y = y + 1;
   };
```

Задача 4. Да се напише програма, която по дадено реално число x намира стойността на y, където

$$y = \begin{cases} 0, & \text{ако } x \le 0 \\ x^2, & \text{ако } x > 0 \end{cases}$$

Задача 5. Да се напише програма, която по зададени стойности на реалните променливи a, b и c намира:

а) min{a+b+c, a.b.c} + 15.2

б) max{a^2 - b^3 + c, a – 17.3 b, 3.1 a + 3.5 b – 8 c} - 17.9.

Задача 6. Да се напише програма, която увеличава по-малкото от две дадени цели неравни числа пет пъти, а по-голямото число намалява 8 пъти.

Задача 7. Да се напише програма, която въвежда четири реални числа и ги извежда във възходящ (низходящ) ред върху екрана.

Задача 8. Дадени са три числа a, b и c. Да се напише програма, в резултат от изпълнението на която, ако е в сила релацията a ≥ b ≥ c, числата се удвояват, в противен случай числата се заменят с техните абсолютни стойности.

Задача 9. Да се намери стойността на z след изпълнението на операторите

```
z = 0;
if (x > 0) if (y > 0) z = 1; else z = 2;
```

ако:

а) x = y = 1   б) x = 1, y = -1   в) x = -1, y = 1

Задача 10. Да се запише указаното действие чрез един условен оператор:

а) $d = \begin{cases} \max\{a, b\} \text{ ако } x < 0 \\ \min(a, b) \text{ ако } x \geq 0 \end{cases}$

б) d = max(a, b, c)

Задача 11. Да се напише условен оператор, който е еквивалентен на оператора за присвояване

```
x =  a || b && c;
```

където всички променливи са булеви и в който не се използват логически операции (Например, операторът x = not a; е еквивалентен на оператора if (a) x = false; else x = true;).

Задача 12. Да се напише оператор за присвояване, еквивалентен на условния оператор

```
if  (a) x = b; else x = c;
```

(всички променливи са булеви).

Задача 13. Да се напише програма, която по зададено число a, намира корена на уравнението f(x) = 0, където

$$f(x) = \begin{cases} 5xa^{\frac{1}{3}} + |a-1|^{\frac{1}{2}}, & \text{ако } a > 0 \\ e^{ax} - a^2 - 5, & \text{ако } a \leq 0 \end{cases}$$

4.5. Оператори за цикъл

Операторите за цикъл се използват за реализиране на циклични изчислителни процеси.

Изчислителен процес, при който оператор или група оператори се изпълняват многократно за различни стойности на техни параметри, се нарича **цикличен.**

Съществуват два вида циклични процеси:

- индуктивни и
- итеративни.

Цикличен изчислителен процес, при който броят на повторенията е известен предварително, се нарича **индуктивен цикличен процес.**

Пример: По дадени цяло число n и реално число x, да се намери сумата

$$S = 1 + \frac{x}{1!} + \frac{x^2}{2!} + ... + \frac{x^n}{n!}.$$

Ако S има начална стойност 1, за да се намери сумата е необходимо n пъти да се повторят следните действия:

а) конструиране на събираемо

$$\frac{x^i}{i!}, (i = 1, 2, ..., n)$$

б) добавяне на събираемото към S.

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

Пример: По дадени реални числа x и $\varepsilon > 0$, да се намери сумата

$$S = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + ...$$

където сумирането продължава до добавяне на събираемо, абсолютната стойност на което е по-малка от ε.

Ако S има начална стойност 1, за да се намери сумата е необходимо да се повторят следните действия:

а) конструиране на събираемо

$$\frac{x^i}{i!}, (i = 1, 2, ...)$$

б) добавяне на събираемото към S

докато абсолютната стойност на последното добавено към сумата S събираемо стане по-малка от ε.

В този случай, броят на повторенията зависи от стойностите на x и ε.

В езика C++ има три оператора за цикъл:

- *оператор for*

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

- *оператори while и do/while*

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

4.5.1. Оператор for

Използва се основно за реализиране на индуктивни изчислителни процеси. Чрез пример ще илюстрираме използването му.

Задача 26. Да се напише програма, която по зададено естествено число n, намира факториела му.

Тъй като n! = 1.2. … .(n-1).n, следната редица от оператори го реализира:

```
int fact = 1;
fact = fact * 1;
fact = fact * 2;
…
fact = fact * (n-1);
fact = fact * n;
```

В нея операторите за присвояване са написани по следния общ шаблон:

```
fact = fact * i;
```

където i е цяла променлива, получаваща последователно стойностите 1, 2, …, (n-1), n.

Следователно, за да се намери n! трябва да се реализира повтаряне изпълнението на оператора fact = fact * i, за i = 1, 2, …, n. Това може да стане с помощта на оператора for.

Програма Zad26.cpp решава задачата.

Program Zad26.cpp

```
#include <iostream.h>
int main()
{cout << "n= ";
```

```
  int n;
  cin >> n;
  if (!cin)
  {cout << "Error, Bad Input! \n";
   return 1;
  }
  if (n <= 0)
  {cout << "Incorrect Input! \n";
   return 1;
  }
  int fact = 1;
  for (int i = 1; i <= n; i++)
     fact = fact * i;
  cout << n << "! = " << fact << "\n";
  return 0;
 }
```

Операторът for е в одебелен шрифт. Той започва със запазената дума for (за). В кръгли скобки след нея е реализацията на конструкцията i = 1, 2, …, n. Тя се състои от три части: инициализация (int i = 1;), условие (i <= n) и корекция (i++), отделени с ;. Забележете, че операторът i++ не завършва с ;. След този фрагмент е записан операторът fact = fact * i;, описващ действията, които се повтарят. Нарича се **тяло на цикъла**.

Изпълнението на програмата започва с въвеждане стойност на променливата n и проверка валидността на въведеното. Нека n = 3. След дефиницията на цялата променлива fact, в ОП за нея са отделени 4 байта, инициализирани с 1. Изпълнението на оператора for предизвиква за цялата променлива i да бъдат заделени 4 байта, които да се инициализират също с 1. Следва проверка на условието i<=n и тъй като то е истина (1<=3), се изпълнява операторът fact = fact*i;, след което fact получава стойност 1. Изпълнението на оператора i++ увеличава текущата стойност на i с 1 и новата ѝ стойност вече е 2. Отново следва проверка на условието i<=n и тъй като то е истина (2<=3), се изпълнява операторът fact = fact*i;, след което fact получава стойност 2. Изпълнението на оператора i++ увеличава текущата стойност на i с 1 и новата ѝ стойност вече е 3. Пак следва проверка на условието i<=n и тъй като то отново е истина (3 <= 3), се изпълнява операторът fact = fact*i;, след което fact получава стойност 2*3, т.е. 6. Изпълнението на оператора i++

увеличава текущата стойност на i с 1 и новата й стойност вече е 4. Условието i<=n е лъжа и изпълнението на оператора for завършва.

Въпреки, че променливата i е дефинирана в оператора for, тя е “видима” (може да се използва) след изпълнението му, като стойността й е първата, за която стойността на условието i<=n не е в сила (в случая 4).

На Фиг. 9 е дадено детайлно описание на оператора for.

Синтаксис

```
for (<инициализация>; <условие>; <корекция>)
    <оператор>
```

където

- for (за) е запазена дума.
- <инициализация> е или **точно една** дефиниция с инициализация на една или повече променливи, или няколко оператора за присвояване или въвеждане, **отделени със , и не завършващи с ;.**
- <условие> е булев израз.
- <корекция> е един или няколко оператора, **незавършващи с ;.** В случай, че са няколко, отделят се със ,.
- <оператор> е точно един произволен оператор. Нарича се **тяло на цикъла.**

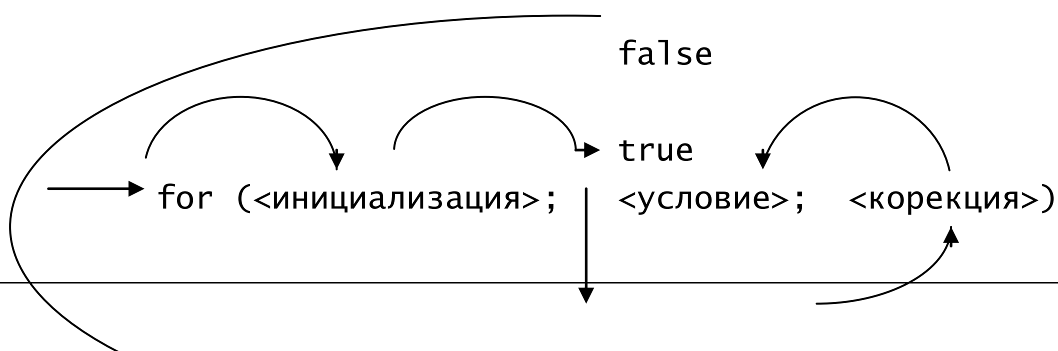
Семантика

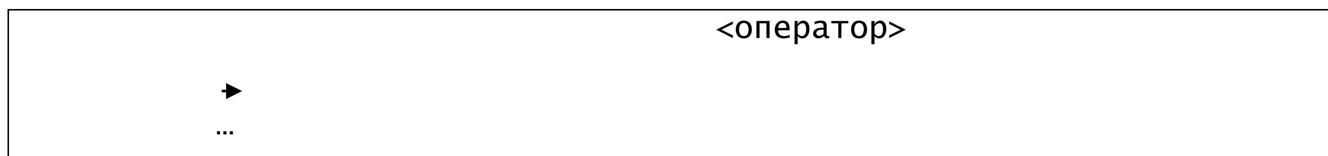
Изпълнението започва с изпълнение на частта <инициализация>. След това се намира стойността на <условие>. Ако в резултат се е получило false, изпълнението на оператора for завършва, без тялото да се е изпълнило нито веднъж. В противен случай последователно се повтарят следните действия:

- Изпълнение на тялото на цикъла;
- Изпълнение на операторите от частта <корекция>;
- Пресмятане стойността на <условие>

докато стойността на <условие> е true.

Следната схема илюстрира изпълнението му:





Фиг. 9.

Възможно е частите <инициализция>, <условие> и <корекция> поотделно или заедно, да са празни. Разделителите (;) между тях обаче трябва да фигурират. Ако частта <условие> е празна, подразбира се true.

Забележки:

1. Тялото на оператора for е точно един оператор. Ако повече оператори трябва да се използват, се оформя блок.

2. Частта <инициализация> се изпълнява само веднъж – в началото на цикъла. Възможно е да се изнесе пред оператора for и остане празна (Пример 1). В нея не са допустими редици от оператори и дефиниция на променливи, т.е. недопустими са:

```
for (int i, i = 4; …
```

или

```
int i;
for (i = 4, int j = 5; …
```

а също две дефиниции, например

```
for(int i=3, double a = 3.5; …
```

Нарича се така, тъй като в нея обикновено се инициализират една или повече променливи.

3. Частта <корекция> се нарича така, тъй като обикновено чрез нея се модифицират стойностите на променливите, инициализирани в частта <инициализация>. Тя може да се премести в тялото на оператора for като се оформи блок от вида {<оператор> <корекция>;} (Пример 2).

4. Ако частта <условие> е празна, подразбира се true. За да се избегне зацикляне, от тялото на цикъла при определени условия трябва да се излезе принудително, например чрез оператора break (Пример 3). Това обаче е лош стил на програмиране и ние не го препоръчваме.

5. Следствие разширената интерпретация на true и false, частта <условие> може да бъде и аритметичен израз. Това също е лош стил на програмиране и не го препоръчваме.

Примери:

```
1. int i = 1;
   for (; i<= n; i++)
       fact = fact* i;
2. for (int i = 1; i<= n;)
   {fact = fact* i;
    i++;
   }
3. for (int i = 1; ; i++)
    if (i > n) break;
    else fact = fact* i;
```

Област на променливите, дефинирани в заглавната част на for

Под област на променлива се разбира мястото в програмата, където променливата може да се използва. Казва се още където тя е "видима".

Съгласно стандарта ANSI, областта на променлива, дефинирана в заглавната част на цикъла for започва от дефиницията ѝ и продължава до края на цикъла.

Това значи, че тези променливи **не** са видими след оператора for, в който са дефинирани, т.е. фрагментът

```
for (int i = 1; i <= n; i++)
{…
}
for (i = 1; i <= m; i++)
{…
}
```

е недопустим - ще предизвика синтактична грешка заради недефинирана променлива i в заглавната част на втория оператор for.

Но тъй като това е ново решение на специалистите, поддържащи езика, повечето реализации в т.число и реализацията на Visual C++ 6.0, използват стария вариант, според който областта на променлива, дефинирана в заглавната част на цикъла for започва от дефиницията й и продължава до края на блока, в който се намира

оператора for. Така, *за реализацията на Visual C++ 6.0,* ***горният фрагмент е напълно допустим. А фрагментът***

```
for (int i = 1; i <= n; i++)
{…
}
for (int i = 1; i <= m; i++)
{…
}
```

сигнализира повторна дефиниция на променливата i.

Задачи върху оператора for

Задача 27. За какво може да бъде използвана следната програма?

```
Program Zad27.cpp
#include <iostream.h>
int main()
{cout << "n= ";
 int n;
 cin >> n;
 if (!cin)
 {cout << "Error, Bad input! \n";
  return 1;
 }
 if (n <= 0)
 {cout << "Incorrect input! \n";
  return 1;
 }
 int f = 1;
 for (int i = 1; i <= n; cin >> i)
 f = f * i;
 cout << f << "\n";
 return 0;
}
```

Забележете, че тази програма илюстрира използването на оператора for за реализиране на итеративни циклични процеси. Това обаче се счита за лош стил за програмиране.

Препоръка: Използвайте оператора for **само** за реализиране на индуктивни циклични процеси. Освен това, ако for има вида:

```
for(i = start; i < (или i <= ) end; i = i + increment)
{ …
}
```

не променяйте i, start, end и increment в тялото на цикъла. Това е лош стил за програмиране. Ако цикличният процес, който трябва да реализирате, не се вмества в тази схема, не използвайте оператора for.

Задача 28. Да се напише програма, която по зададени x – реално и n – естествено число, пресмята сумата

$$s = 1 + \frac{x}{1!} + \frac{x^2}{2!} + ... + \frac{x^n}{n!}.$$

Програма Zad28.cpp решава задачата.

Program Zad28.cpp

```
#include <iostream.h>
int main()
{cout << "x= ";
 double x;
 cin >> x;
 if (!cin)
 {cout << "Error, Bad input! \n";
  return 1;
 }
 cout << "n= ";
 short n;
 cin >> n;
 if (!cin)
 {cout << "Error, Bad input! \n";
  return 1;
 }
 if (n <= 0)
 {cout << "Incorrect input! \n";
  return 1;
 }
 double x1 = 1;
 double s = 1;
```

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

Задача 29. Нека n и m са дадени естествени числа, $n \geq 1$, $m > 1$. Да се напише програма,която определя броя на елементите от серията

$i^3 + 7.i^2 + n^3$, $i = 1, 2, ..., n$.
числа

които са кратни на m.

Програма Zad29.cpp решава задачата.

Program Zad29.cpp

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

```
 {cout << "Incorrect input! \n";
  return 1;
 }
 int br = 0;
 for (int i = 1; i <= n; i++)
 if ((i*i*i + 7*i*i + n*n*n) % 7 == 0) br++;
 cout << "br= " << br << "\n";
 return 0;
}
```

Задача 30. Дадено е естественото число n, $n \geq 1$. Да се напише програма, която намира най-голямото число от серията числа:

$$i^2.\cos(n + \frac{i}{n}),\ i = 1, 2, ..., n.$$

Програма Zad30.cpp решава задачата.

```
Program Zad30.cpp
#include <iostream.h>
#include <math.h>
int main()
{cout << "n= ";
 int n;
 cin >> n;
 if (!cin)
 {cout << "Error, Bad input! \n";
  return 1;
 }
 if (n < 1)
 {cout << "Incorrect input! \n";
  return 1;
 }
 double max = cos(n+1/n);
 for (int i = 2; i <= n; i++)
 {double p = i*i*cos(n+i/n);
  if (p > max) max = p;
 }
 cout << "max= " << max << "\n";
 return 0;
}
```

Задача 31. Да се напише програма, която извежда върху екрана таблицата от стойностите на функциите sin x и cos x в интервала [0, 1].

Програма Zad31.cpp решава задачата.

```
Program Zad31.cpp
#include <iostream.h>
#include <iomanip.h>
#include <math.h>
int main()
{cout << setprecision(5) << setiosflags(ios :: fixed);
 for (double x = 0; x <= 1; x = x + 0.1)
   cout << setw(10) << x << setw(10) << sin(x)
        << setw(10) << cos(x) << "\n";
 return 0;
}
```

4.5.2. Оператор while

Чрез този оператор може да се реализира произволен цикличен процес. С пример ще илюстрираме използването му.

Задача 32. Да се напише програма, която по дадени реални числа x и ε ($\varepsilon > 0$), прилижено пресмята сумата

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

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

В тази задача броят на повторенията предварително не е известен, а зависи от условието $|x1| < \varepsilon$, където с x1 е означено произволно събираемо. За решаването ѝ е необходимо да се премине през следните стъпки:

- Въвеждане на стойности на x и ε.
- Инициализация x1 = 1; s = 1.
- Докато е в сила условието $|x1| \geq \varepsilon$, повтаряне на

```
{ конструиране на ново събираемо x1;
  s = s + x1;
}
```

За реализирането на тези действия, ще използваме оператора за цикъл while.

Програма Zad32.cpp решава задачата.

```
Program Zad32.cpp
#include <iostream.h>
#include <math.h>
int main()
{cout << "x= ";
 double x;
 cin >> x;
 if (!cin)
 {cout << "Error, Bad input! \n";
  return 1;
 }
 cout << "eps= ";
 double eps;
 cin >> eps;
 if (!cin)
 {cout << "Error, Bad input! \n";
  return 1;
 }
 if (eps <= 0)
 {cout << "Incorrect input! \n";
  return 1;
 }
 double x1 = 1;
 double s = 1;
 int i = 1;
 while (fabs(x1) >= eps)
 {x1 = x1 * x / i;
  s = s + x1;
  i++;
 }
 cout << "s=" << s << "\n";
 return 0;
```

```
}
```

Операторът while в нея е в одебелен шрифт. Той започва със запазената дума while (докато), след която оградено в кръгли скобки е условието fabs(x1) >= eps и завършва с оператора

```
{x1 = x1 * x / i;
 s = s + x1;
 i++;
}
```

представляващ тялото на цикъла. Този оператор може да се прочете по следния начин: "**докато е в сила условието fabs(x1) >= eps, повтаряй следното …**".

Ще проследим изпълнението на програмата за x = 1 и eps = 0.5.

След изпълнението на операторите за въвеждане и дефинициите на s, x1 и i, състоянието на паметта е следното:

x	eps	x1	s	i
1.0	0.5	1.0	1.0	1

Изпълнението на оператора за цикъл започва с пресмятане стойността на булевия израз fabs(x1) >= eps и тъй като тя е true, се изпълнява тялото на цикъла, след което имаме:

x	eps	x1	s	i
1.0	0.5	1.0	2.0	2

S е натрупало първите два члена на сумата. Отново се намира стойността на булевия израз fabs(x1) >= eps - пак true и се изпълнява тялото на цикъла. В резултат се получава:

x	eps	x1	s	i
1.0	0.5	0.5	2.5	3

Сега вече S е натрупало първите три члена на сумата. Стойността на булевия израз продължава да бъде true, заради което следва поредно изпълнение на тялото. Имаме:

x	eps	x1	s	i
1.0	0.5	0.166667	2.66667	4

Отново се намира стойността на условието. Тя вече е false, което причинява завършване изпълнението на цикъла while.

Следователно, изпълнението на оператора за цикъл while продължава докато е изпълнено условието след запазената дума while и завършва веднага когато за текущите стойности на неговите параметри то не е в сила.

Задаването на по-голяма точност (по-малка стойност на eps), ще доведе до пресмятане на сумата с по-голяма точност.

На Фиг. 10 е дадено описание на синтаксиса и семантиката на оператора.

Синтаксис

while (<условие>) <оператор>

където

- while (докато) запазена дума;

- <условие> е булев израз;

- <оператор> е произволен оператор. Той описва действията, които се повтарят и се нарича **тяло на цикъла**.

Семантика

Пресмята се стойността на <условие>. Ако тя е false, изпълнението на оператора while завършва без да се е изпълнило тялото му нито веднъж. В противен случай изпълнението на <оператор> и пресмятането на стойността на <условие> се повтарят докато <условие> е true. В първия момент, когато <условие> стане false, изпълнението на while завършва. Изпълнението на while може графично да се илюстрира по следния начин:

false
true
while (<условие>) <оператор>
…

Фиг. 10.

Забележки:

1. Ако е необходимо да се изпълнят могократно няколко оператора, те трябва да се оформят като блок.

2. Следствие разширената интерпретация на true и false, частта <условие> може да бъде и аритметичен израз. Това обаче е лош стил на програмиране и ние не го препоръчваме.

3. Тъй като първото действие, свързано с изпълнението на оператора while, е проверката на условието му, операторът се нарича още **оператор за цикъл с пред-условие**.

4. Операторът

for (<инициализация>; <условие>; <корекция>)

<оператор>

е еквивалентен на

<инициализация>

while (<условие>)

{<оператор>

<корекция>;

}

Пример за това е задача 32.

Задачи върху оператора while

Задача 33. Да се напише програма, която по зададено естествено число, намира факториела му. За целта да се използва оператора while.

Програма Zad33.cpp решава задачата.

```
Program Zad33.cpp
#include <iostream.h>
int main()
{cout << "n= ";
 int n;
 cin >> n;
 if (!cin)
 {cout << "Error, Bad Input! \n";
  return 1;
 }
 if (n <= 0)
 {cout << "Incorrect Input! \n";
  return 1;
 }
 int fact = 1;
 int i = 1;
```

```
 while (i <= n)
 {fact = fact * i;
  i++;
 }
 cout << n << "! = " << fact << "\n";
 return 0;
}
```

Задача 34. Да се напише програма, която въвежда от клавиатурата редица от цели числа и намира средноаритметичното им. Въвеждането да продължава до въвеждане на 0.

Програма Zad34.cpp решава задачата.

Program Zad34.cpp

```
#include <iostream.h>
int main()
{int count = 0;
 double average = 0;
 cout << "> ";
 int number;
 cin >> number;
 while (number != 0)
 {count++;
  average = average + number;
  cout << "> ";
  cin >> number;
 }
 if (count != 0) average = average/count;
 cout << "average= " << average << "\n";
 return 0;
}
```

Забележете отсъствието на проверка за валидност на входните данни. Както вече е известно, това е лош стил за програмиране. Освен това, изборът на числото 0 за край на входните данни, също не винаги е подходящ. Zad35.cpp е подобрение на горното решение.

Задача 35. Да се напише програма, която въвежда от клавиатурата редица от цели числа и намира средноаритметичното им. Въвеждането да продължи до въвеждане на дума, при която cin попада в състояние fail.

Тъй като редицата е числова, за неин край може да служи която и да е дума, която не започва със цифра.

Програма Zad35.cpp решава задачата.

Program Zad35.cpp

```
#include <iostream.h>
int main()
{int count = 0;
 int number;
 double average = 0;
 cout << "> ";
 cin >> number;
 while (cin)
 {count++;
  average = average + number;
  cout << "> ";
  cin >> number;
 }
 if (count != 0) average = average/count;
 cout << "average= " << average << "\n";
 return 0;
}
```

Тъй като cin е стойност на израза cin >> number;, който два пъти е използван в програмата, ще го заменим с него. Получаваме програмата:

```
#include <iostream.h>
int main()
{int count = 0;
 int number;
 double average = 0;
 cout << "> ";
```

```
 while (cin >> number)
 {count++;
  average = average + number;
  cout << "> ";
 }
 if (count != 0) average = average/count;
 cout << "average= " << average << "\n";
 return 0;
}
```

Забележка: Условието cin >> number на оператора while не завършва с ;.

Въпрос: Какво ще е поведението на програми Zad34.cpp и Zad35.cpp ако се промени типът на average от double на int?

Задача 36. Да се напише програма, която пресмята приближено следната безкрайна сума

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

за произволно реално число x от интервала [-1, 1]. Сумирането да продължи докато последното добавено събираемо по модул стане по-малко от ε.

Програма Zad36.cpp решава задачата.

Program Zad36.cpp

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

```
if (x < -1 || x > 1)
{cout << "Incorrect Input! \n";
 return 1;
}
cout << "eps= ";
double eps;
cin >> eps;
if (!cin)
{cout << "Error, Bad input! \n";
 return 1;
}
if (eps <= 0)
{cout << "Incorrect input! \n";
 return 1;
}
 double x1 = x;
 double s = x;
 int i = 2;
 while (fabs(x1) >= eps)
 {x1 = -x1 * x * x / (i*(i+1));
  s = s + x1;
  i = i + 2;
 }
 cout << setprecision(6) << setiosflags(ios :: fixed);
 cout << "s=" << setw(10) << s << "\n";
 return 0;
}
```

Нека сега решим същата задача, но с друго условие за край.

Задача 37. Да се напише програма, която пресмята приближено следната безкрайна сума

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

за произволно реално число x от интервала [-1, 1]. Сумирането да продължи докато абсолютната стойност на разликата на последните две добавени събираеми стане по-малка от ε.

В този случай е необходимо да се конструират и добавят първите две събираеми и чак тогава да се провери условието $|x1 - x2| < \varepsilon$. Ако то е в сила трябва да се съобщи сумата, а в противен случай да се продължи с конструирането и добавянето на следващото събираемо.

Програма zad37.cpp решава задачата.

Program Zad37.cpp

```
#include <iostream.h>
#include <math.h>
int main()
{cout << "x= ";
 double x;
 cin >> x;
 if (!cin)
 {cout << "Error, Bad input! \n";
  return 1;
 }
 if (x < -1 || x > 1)
 {cout << "Incorrect Input! \n";
  return 1;
 }
 cout << "eps= ";
 double eps;
 cin >> eps;
 if (!cin)
 {cout << "Error, Bad input! \n";
  return 1;
 }
 if (eps <= 0)
```

```
 {cout << "Incorrect input! \n";
  return 1;
 }
  double x1 = x;
  double x2 = -x*x*x/6.0;
  double s = x1 + x2;
  int i = 4;
  while (fabs(x1-x2) >= eps)
  {x1 = x2;
   x2 = -x1 * x * x / (i*(i+1));
   s = s + x2;
   i = i + 2;
  }
  cout << "s=" << s << "\n";
  return 0;
}
```

Това решение не е много добро. Тъй като условието за спиране съдържа две последователни събираеми, преди използването на оператора while, те трябва да бъдат конструирани, а след това поддържани в тялото на цикъла. По-добро решение може да се получи като се използва операторът **do/while** – **оператор за цикъл с пост-условие**.

4.5.3. Оператор do/while

Използва се за реализиране на произволни циклични процеси. Ще го илюстрираме чрез пример, след което ще опишем неговите синтаксис и семантика. За целта ще използваме задача 37.

Програма Zad37_1.cpp реализира тази задача, като използва оператора do/while.

Program Zad37_1.cpp

```
#include <iostream.h>
#include <iomanip.h>
#include <math.h>
int main()
```

```
{cout << "x= ";
 double x;
 cin >> x;
 if (!cin)
 {cout << "Error, Bad input! \n";
  return 1;
 }
 if (x < -1 || x > 1)
 {cout << "Incorrect Input! \n";
  return 1;
 }
 cout << "eps= ";
 double eps;
 cin >> eps;
 if (!cin)
 {cout << "Error, Bad input! \n";
  return 1;
 }
 if (eps <= 0)
 {cout << "Incorrect input! \n";
  return 1;
 }
  double x1;
  double x2 = x;
  double s = x;
  int i = 2;
  do
  {x1 = x2;
   x2 = -x1 * x * x / (i*(i+1));
   s = s + x2;
   i = i + 2;
  } while (fabs(x1-x2) >= eps);
  cout << setprecision(5) << setiosflags(ios :: fixed);
  cout << "s=" << setw(10) << s << "\n";
```

```
  return 0;
}
```

Операторът do/while в нея е в одебелен шрифт. Започва със запазената дума do (прави, повтаряй следното), следва оператор (в случая блок), който определя действията, които се повтарят и затова се нарича **тяло на цикъла**. Запазената дума while (докато) отделя тялото на оператора от булевия израз fabs(x1-x2) >= eps. Последният е ограден в кръгли скобки и определя условието за завършване изпълнението на цикъла.

Ще проследим изпълнението на програмата за x = 1 и eps = 0.5.

След изпълнението на операторите за въвеждане и дефинициите на s, x1, x2 и i, състоянието на паметта е следното:

x	eps	x1	x2	s	i
1.0	0.5	-	1.0	1.0	2

Изпълнението на оператора за цикъл започва с изпълнение на тялото на цикъла – блока

```
{x1 = x2;
 x2 = -x1 * x * x / (i*(i+1));
 s = s + x2;
 i = i + 2;
}
```

след което се получава:

x	eps	x1	x2	s	i
1.0	0.5	1.0	-0.16667	0.83333	4

Пресмята се стойността на булевия израз fabs(x1-x2) >= eps и тъй като тя е true, повторно се изпълняват операторите от блока, съставящ тялото. В резултат имаме:

x	eps	x1	x2	s	i
1.0	0.5	-0.16667	0.00833	0.84167	6

Сега вече стойността на булевия израз, определящ условието за завършване, има стойност false. Изпълнението на оператора за цикъл завършва. С извеждането на стойността на сумата s завършва и изпълнението на програмата.

Забелязваме, че в тази програма, настройката на променливите x1 и x2 става в тялото на цикъла do/while, а не извън него. Това се обуславя от факта, че тялото на този вид цикъл поне веднъж ще се изпълни.

Описанието на синтаксиса и семантиката на оператора do/while е илюстрирано на Фиг. 11.

Синтаксис

do

<оператор>

while (<условие>);

където

- do (прави, повтаряй докато …) и while (докато) са запазени думи на езика.

- <оператор> е точно един оператор. Той описва действията, които се повтарят и се нарича тяло на цикъла.

- <условие> е булев израз. Нарича се условие за завършване на цикъла. Огражда се в кръгли скобки.

Семантика

Изпълнява се тялото на цикъла, след което се пресмята стойността на <условие>. Ако то е false, изпълнението на оператора do/while завършва. В противен случай се повтарят действията: изпълнение на тялото и пресмятане стойността на <условие>, докато стойността на <условие> е true. Веднага, след като стойността му стане false, изпълнението на оператора завършва.

Фиг. 11.

Забележки:

1. Между запазените думи do и while стои точно един оператор. Ако няколко действия трябва да се извършат, оформя се блок.

2. *Дефинициите* в тялото, не са видими в <условие>. Например, **не е допустим** фрагментът:

```
double x2 = x;
double s = x;
int i = 2;
do
{double x1 = x2;
 x2 = -x1 * x * x / (i*(i+1));
 s = s + x2;
 i = i + 2;
```

```
 } while (fabs(x1-x2) >= eps);
```
Компилаторът ще съобщи, че x1 не е дефиниран на линия:
```
} while (fabs(x1-x2) >= eps);
```
Следователно, всички променливи в <условие> трябва да са дефинирани извън оператора do/while.

3. Следствие разширената интерпретация на true и false, частта <условие> може да е аритметичен израз. Това е лош стил за програмиране и ние не го препоръчваме.

4. Операторът do/while завършва с ;.

Задачи върху оператора do/while

Задача 38. Да се напише програма, която намира произведението на целите числа от m до n, където m и n са дадени естествени числа и m ≤ n. За целта да се използва операторът do/while.

Програма Zad38.cpp решава задачата.

Program Zad38.cpp
```
#include <iostream.h>
int main()
{cout << "m= ";
 int m;
 cin >> m;
 if (!cin)
 {cout << "Error, Bad Input! \n";
  return 1;
 }
 if (m <= 0)
 {cout << "Incorrect Input! \n";
  return 1;
 }
 cout << "n= ";
 int n;
 cin >> n;
 if (!cin)
```

```
{cout << "Error, Bad Input! \n";
 return 1;
}
if (n <= 0)
{cout << "Incorrect Input! \n";
 return 1;
}
if (m > n)
{cout << "Incorrect Input! \n";
 return 1;
}
int prod = 1;
int i = m;
do
{prod = prod * i;
 i++;
} while (i <= n);
cout << prod << "\n";
return 0;
}
```

Тъй като е в сила релацията $m \leq n$, произведението ще съдържа поне един елемент от редицата от цели числа {m, m+1, m+2, …, n}. Това прави възможно използването на оператора do/while.

Задача 39. Да се напише програма, в резултат от изпълнението на която се изяснява, има ли сред числата от серията: $i^3 - 3.i + n^3$, i = 1, 2, …, n, число, кратно на 5. Ако има, да се изведе true, иначе – false.

Решението на тази задача изисква последователно да се конструират елементите от серията числа. Това продължава до намиране на първото число, кратно на 5, или до изчерпване на редицата без число с това свойство да е намерено.

Програма Zad39_1.cpp е едно решение на задачата.

Program Zad39_1.cpp

```
#include <iostream.h>
int main()
{cout << "n= ";
int n;
cin >> n;
if (!cin)
{cout << "Error, Bad Input!!! \n";
 return 1;
}
if (n <= 0 )
{cout << "Incorrect Input! \n";
 return 1;
}
int i = 0;
int a;
do
{i++;
 a = i*i*i - 3*i + n*n*n;
} while (a%5 != 0 && i < n);
if (a%5 == 0) cout << "true\n";
else cout << "false\n";
return 0;
}
```

Възникват два въпроса:

1) Ако условието условие (a%5 != 0 && i < n) стане лъжа, тъй като се излиза от цикъла, правилно ли следващият оператор определя резултата?

От законите на де Морган следва, че е истина a%5 == 0 || i = n. Ако a%5 == 0, тъй като a е i-тият елемент на серията и $i \leq n$, наистина в серията съществува елемент с исканото свойство. Ако a%5 не е 0, следва че i = n ще е в сила, т.е. a е n – тият елемент и за него свойството не е в сила. Но тъй като са сканирани всички елементи от серията, наистина в нея не съществува елемент с търсеното свойство.

2) Ще се стигне ли до състояние, при което горното условие наистина ще е лъжа?

Условието (a%5 != 0 && i < n) ще е лъжа, ако a%5 == 0 || i = n е в сила и се достига или когато в серията има елемент с търсеното свойство, или е сканирана цялата серия и i указва последния й елемент. Ако в серията няма елемент с исканото свойство, тъй като i е инициализирано с 0 и се увеличава с 1 на всяка стъпка от изпълнението на програмата, в един момент ще стане вярно условието i = n, т.е. цикълът ще завърши изпълнението си.

Друго решение дава програмата Zad39_2.cpp. То не съдържа фрагментът, въвеждащ стойност на променливата n.

Program Zad39_2.cpp

```
#include <iostream.h>
int main()
{ …
 int i = 1;
 int a;
 do
 {a = i*i*i - 3*i + n*n*n;
  i++;
 } while (a%5 != 0 && i <= n);
 if (a%5 == 0) cout << "true\n";
 else cout << "false\n";
 return 0;
}
```

Лошото при това решение, че в тялото на цикъла има разминаване на елемента от серията, запомнен в a, и поредния му номер.

Задача 40. Да се напише програма, която въвежда естествено число и установява, дали цифрата 5 участва в записа на числото.

Програма Zad40.cpp решава задачата.

Program Zad40.cpp

```
#include <iostream.h>
int main()
{cout << "n= ";
```

```
    int n;
    cin >> n;
    if (!cin)
    {cout << "Error, Bad Input!!! \n";
     return 1;
    }
    if (n <= 0 )
    {cout << "Incorrect Input! \n";
     return 1;
    }
    int d;
    do
    {d = n % 10;
     n = n / 10;
    } while (d != 5n && n != 0);
    if (d == 5) cout << "true\n";
    else cout << "false\n";
    return 0;
}
```

В тялото на цикъла последователно се намират цифрата на единиците на числото n и числото без цифрата на единиците си. Това продължава докато поредната цифра е различна от 5 и останалото число е различно от 0.

Задача 41. Нека a е неотрицателно реално число. Да се напише програма, която приближено пресмята квадратен корен от a по метода на Нютон.

Упътване: (метод на Нютон) Дефинира се редица от реални числа x_0, x_1, x_2, x_3, … по следния начин:

$$x_0 = 1$$

$$x_{i+1} = \frac{1}{2}\left(x_i + \frac{a}{x_i}\right), \quad i = 0, 1, 2, ...$$

Сумирането да продължи докато абсолютната стойност на разликата на последните два констуирани елемента на редицата е по-малка от ε, ε>0 е дадено достатъчно малко реално число.

Програма Zad41.cpp решава задачата.

Program Zad41.cpp

```
#include <iostream.h>
#include <iomanip.h>
#include <math.h>
int main()
{cout << "a= ";
 double a;
 cin >> a;
 if (!cin)
 {cout << "Bad Input! \n";
  return 1;
 }
 if (a < 0)
 {cout << "Incorrect Input! \n";
  return 1;
 }
 cout << "eps= ";
 double eps;
 cin >> eps;
 if (!cin)
 {cout << "Bad Input! \n";
  return 1;
 }
 if (eps <= 0 || eps > 0.5)
 {cout << "Incorrect Input! \n";
  return 1;
 }
 double x0;
```

```
double x1 = 1;
do
{x0 = x1;
 x1 = 0.5*(x0 + a / x0);
} while (fabs (x1-x0) >= eps);
cout << setprecision(6) << setiosflags(ios :: fixed);
cout << "sqrt(" << a << ")= " << setw(10) << x1 << "\n";
return 0;
}
```

4.5.4. Вложени оператори за цикъл

Тялото на кой де е от операторите за цикъл е произволен оператор. Възможно е да е оператор за цикъл или блок, съдържащ оператор за цикъл. В тези случаи се говори за вложени оператори за цикъл.

Пример: Програмният фрагмент

```
for (int i = 1; i <= 3; i++)
for (int j = 1; j <= 5; j++)
cout << "(" << i << ", " << j << ")\n";
```

съдържа вложен оператор for и се изпълнява по следния начин: Променливата i получава последователно целите стойности 1, 2 и 3. За всяка от тези стойности, променливата j получава стойностите 1, 2, 3, 4 и 5 и за тях се изпълнява операторът

```
cout << "(" << i << ", " << j << ")\n";
```

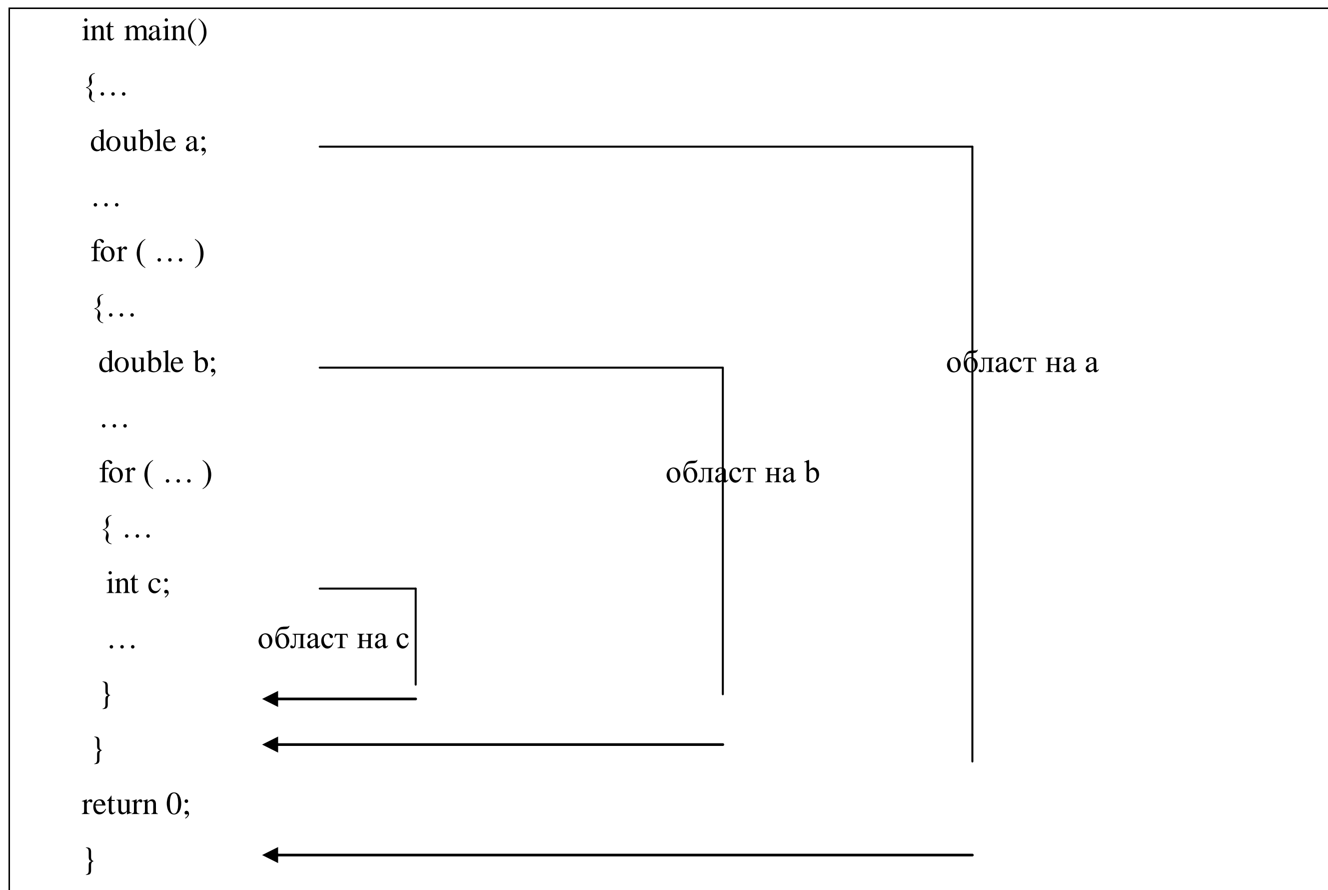
В резултат се конструират и извеждат на отделни редове всички двойки от вида (i, j), където i = 1, 2, 3 и j = 1, 2, 3, 4, 5.

При влагането на цикли, а също при използването на блокове, възникват проблеми, свързани с видимостта на дефинираните променливи.

Област на променлива

Общото правило за дефиниране на променлива е, дефиницията ѝ да е възможно най-близко до мястото където променливата ще се използва най-напред.

Областта на една променлива започва от нейната дефиниция и продължава до края на блока, в който променливата е дефинирана. На Фиг. 12 за променливите a, b и c са определени областите им.



Фиг. 12

Променлива, дефинирана в някакъв блок, се нарича **локална променлива за блока**.

Променлива, дефинирана извън даден блок, но така, че областта ѝ включва блока, се нарича **нелокална променлива за този блок**.

Всяка променлива е видима – може да се използва в областта си. Така b и c не могат да се използват навсякъде в тялото на main, а само с означените области.

Възниква въпросът: *Може ли променливи с еднакви имена да бъдат дефинирани в различни блокове на програма?*

Ако областите на променливите не се припокриват, очевидно няма проблем. Ако обаче те са вложени една в друга, пак е възможно, но е реализирано следното правило: **локалната променлива "скрива" нелокалната в областта си.**

Пример:

```
int main()
{…
 double i;
 …
 for ( … )
 {…
  int i;
   …
 }
 …
}
```

област на int i;

област на double i

Според правилото, в тялото на оператора for е видима цялата променлива i (локална за тялото), а не нелокалната double i.

Ако това води до конфликт с желанията ви, преименувайте например локалната за тялото на for променлива int i.

Задачи върху вложени оператори за цикъл

Задача 42. Да се напише програма, която намира всички решения на деофантовото уравнение a1.x1 + a2.x2 + a3.x3 + a4.x4 = a, където a1, a2, a3, a4 и a са дадени цели числа, а неизвестните x1, x2, x3 и x4 приемат стойности от интервала [p, q] / p и q са дадени цели числа, p<q /.

Програма Zad42.cpp решава задачата.

Program Zad42.cpp

```
#include <iostream.h>
#include <iomanip.h>
int main()
{cout << "a1= ";
 int a1;
```

```
cin >> a1;
if (!cin)
{cout << "Error, Bad Input!\n";
 return 1;
}
cout << "a2= ";
int a2;
cin >> a2;
if (!cin)
{cout << "Error, Bad Input!\n";
 return 1;
}
cout << "a3= ";
int a3;
cin >> a3;
if (!cin)
{cout << "Error, Bad Input! \n";
 return 1;
}
cout << "a4= ";
int a4;
cin >> a4;
if (!cin)
{cout << "Error, Bad Input!\n";
 return 1;
}
cout << "a= ";
int a;
cin >> a;
if (!cin)
{cout << "Error, Bad Input!\n";
 return 1;
}
cout << "p= ";
```

```
int p;
cin >> p;
if (!cin)
{cout << "Error, Bad Input!\n";
 return 1;
}
cout << "q= ";
int q;
cin >> q;
if (!cin)
{cout << "Error, Bad Input!\n";
 return 1;
}
if (p>=q)
{cout << "Error!\n";
 return 1;
}
for (int x1 = p; x1<= q; x1++)
    for (int x2 = p; x2 <= q; x2++)
        for (int x3 = p; x3 <= q; x3++)
            for (int x4 = p; x4 <= q; x4++)
                if (a1*x1 + a2*x2 + a3*x3 + a4*x4 == a)
                    cout << setw(5) << x1 << setw(5) << x2
                         << setw(5) << x3 << setw(5) << x4 << "\n";
return 0;
}
```

Задача 43. Да се напише програма, която проверява дали *съществува* решение на деофантовото уравнение a1.x1 + a2.x2 + a3.x3 + a4.x4 = a в интервала [p, q], където a1, a2, a3, a4, a, p и q са дадени цели числа, $p<q$.

Програма Zad43.cpp решава задачата. Ще пропуснем дефинициите на променливите a1, a2, a3, a4, a, p и q. Те са аналогични на тези от задача 42. Условието $p < q$ прави подходящ оператора do/while.

Program Zad43.cpp

```
#include <iostream.h>
int main()
{…
 if (p>=q)
 {cout << "Error!!\n";
  return 1;
 }
 int x1 = p;
 bool b;
 do
 {int x2 = p;
  do
  {int x3 = p;
   do
   {int x4 = p;
       do
       {b = a1*x1 + a2*x2 + a3*x3 + a4*x4 == a;
        x4++;
       } while (!b && x4 <= q);
    x3++;
   } while (!b && x3 <= q);
   x2++;
  } while (!b && x2 <= q);
  x1++;
 } while (!b && x1 <= q);
 if (b) cout << "yes\n";
 else cout << "no\n";
 return 0;
}
```

Задачи

Задача 1. Да се напише програма, която по дадено реално число x намира стойността на y:

а) $y = (\ldots(((x + 2) x + 3) x + 4) x + \ldots + 10) x + 11$

б) $y = (\ldots(((11x + 10)x + 9)x + 8)x + \ldots + 2)x + 1$.

Задача 2. Да се напише програма, която намира сумата от кубовете на всички цели числа, намиращи се в интервала $(x + \ln x, x^2 + 2x + e^x)$, където $x > 1$.

Задача 3. Дадено е естественото число n ($n \geq 1$). Да се напише програма, която намира броя на тези елементи от серията числа $i^3 - 7.i.n + n^3$, $i = 1, 2, \ldots, n$, които са кратни на 3 или на 7.

Задача 4. Да се напише програма, която по дадено реално число x, намира стойността на сумата

а) $y = \sin x + \sin x^2 + \sin x^3 + \ldots + \sin x^n$;

б) $y = \sin x + \sin^2 x + \sin^3 x + \ldots + \sin^n x$;

в) $y = \sin x + \sin\sin x + \sin\sin\sin x + \ldots + \underbrace{\sin\sin \ldots \sin x}_{n \text{ пъти}}$

Задача 5. Да се напише програма, която намира

$$\sqrt{1 + \sqrt{3 + \sqrt{5 + \ldots \sqrt{97 + \sqrt{99}}}}}$$

Задача 6. Да се напише програма, която по дадено естествено число n (n >= 1) намира стойността на f:

а) $f = (2n)!! = 2.4.6. \ldots .2n$;

б) $f = (2n-1)!! = 1.3.5. \ldots .(2n-1)$;

в) $f = n!!$.

Задача 7. Дадено е естественото число n ($n \geq 1$). Да се напише програма, която пресмята сумата:

а) $\left(\frac{1}{1}\right)^n + \left(\frac{1}{2}\right)^n + \ldots + \left(\frac{1}{n}\right)^n$ б) $\left(\frac{1}{1}\right)^1 + \left(\frac{1}{2}\right)^2 + \ldots + \left(\frac{1}{n}\right)^n$

в) $\left(\frac{1}{1}\right)^n + \left(\frac{1}{2}\right)^{n-1} + \ldots + \left(\frac{1}{n}\right)^1$

(Да не се използват функциите exp и log).

Задача 8. Да се напише програма, която извежда в нарастващ ред всички трицифрени естествени числа, които не съдържат еднакви цифри (/ и % да не се използват).

Задача 9. Да се напише програма, която намира и извежда броя на точките с цели координати, попадащи в кръга с радиус R (R > 0) и център - координатното начало.

Задача 10. Да се напише програма, която извежда таблицата на истинност за булевата функция f = (a and b) or not (b or c) в следния вид

```
a      b      c      f
------------------------------------
true   true   true   true
true   true   false  true
          . . .
false  false  false  true
```

Задача 11. Да се напише програма, която извежда върху екрана следната таблица:

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
1 2 3 4 5 6
1 2 3 4 5 6 7
```

Задача 12. Дадено е естествено число n (n ≥ 1). Да се напише програма, която намира и извежда първите n елемента от серията

$$\left(1 + \frac{1}{i}\right)^i, \; i = 1, 2, 3, \ldots$$

числа

(Да не се използват функциите exp и log).

Задача 13. Едно естествено число е съвършено, ако е равно на сумата от своите делители (без самото число). Например, 6 е съвършено, защото 6 = 1+2+3. Да се напише програма, която намира всички съвършени числа ненадминаващи дадено естествено число n.

Задача 14. Да се напише програма, която намира всички трицифрени числа от интервала [m, n], на които като се задраска цифрата на

десетиците, намаляват цяло число пъти (m и n са дадени естествени числа, $m < n$).

Задача 15. Да се напише програма, която намира всички четирицифрени числа от интервала [m, n], на които като се задраска цифрата на стотиците, се делят на 11 (m и n са дадени естествени числа, $m < n$).

Задача 16. Да се напише програма, която намира всички четирицифрени числа от интервала [m, n], в записа на които участва цифрата 5 (m и n са дадени естествени числа, $m < n$).

Задача 17. Да се напише програма, която намира всички четирицифрени числа от интервала [m, n], цифрите на които образуват намаляваща редица (m и n са дадени естествени числа, $m < n$).

Задача 18. Да се напише програма, която намира всички петцифрени числа от интервала [m, n], цифрите на които са различни (m и n са дадени естествени числа, $m < n$).

Задача 19. Дадено е естествено число n ($n > 1$). Да се напише програма, която намира всички прости числа от интервала [2, n].

Задача 20. Да се напише програма, която намира всички прости делители на дадено естествено число n.

Задача 21. Да се напише програма, която по дадени реални числа x, a и ε,ε>0,

$$S = 1 + \frac{a.x}{1!} + \frac{a(a-1)}{2!}.x^2 + ... + \frac{a(a-1)...(a-n+1)}{n!}x^n + ...$$

приближено пресмята сумата.

Събирането да продължи докато бъде добавено събираемо, абсолютната стойност на което е по-малка от ε ($\varepsilon > 0$ е дадено реално достатъчно малко число).

Задача 22. Да се напише програма, която намира броя на цифрите в десетичния запис на дадено естествено число.

Задача 23. Да се напише програма, която проверява дали дадено естествено число е щастливо, т.е. едно и също е при четене отляво надясно и отдясно наляво.

Задача 24. Да се напише програма, която проверява дали сумата от цифрите на дадено естествено число е кратна на 3.

Задача 25. Да се напише програма, която намира всички естествени числа, ненадминаващи дадено естествено число n, които при преместване на първата им цифра най-отзад, се увеличават k пъти (k е дадено естествено число, $k > 1$).

Задача 26. Да се напише програма, която намира всички естествени числа от интервала [m, n], на които като се задраска k – тата цифра (отляво надясно), намаляват цяло число пъти (m, n и k са дадени естествени числа, m < n).

Задача 27. Да се напише програма, която намира всички естествени числа от интервала [m, n], на които като се задраска k – тата цифра (надясно наляво), намаляват цяло число пъти (m, n и k са дадени естествени числа, m < n).

Задача 28. За дадено естествено число да се провери дали цифрите му, гледани отляво надясно, образуват монотонно растяща редица.

Задача 29. За дадено естествено число да се провери дали цифрите му са различни.

Задача 30. Да се намерят всички прости делители на дадено естествено число.

Задача 31. Числата на Фибоначи се дефинират по следния начин:

$$\mathrm{fib}(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ \mathrm{fib}(n-1) + \mathrm{fib}(n-2), & n > 1 \end{cases}$$

Да се напише програма, която намира сумата на числата на Фибоначи от интервала [a, b] (a и b са дадени естествени числа).

Задача 32. За естествените числа n и m операцията ++ се определя по следния начин: n оо m = n + m + n%m. Да се напише програма, която намира всички двойки (n, m) от естествени числа, за които е в сила
n оо m = m оо n (m и n са естествени числа от интервала [a, b]).

Задача 33. За естествените числа n и m операцията ++ се определя по следния начин: n оо m = n + m + n%m. Да се напише програма, която проверява дали съществува двойкА (n, m) от естествени числа, за която е в сила релацията n оо m = m оо n (m и n са естествени числа от интервала [a, b]).

Допълнителна литература

7. К. Хорстман, Принципи на програмирането със C++, С., СОФТЕХ, 2000.
8. Ст. Липман, Езикът C++ в примери, “КОЛХИДА ТРЕЙД” КООП, С. 1993.

9. B. Stroustrup, C++ Programming Language. Third Edition,Addison – Wesley, 1997.

Глава 6

Съставни типове данни.

Тип масив

1. Структура от данни масив

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

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

- **логическо описание на структурата**, което я описва на базата на декомпозицията й 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.

В предходните глави разгледахме структурите числа и символи. За всяка от тях в езика C++ са дадени съответни типове данни, които ги реализират. Тъй като елементите на тези структури се състоят от една компонента, те 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.**

В тази глава ще разгледаме структурата от данни масив и средствата, които я реализират.

Логическо описание

Масивът е крайна редица от фиксиран брой елементи от един и същ тип. Към всеки елемент от редицата е възможен пряк достъп, който се осъществява чрез индекс. Операциите включване и изключване на елемент в/от масива са недопустими, т.е. масивът е статична структура от данни.

Физическо представяне

Елементите на масива се записват последователно в паметта на компютъра, като за всеки елемент на редицата се отделя определено количество памет.

В езика C++ структурата масив се реализира чрез типа масив.

2. Тип масив

В C++ структурата от данни масив е реализирана малко ограничено. Разглежда се като крайна редица от елементи от един и същ тип с пряк достъп до всеки елемент, осъществяващ чрез индекс с цели стойности, започващи от 0 и нарастващи с 1 до указана горна граница.

Задаване на масив

Типът масив се определя чрез задаване на типа и броя на елементите на редицата, определяща масив. Нека T е име или дефиниция на произволен тип, различен от псевдоним, void и функционален. За типа T и константния израз от интегрален или изброен тип с положителна стойност size, T[size] е тип масив от size елемента от тип T. Елементите се индексират от 0 до size–1. T се нарича **базов тип** за типа масив, а size – горна граница.

Примери:

int[5] е масив от 5 елемента от тип int, индексирани от 0 до 4;

double[10] е масив от 10 елемента от тип double, индексирани от 0 до 9;

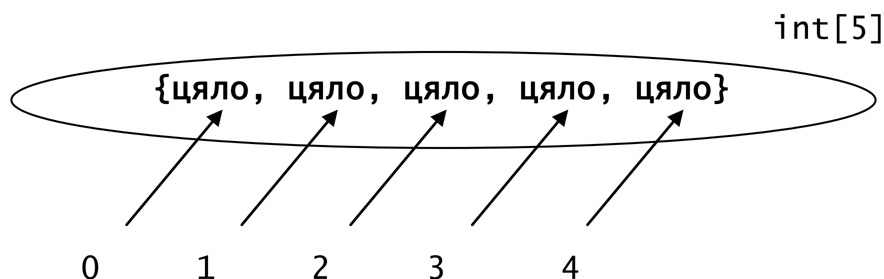
bool[4] е масив от 4 елемента от тип bool, индексирани от 0 до 3.

Множество от стойности

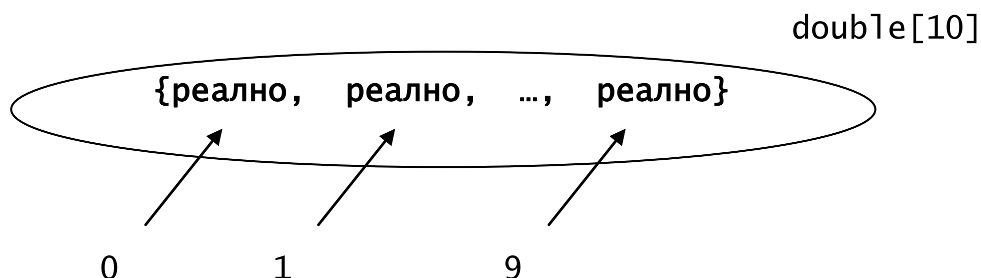
Множеството от стойности на типа T[size] се състои от всички редици от по size елемента, които са произволни константи от тип T. Достъпът до елементите на редиците е пряк и се осъществява с помощта на индекс, като достъпът до първия елемент се осъществява с индекс със стойност 0, до последния – с индекс със стойност size-1, а до всеки от останалите елементи – с индекс със стойност с 1 по-голяма от тази на индекса на предишния елемент.

Примери:

1. Множеството от стойности на типа int[5] се състои от всички редици от по 5 цели числа. Достъпът до елементите на редиците се осъществява с индекс със стойности 0, 1, 2, 3 и 4.



2. Множеството от стойности на типа double[10] се състои от всички редици от по 10 реални числа. Достъпът до елементите на редиците се осъществява с индекс със стойности 0, 1, 2, 3 и т.н. 9.



Елементите от множеството от стойности на даден тип масив са **константите** на този тип масив.

Примери:

1. Следните редици {1,2,3,4,5}, {-3, 0, 1, 2, 0}, {12, -14, 8, 23, 1000} са константи от тип int[5].

2. Редиците {1.5, -2.3, 3.4, 4.9, 5.0, -11.6, -123, 13.7, -32.12, 0.98}, {-13, 0.5, 11.9, 21.98, 0.03, 1e2, -134.9, 0.09, 12.3, 15.6} са константи от тип double[10].

Променлива величина, множеството от допустимите стойности на която съвпада с множеството от стойности на даден тип масив, се нарича променлива от дадения тип масив. Понякога ще я наричаме само масив.

Фиг. 1 определя дефиницията на променлива от тип масив. Тук общоприетият запис е нарушен. Променливата се записва между името на типа и размерността.

<дефиниция_на_променлива_от_тип_масив> ::=
T <променлива>[size]; |
T <променлива>[size] = {<редица_от_константни_изрази>};

където

T е име или дефиниция на произволен тип, различен от псевдоним, void, функционален;

<променлива> ::= <идентификатор>

size е константен израз от интегрален или изброен тип със *положителна* стойност;

<редица_от_константни_изрази> ::= <константен_израз>|
<константен_израз>, <редица_от_константни_изрази>

като константните изрази са от тип T или от тип, съвместим с него.

Фиг. 1.

Примери:

```
int a[5];
double c[10];
bool b[3];
enum {FALSE, TRUE} x[20];
double p[4] = {1.25, 2.5, 9.25, 4.12};
```

Вторият случай от дефиницията от Фиг. 1 се нарича **дефиниция на масив с инициализация.** При нея е възможно size да се пропусне. Тогава за стойност на size се подразбира броят на константните изрази, изброени при инициализацията. Ако size е указано и изброените константни изрази в инициализацията са по-малко от size, останалите се инициализират с 0.

Примери:

1. Дефиницията

```
int q[5] =  {1, 2, 3};
```

е еквивалентна на

```
int q[] =  {1, 2, 3, 0, 0};
```

2. Дефиницията

```
double r[] = {0, 1, 2, 3};
```

е еквивалентна на

```
double r[4] = {0, 1, 2, 3};
```

Забележка: Не са възможни конструкции от вида:

```
int q[5];
q = {0, 1, 2, 3, 4};
```

а също

```
int q[];
```

и

```
double r[4] = {0.5, 1.2, 2.4, 1.2, 3.4};
```

Фрагментите

<променлива>[size] и

<променлива>[size] = {<редица_от_константни_изрази>}

от дефиницията от Фиг. 1. могат да се повтарят. За разделител се използва знакът запетая.

Пример: Дефиницията

```
double m1[20], m2[35], proben[30];
```

е еквивалентна на дефинициите

```
double m1[20];
double m2[35];
double proben[30];
```

Инициализацията е един начин за свързване на променлива от тип масив с конкретна константа от множеството от стойности на този тип масив. Друг начин предоставят т.нар. индексирани променливи. С

всяка променлива от тип масив е свързан набор от индексирани променливи. Фиг. 2. илюстрира техния синтаксис.

```
   <индексирана_променлива> ::=
         <променлива_от_тип_масив>[<индекс>]
където
   <индекс> е израз от интегрален или изброен тип.
   Всяка индексирана променлива е от базовия тип.
```

Фиг. 2.

Примери:

1. С променливата a от примера по-горе са свързани индексираните променливи a[0], a[1], a[2], a[3] и a[4], които са от тип int.
2. С променливата b са свързани индексираните променливи b[0], b[1],…, b[9], които са от тип double.
3. С променливата x са свързани индексираните променливи x[0], x[1],…, x[19], които са от тип enum {FALSE, TRUE}.

Дефиницията на променлива от тип масив не само свързва променливата с множеството от стойности на указания тип, но и отделя определено количество памет (обикновено 4в), в която записва адреса в паметта на първата индексирана променлива на масива. Останалите индексирани променливи се разполагат последователно след първата. За всяка индексирана променлива се отделя по толкова памет, колкото базовият тип изисква.

Пример:

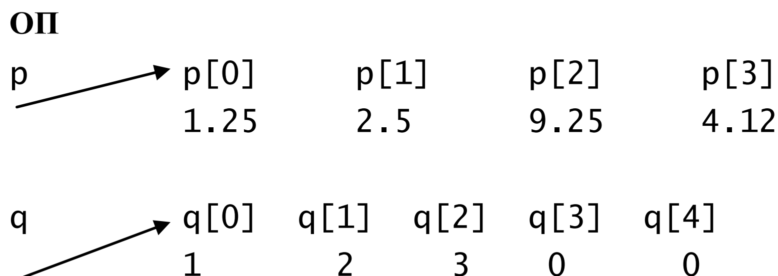
ОП

a	a[0]	a[1]	…	a[4]	b	b[0]	b[1]	…	b[9]	…
адрес на a[0]	-	-		-	адрес на b[0]	-	-		-	
4в	4в	4в	…	4в	4в	8в	8в	…	8в	

За краткост, вместо "адрес на a[0]" ще записваме стрелка от a към a[0]. Съдържанието на отделената за индексираните променливи памет

е неопределено освен ако не е зададена дефиниция с инициализация. Тогава в клетките се записват инициализиращите стойности.

Пример: Разпределението на паметта за променливите p и q, дефинирани в примерите по-горе, е следното:



Операции и вградени функции

Не са възможни операции над масиви като цяло, но всички операции и вградени функции, които базовият тип допуска, са възможни за индексираните променливи, свързани с масива.

Пример: Недопустими са:

```
int a[5], b[5];
cin >> a >> b;
a = b;
```

а също a == b или a != b.

Операторът

```
cout << a;
```

извежда адреса на a[0].

Задачи върху тип масив

Задача 48. Да се напише програма, която въвежда последователно n числа, след което ги извежда в обратен ред.

Програма Zad48.cpp решава задачата.

```
Program Zad48.cpp
#include <iostream.h>
int main()
{double x[100];
 cout << "n= ";
 int n;
```

```
cin >> n;
if (!cin)
{cout << "Error, Bad input! \n";
 return 1;
}
if (n < 0 || n > 100)
{cout << "Incorrect input! \n";
 return 1;
}
for (int i = 0; i <= n-1; i++)
{cout << "x[" << i << "]= ";
 cin >> x[i];
 if (!cin)
 {cout << "Error, Bad Input! \n";
  return 1;
 }
}
for (i = n-1; i >= 0; i--)
   cout << x[i] << "\n";
return 0;
}
```

Изпълнение на програма Zad48.cpp

Дефиницията double x[100]; води до отделяне на 800B ОП, които се именуват последователно с x[0], x[1], …, x[99] и са с неопределено съдържание. Освен това се отделят 4B ОП за променливата x, в които записва адресът на индексираната променлива x[0]. Следващият програмен фрагмент въвежда стойност на n (броя на елементите на масива, които ще бъдат използвани). Операторът

```
for (int i = 0; i <= n-1; i++)
 {cout << "x[" << i << "]= ";
  cin >> x[i];
  if (!cin)
  {cout << "Error, Bad Input! \n";
   return 1;
  }
 }
```

въвежда стойности на целите променливи x[0], x[1], …, x[n-1]. Всяка въведена стойност е предшествана от подсещане. Операторът

```
for (i = n-1; i >= 0; i--)
    cout << x[i] << "\n";
```

извежда в обратен ред компонентите на масива x.

Забележка: Фрагментите:

```
…
cout << "n= ";
int n;
cin >> n;
int x[n];
…
```

и

```
…
int n = 10;
int x[10];
```

са недопустими, тъй като n не е константен израз. Фрагментът

```
…
const int n = 10;
double x[n];
```

е допустим.

3. Някои приложения на структурата от данни масив

Търсене на елемент в редица

Нека са дадени редица от елементи a_0, a_1, …, a_{n-1}, елемент x и релация r. Могат да се формулират две основни задачи, свързани с търсене на елемент в редицата, който да е в релация r с елемента x.

а) Да се намерят **всички елементи** на редицата, които са в релация r с елемента x.

б) Да се установи, **съществува ли елемент** от редицата, който е в релация r с елемента x.

Съществуват редица методи, които решават едната, другата или и двете задачи. Ще разгледаме метода на **последователното търсене**, чрез който магат да се решат и двете задачи. Методът се състои в следното: последователно се обхождат елементите на редицата и за всеки елемент се проверява дали е в релация r с елемента x. При първата задача процесът продължава до изчерпване на редицата, а при втората – до намиране на първия елемент a_k (k = 0, 1, …, n-1),

който е в релация r с x, или до изчерпване на редицата без да е намерен елемент с търсеното свойство.

Следващите четири задачи илюстрират този метод.

Задача 49. Дадени са редицата от цели числа a_0, a_1, …, a_{n-1} ($n \geq 1$) и цялото число x. Да се напише програма, която намира колко пъти x се съдържа в редицата.

В случая релацията r е операцията ==. Налага се всеки елемент на редицата да бъде сравнен с x, т.е. имаме задача от първия вид. Тя описва индуктивен цикличен процес.

Програма Zad49.cpp решава задачата.

```
Program Zad49.cpp
#include <iostream.h>
int main()
{int a[20];
 cout << "n= ";
 int n;
 cin >> n;
 if (!cin)
 {cout << "Error, Bad input! \n";
  return 1;
 }
 if (n < 1 || n > 20)
 {cout << "Incorrect input! \n";
  return 1;
 }
 int i;
 for (i = 0; i <= n-1; i++)
 {cout << "a[" << i << "]= ";
  cin >> a[i];
  if (!cin)
  {cout << "Error, Bad input! \n";
   return 1;
  }
 }
 int x;
 cout << "x= ";
```

```
 cin >> x;
 if (!cin)
 {cout << "Error, Bad input! \n";
  return 1;
 }
 int br = 0;
 for (i = 0; i <= n-1; i++)
   if (a[i] == x) br++;
 cout << "number = " << br << "\n";
 return 0;
}
```

Задача 50. Дадени са редицата от цели числа a_0, a_1, …, a_{n-1} ($n \geq 1$) и цялото число x. Да се напише програма, която проверява дали x се съдържа в редицата.

В този случай се изисква при първото срещане на елемент от редицата, който е равен на x, да се преустанови работата с подходящо съобщение. Броят на сравненията на x с елементите от редицата е ограничен отгоре от n, но не е известен.

Програма Zad50.cpp решава задачата. Фрагментът, реализиращ входа е същия като в Zad49.cpp и затова е пропуснат.

```
Program Zad50.cpp
#include <iostream.h>
int main()
{int a[20];
 …
 i = 0;
 while (a[i] != x && i < n-1)
    i++;
 if (a[i] == x) cout << "yes \n";
 else cout << "no \n";
 return 0;
}
```

Обхождането на редицата става чрез промяна на стойностите на индекса i - започват от 0 и на всяка стъпка от изпълнението на тялото на цикъла се увеличават с 1. Максималната им стойност е n-1.

При излизането от цикъла ще е в сила отрицанието на условието (a[i] != x && i < n-1), т.е. (a[i] == x || i == n-1). Ако е в сила a[i] == x, тъй като сме осигурили a[i] да е елемент на редицата, отговорът “yes” е коректен. В противен случай е в сила i == n-1, т.е. сканиран е и последният елемент на редицата и за него не е вярно a[i] == x. Това е реализирано чрез отговора “no” от алтернативата на условния оператор.

Фрагментът

```
i = -1;
do
i++;
while (a[i] != x && i < n-1);
if (a[i] == x) cout << "yes \n";
else cout << "no \n";
```

реализира търсенето чрез използване на оператора do/while.

Задача 51. Да се напише програма, която установява, дали редицата от цели числа a_0, a_1, ..., a_{n-1} е монотонно намаляваща.

а) За решаването на задачата е необходимо да се установи, дали за всяко i (0 ≤ i ≤ n-2) е в сила релацията a[i] >= a[i+1]. Това може да се реализира като се провери дали броят на целите числа i (0≤i≤n-2), за които е в сила релацията a[i] ≥ a[i+1], е равен на n-1.

Програмата Zad51_1.cpp реализира този начин за проверка дали редица е монотонно намаляваща. Фрагментите, реализиращи въвеждането на n и масива a, са известни вече и затова са пропуснати.

```
Program Zad51_1.cpp
#include <iostream.h>
int main()
{int a[100];
 // дефиниране и въвеждане на стойност на n
 …
 // въвеждане на масива a
 …
 int br = 0;
 for (i = 0; i <= n-2; i++)
    if (a[i] >= a[i+1]) br++;
```

```
 if (br == n-1) cout << "yes \n";
 else cout << "no \n";
 return 0;
}
```

б) Задачата може да се сведе до търсене на i (i = 0, 1,..., n-2), така че ai < ai+1, т.е. до задача за съществуване.

Програма Zad51_2.cpp реализира този начин за проверка дали редица е монотонно намаляваща. Фрагментите, реализиращи въвеждането на n и масива a отново са пропуснати.

```
Program Zad51_2.cpp;
#include <iostream.h>
int main()
{int a[100];
 //въвеждане на размерността n и масива a
 …
 i = 0;
 while (a[i] >= a[i+1] && i < n-2) i++;
 if (a[i] >= a[i+1]) cout << "yes \n";
 else cout << "no \n";
 return 0;
}
```

Решение б) е по-ефективно, тъй като при първото срещане на a[i], така че релацията a[i] < a[i+1] е в сила, изпълнението на цикъла завършва. Решение а) реализира последователно търсене е пълно изчерпване, а решение б) – задача за съществуване на елемент в редица, който е в определена релация с друг елемент (в случая съседния му).

Задача 52. Да се напише програма, която установява, дали редицата от цели числа a_0, a_1, ..., a_{n-1} се състои от различни елементи.

а) За решаването на задачата е необходимо да се установи, дали за всяка двойка (i, j): 0 ≤ i ≤ n-2 и i+1 ≤ j ≤ n-1 е в сила релацията a[i] != a[j]. Това може да се постигне като се провери дали броят на двойките (i, j): 0 ≤ i ≤ n-2 и i+1 ≤ j ≤ n-1, за които е в сила релацията a[i] != a[j], е равен на n*(n-1)/2.

Програма Zad52_1.cpp реализира горната формулировка на задачата – търсене с пълно изчерпване. Фрагментите, реализиращи въвеждането на n и масива a отново са пропуснати.

```
Program Zad52_1.cpp
#include <iostream.h>
int main()
{int a[100];
 //въвеждане на размерността n и масива a
 …
 int br = 0;
 for (i = 0; i <= n-2; i++)
    for (int j = i+1; j <= n-1; j++)
       if (a[i] != a[j]) br++;
 if (br == n*(n-1)/2) cout << "yes \n";
 else cout << "no \n";
 return 0;
}
```

б) Задачата може да се сведе до проверка за съществуване на двойка индекси (i, j): $0 \le i \le n-2$ и $i+1 \le j \le n-1$, за които не е в сила релацията a[i] != a[j]. Програма Zad52_2.cpp решава задачата.

```
Program Zad52_2.cpp
#include <iostream.h>
int main()
{int a[100];
 // въвеждане стойности на размерността n и масива a
 …
 i = -1;
 int j;
 do
 {i++;
  j = i+1;
  while (a[i] != a[j] && j < n-1) j++;
 } while (a[i] != a[j] && i < n-2);
 if (a[i] != a[i+1]) cout << "yes \n";
 else cout << "no \n";
 return 0;
```

```
}
```

Решение б) е по-ефективно, тъй като при първото срещане на a[i] и a[j], така че релацията a[i] == a[j] е в сила, изпълнението на операторите за цикъл завършва. То реализира задача за съществуване на метода за търсене.

Сортиране на редица

Съществуват много методи за сортиране на редици от елементи. В тази глава ще разгледаме **метода на пряката селекция** и чрез него ще реализираме възходяща сортировка на редица.

Метод на пряката селекция

Разглежда се редицата a_0, a_1, …, a_{n-1} и се извършват следните действия:

- Намира се k, така че $a_k = min\{a_0, a_1, …, a_{n-1}\}$.
- Разменят се стойностите на a_k и a_0.

Така на първо място в редицата се установява най-малкият ѝ елемент.

Разглежда се редицата a_1, a_2, …, a_{n-1} и се извършват действията:

- Намира се k, така че $a_k = min\{a_1, a_2, …, a_{n-1}\}$.
- Разменят се стойностите на a_k и a_1.

Така на второ място в редицата се установява следващият по големина елемент на редицата и т.н.

Разглежда се редицата a_{n-2}, a_{n-1} и се извършват действията:

- Намира се k, така че $a_k = min\{a_{n-2}, a_{n-1}\}$.
- Разменят се стойностите на a_k и a_{n-2}.

Получената редица е сортирана във възходящ ред.

Задача 53. Да се сортира във възходящ ред по метода на пряката селекция числовата редица a_0, a_1, …, a_{n-1} ($n \geq 1$).

Програма Zad53.cpp решава задачата. Фрагментът за въвеждане стойности на размерността n и масива a отново е пропуснат.

```
Program Zad53.cpp
#include <iostream.h>
#include <iomanip.h>
```

```
int main()
{int a[100];
 ...
 int i;
 for (i = 0; i <= n-2; i++)
 {int min = a[i];
  int k = i;
  for (int j = i+1; j <= n-1; j++)
    if (a[j] < min)
    {min = a[j];
     k = j;
    }
  int x = a[i]; a[i] = a[k]; a[k] = x;
 }
 for (i = 0; i <= n-1; i++)
   cout << setw(10) << a[i];
 cout << '\n';
 return 0;
}
```

Сливане на редици

Сливането е вид сортиране. Нека са дадени сортираните във възходящ ред редици:

a_0, a_1, …, a_{n-1}

b_0, b_1, …, b_{n-1}

Да се слеят редиците означава да се конструира нова, сортирана във възходящ ред редица, съставена от елементите на дадените редици. Осъществява се по следния начин:

- Поставят се "указатели" към първите елементи на редиците $\{a_i\}$ и $\{b_j\}$.

- Докато има елементи и в двете редици, се сравняват елементите, сочени от "указателите". По-малкият елемент се записва в новата редица, след което се прескача.

- След изчерпване на елементите на едната от дадените редици, елементите на другата от "указателя" (включително) се прехвърлят в новата редица.

Задача 54. Дадени са сортираните във възходящ ред редици:

$a_0, a_1, \ldots, a_{n-1}$

$b_0, b_1, \ldots, b_{m-1}$

($n \geq 1, m \geq 1$). Да се напише програма, която слива двете редици в редицата

$c_0, c_1, \ldots, c_{k-1}$.

Програма Zad54.cpp решава задачата.

Program Zad54.cpp

```
#include <iostream.h>
#include <iomanip.h>
int main()
{int a[20];
 cout << "n= ";
 int n;
 cin >> n;
 if (!cin)
 {cout << "Error, Bad input! \n";
  return 1;
 }
 if (n < 1 || n > 20)
 {cout << "Incorrect input! \n";
  return 1;
 }
 int i;
 for (i = 0; i <= n-1; i++)
 {cout << "a[" << i << "]= ";
  cin >> a[i];
 }
 int b[10];
 cout << "m= ";
 int m;
 cin >> m;
 if (!cin)
 {cout << "Error, Bad input! \n";
  return 1;
 }
 if (m < 1 || m > 10)
 {cout << "Incorrect input! \n";
```

```
 return 1;
}
for (i = 0; i <= m-1; i++)
{cout << "b[" << i << "]= ";
 cin >> b[i];
}
int p1 = 0, p2 = 0;
int c[30];
int p3 = -1;
while (p1 <= n-1 && p2 <= m-1)
if (a[p1] <= b[p2])
{p3++;
 c[p3] = a[p1];
 p1++;
}
else
{p3++;
c[p3] = b[p2];
p2++;
}
if (p1 > n-1)
   for (i = p2; i <= m-1; i++)
    {p3++;
     c[p3] = b[i];
    }
else
   for (i = p1; i <= n-1; i++)
    {p3++;
     c[p3] = a[i];
    } // извеждане на редицата
 for (i=0; i<=p3; i++)
   cout << setw(10) << c[i];
 cout << '\n';
 return 0;
}
```

Разглежданите досега масиви се наричат **едномерни.** Те реализират крайни редици от елементи от скаларен тип. Възможно е обаче типът

на елементите да е масив. В този случай се говори за многомерни масиви.

4. Многомерни масиви

Масив, базовият тип на който е едномерен масив, се нарича **двумерен**. Масив, базовият тип на който е двумерен масив, се нарича **тримерен** и т.н. На практика се използват масиви с размерност най-много 3.

Задаване на многомерни масиви

Нека T е име или дефиниция на произволен тип, различен от псевдоним, void и функционален, $size_1$, $size_2$, …, $size_n$ ($n>1$ е дадено цяло число) са константни изрази от интегрален или изброен тип с положителни стойности. T[$size_1$][$size_2$] … [$size_n$] е тип n-мерен масив от тип T. T се нарича базов тип за типа масив.

Примери:

int [5][3] е двумерен масив от тип int;

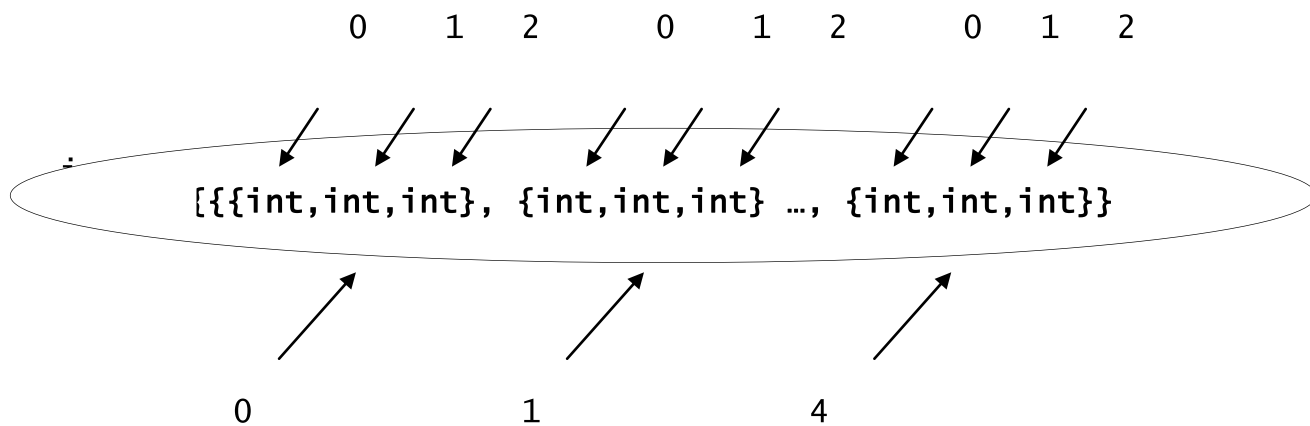
double [4][5][3] е тримерен масив от тип double;

Множество от стойности

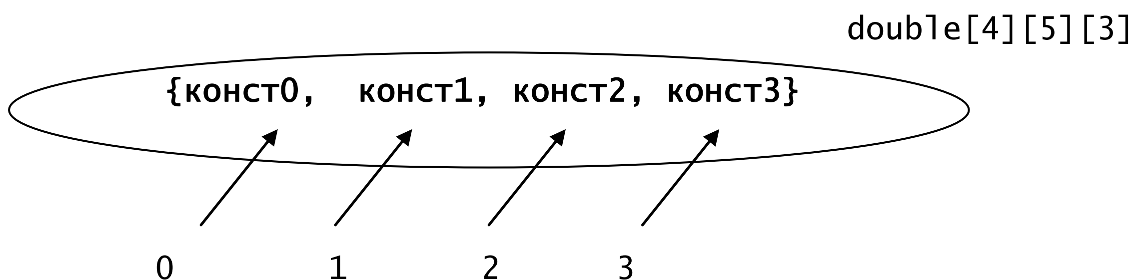
Множеството от стойности на типа T[$size_1$][$size_2$] … [$size_n$] се състои от всички редици от по $size_1$ елемента, които са произволни константи от тип T[$size_2$] … [$size_n$]. Достъпът до елементите на редиците е пряк и се осъществява с помощта на индекс, като достъпът до първия елемент се осъществява с индекс със стойност 0, до последния – с индекс със стойност $size_1$-1, а до всеки от останалите елементи – с индекс със стойност с 1 по-голяма от тази на индекса на предишния елемент. Елементите от множеството от стойности на даден тип многомерен масив са **константите** на този тип масив.

Примери:

1. Множеството от стойности на типа int[5][3] се състои от всички редици от по 5 елемента, които са едномерни масиви от тип int[3]. Достъпът до елементите на редиците се осъществява с индекс със стойности 0, 1, 2, 3 и 4.



2. Множеството от стойности на типа double[4][5][3] се състои от всички редици от по 4 константи от тип double[5][3]. Достъпът до елементите на редиците се осъществява с индекс със стойности 0, 1, 2 и 3.



където с $конст_i$ (i = 0, 1, 2, 3) е означена произволна константа от тип double[5][3].

Променлива величина, множеството от допустимите стойности на която съвпада с множеството от стойности на даден тип масив, се нарича променлива от дадения тип масив или само масив. Фиг. 3 дава обобщение на синтаксиса на дефиницията на променлива от тип масив.

```
   <дефиниция_на_променлива_от_тип_многомерен_масив> ::=
      T <променлива>[size1][size2] … [sizen]; |
      T <променлива>[size1][size2]…[sizen]
            = {<редица_от_константи_от_тип T1>};|
      T <променлива>[size1][size2]…[sizen]
            = {<редица_от_константи_от_тип T>};
```

където

T е име или дефиниция на произволен тип, различен от псевдоним, void и функционален;

T1 е име на типа T[$size_2$]…[$size_n$];

$size_1$, $size_2$, …$size_n$ са константни изрази от интегрален или изброен тип със *положителни* стойности;

<променлива> ::= <идентификатор>

<редица_от_константи_от_тип T1> ::= <константа_от_тип T1>|
<константа_от_тип T1>, <редица_от_константи_от_тип T1>

а <редица_от_константи_от_тип T> се определя по аналогичен начин.

Фиг. 3.

Примери:

```
int x[10][20];
double y[20][10][5];
int z[3][2] = {{1, 3},
               {5, 7},
               {2, 9}};
int t[2][3][2] = {{{1, 3}, {5, 7}, {6, 9}},
                  {{7, 8}, {1, 8}, {-1, -4}}};
```

Фрагментите

<променлива>[$size_1$][$size_2$] … [$size_n$],

<променлива>[$size_1$][$size_2$]…[$size_n$]={<редица_от_константи_от_тип T1>}

<променлива>[$size_1$][$size_2$]…[$size_n$]={<редица_от_константи_от_тип T>};

от дефиницията от Фиг. 3, могат да се повтарят. За разделител се използва символът запетая.

Примери:

```
int a[3][4], b[2][3][2] = {{{1, 2}, {3, 4}, {5, 6}},
                           {{7, 8}, {9, 0}, {1, 2}}};
double c[2][3] = {1, 2, 3, 4, 5, 6}, d[3][4][5][6];
```

При дефиницията с инициализация, от Фиг. 3., е възможно $size_1$ да се пропусне. Тогава за стойност на $size_1$ се подразбира броят на редиците от константи на най-външно ниво, изброени при инициализацията.

Пример: Дефиницията

```
int s[][2][3] =  {{{1,2,3}, {4, 5, 6}},
                  {{7, 8, 9}, {10, 11, 12}},
```

```
                     {{13, 14, 15}, {16, 17, 18}}};
```

е еквивалентна на

```
   int s[3][2][3] = {{{1,2,3}, {4, 5, 6}},
                     {{7, 8, 9}, {10, 11, 12}},
                     {{13, 14, 15}, {16, 17, 18}}};
```

Ако изброените константни изрази в инициализацията на ниво i са по-малко от $size_i$, останалите се инициализират с нулеви стойности.

Примери: Дефиницията

```
      int fi[5][6] =   {{1, 2}, {5}, {3, 4, 5},
                        {2, 3, 4, 5}, {2, 0, 4}};
```

е еквивалентна на

```
      int fi[5][6] =  { {1, 2, 0, 0, 0, 0},
                        {5, 0, 0, 0, 0, 0},
                        {3, 4, 5, 0, 0, 0},
                        {2, 3, 4, 5, 0, 0},
                        {2, 0, 4, 0, 0, 0}};
```

Вложените фигурни скобки не са задължителни. Следното инициализиране

```
   int ma[4][3] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};
```

е еквивалентно на

```
   int ma[4][3] = {{0, 1, 2}, {3, 4, 5}, {6, 7, 8}, {9, 10, 11}};
```

но е по-неясно.

Следващата дефиниция

```
   int ma[4][3] = {{0}, {1}, {2}, {3}};
```

е еквивалентна на

```
   int ma[4][3] = {{0, 0, 0 }, {1, 0, 0}, {2, 0, 0}, {3, 0, 0}};
```

и е различна от

```
   int ma[4][3] = {0, 1, 2, 3};
```

която пък е еквивалентна на

```
   int ma[4][3] = {{0, 1, 2}, {3, 0, 0}, {0, 0, 0}, {0, 0, 0}};
```

Инициализацията е един начин за свързване на променлива от тип масив с конкретна константа от множеството от стойности на този тип масив. Друг начин предоставят т.нар. индексирани променливи. С всяка променлива от тип масив е свързан набор от индексирани променливи. Фиг. 4. обобщава техния синтаксис.

```
   <индексирана_променлива> ::=
```

<променлива_от_тип_масив>[<индекс$_1$>][<индекс$_2$>][<индекс$_n$>]
където
<индекс$_i$> е *израз* от интегрален или изброен тип.
Всяка индексирана променлива е от базовия тип.

Фиг. 4.

Примери:

1. С променливата x, дефинирана по-горе, са свързани индексираните променливи

x[0][0], x[0][1], …, x[0][19],
x[1][0], x[1][1], …, x[1][19],
…
x[9][0], x[9][1], …, x[9][19],

които са от тип int.

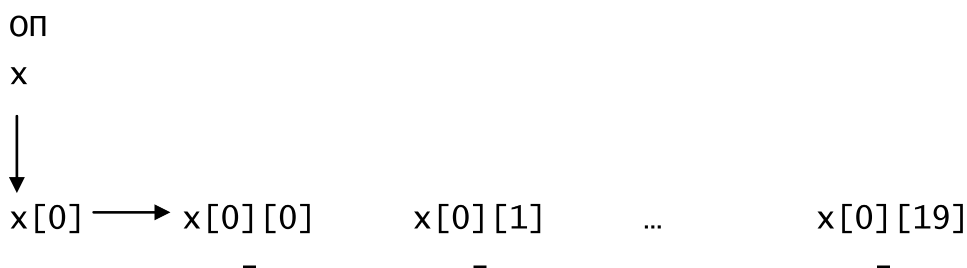
2. С променливата y са свързани следните реални индексирани променливи:

y[i][0][0], y[i][0][1], …, y[i][0][4],
y[i][1][0], y[i][1][1], …, y[i][1][4],
…
y[i][9][0], y[i][9][1], …, y[i][9][4],

за i = 0, 1, …, 19.

Дефиницията на променлива от тип многомерен масив не само свързва променливата с множеството от стойности на указания тип, но и отделя 4B памет, в която записва адреса на първата индексирана променлива на масива. Останалите индексирани променливи се разполагат последователно след първата по по-бързото нарастване на по-далечните си индекси. За всяка индексирана променлива се отделя толкова памет, колкото базовият тип изисква. Следващият пример илюстрира по-подробно представянето.

Пример:



x[1] ⟶ x[1][0] x[1][1] … x[1][19] …

x[9] ⟶ x[9][0] x[9][1] … x[9][19]

като за всяка индексирана променлива са отделени 4B ОП, които са с неопределено съдържание, тъй като x не е дефинирана с инициализация. Освен това, чрез индексираните променливи x[0], x[1], ..., x[9] могат да се намерят адресите на x[0][0], x[1][0], ..., x[9][0] съответно, т.е.

```
cout << x[0] << x[1] << ... << x[9];
```

ще изведе адресите на x[0][0], x[1][0], ... и x[9][0].

Забележка: Двумерните масиви разполагат в ОП индексираните си променливи по по-бързото нарастване на втория индекс. Това физическо представяне се нарича **представяне по редове**. Тези масиви могат да бъдат използвани за реализация и работа с матрици и др. правоъгълни таблици.

Важно допълнение: При работа с масиви трябва да се има предвид, че повечето реализации не проверяват дали стойностите на индексите са в рамките на границите, зададени при техните дефиниции. Тази особеност крие опасност от допускане на труднооткриваеми грешки.

Често допускана грешка: В Паскал, Ада и др. процедурни езици, индексите на индексираните променливи се ограждат само в една двойка квадратни скобки и се отделят със запетаи. По навик, при програмиране на C++, често се използва същото означение. Това е неправилно, но за съжаление не винаги е съпроводено със съобщение за грешка, тъй като в езика C++ съществуват т.нар. comma-изрази. Използвахме ги вече в заглавните части на оператора за цикъл for. Comma-изразите са изрази, отделени със запетаи. Стойността на най-десния израз е стойността на comma-израза. Операцията за последователно изпълнение запетая е лявоасоциативна. Така 1+3, 8, 21-15 е comma-израз със стойност 6, а [1, 2] е comma-израз със стойност [2]. в C++ ma[1,2] означава адреса на индексираната променлива ma[2][0] (индексът [0] се добавя автоматично).

Задача 55. Да се напише програма, която въвежда елементите на правоъгълна матрица a[nxм] и намира и извежда матрицата, получена от дадената като всеки от нейните елементи е увеличен с 1.

Програма Zad55.cpp решава задачата.

```
Program Zad55.cpp
#include <iostream.h>
#include <iomanip.h>
int main()
{int a[10][20];
 // въвеждане на броя на редовете на матрицата
 cout << "n= ";
 int n;
 cin >> n;
 if (!cin)
 {cout << "Error, Bad input! \n";
  return 1;
 }
 if (n < 1 || n > 10)
 {cout << "Incorrect input! \n";
  return 1;
 }
// въвеждане на броя на стълбовете на матрицата
 cout << "m= ";
 int m;
 cin >> m;
 if (!cin)
 {cout << "Error, Bad input! \n";
  return 1;
 }
 if (m < 1 || m > 20)
 {cout << "Incorrect input! \n";
  return 1;
 }
 // въвеждане на матрицата по редове
 int i, j;
```

```
    for (i = 0; i <= n-1; i++)
       for (j = 0; j <= m-1; j++)
       {cout << "a[" << i << ", " << j << "]= ";
        cin >> a[i][j];
        if (!cin)
        {cout << "Error, Bad Input! \n";
         return 1;
        }
       }
    // конструиране на нова матрица b
    int b[10][20];
    for (i = 0; i <= n-1; i++)
       for (j = 0; j <= m-1; j++)
          b[i][j] = a[i][j] + 1;
    // извеждане на матрицата b по редове
    for (i = 0; i <= n-1; i++)
    {for (j = 0; j <= m-1; j++)
      cout << setw(6) << b[i][j];
      cout << '\n';
    }
    return 0;
   }
```

Забележка: За реализиране на операциите извеждане и конструиране се извърши обхождане на елементите на двумерен масив по редове.

Задача 56. Да се напише програма, която намира и извежда сумата от елементите на всеки стълб на квадратната матрица a[nxn].

Програма Zad56.cpp решава задачата. В нея въвеждането на матрицата и нейната размерност са пропуснати, тъй като са аналогични на тези от Zad55.cpp.

Program Zad56.cpp

```
#include <iostream.h>
#include <iomanip.h>
int main()
{int a[10][10];
 int n;
 …
```

```
 int i, j;
 for (j = 0; j <= n-1; j++)
 {int s = 0;
  for (i = 0; i <= n-1; i++)
    s += a[i][j];  // s = s + a[I][j]
  cout << setw(10) << j << setw(10) << s << "\n";
 }
 return 0;
}
```

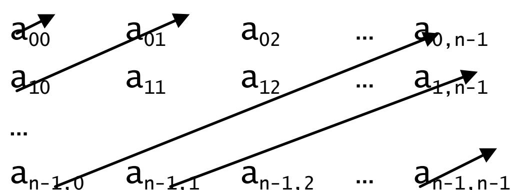
Реализирано е обхождане на масива по стълбове (първият индекс се изменя по-бързо).

Задача 57. Да се напише програмен фрагмент, който намира номерата на редовете на целочислената квадратна матрица a[nxn], в които има елемент, равен на цялото число x.

```
 …
 for (i = 0; i <= n-1; i++)
 {j = -1;
  do
  j++;
  while (a[i][j] != x && j < n-1);
  if (a[i][j] == x) cout << setw(5) << i << '\n';
  }
 …
```

Фрагментът реализира последователно обхождане на *всички* редове на матрицата и за всеки ред проверява дали *съществува* елемент, равен на дадения елемент x.

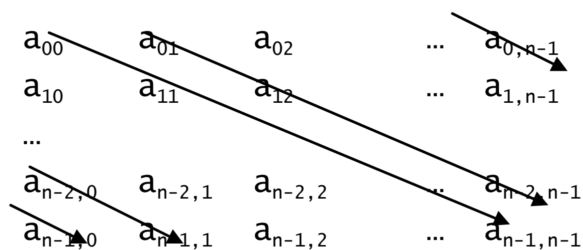
Задача 58. Да се напише програмен фрагмент, който обхожда кватратната матрица a[nxn] по диагонали, започвайки от елемента a_{00}, както е показано по-долу:




```
 …
 int k;
 for (k = 0; k <= n-1; k++)
```

```
{for (i = k; i >= 0; i--)
   cout << "(" << i << ", "<< k-i << ") ";
 cout << '\n';
}
for (k = n; k <= 2*n-2; k++)
{for (i = n-1; i >= k-n+1; i--)
   cout << "(" << i << ", "<< k-i << ") ";
 cout << '\n';
}
…
```

Задача 59. Да се напише програмен фрагмент, който обхожда кватратната матрица a[nxn] по диагонали, започвайки от елемента $a_{n-1,0}$, както е показано по-долу:




```
…
int k;
for (k = n-1; k >= 0; k--)
{for (i = k; i <= n-1; i++)
   cout << "(" << i << ", "<< i-k << ") ";
 cout << '\n';
}

for (k = -1; k >= 1-n; k--)
{for (i = 0; i <= n+k-1; i++)
   cout << "(" << i << ", "<< i-k << ") ";
 cout << '\n';
}
…
```

Задача 60. Да се напише програма, която:

а) въвежда по редове елементите на квадратната реална матрица A с размерност n x n;

б) от матрицата A конструира редицата B: b_0, b_2,..., b_{m-1}, където m = n.n, при което първите n елемента на B съвпадат с елементите на първия стълб на A, вторите n елемента на B съвпадат с елементите на втория стълб на A и т.н., последните n елемента на B съвпадат с елементите на последния стълб на A;

в) сортира във възходящ ред елементите на редицата B;

г) образува нова квадратна матрица A с размерност n x n, като елементите от първия ред на A съвпадат с първите n елемента на B, елементите от втория ред на A съвпадат с вторите n елемента на B и т.н. елементите от n - тия ред на A съвпадат с последните n елемента на B;

д) извежда по редове новата матрица A.

Програма Zad60.cpp решава задачата.

Program Zad60.cpp

```
#include <iostream.h>
   #include <iomanip.h>
   int main()
   {int a[10][10];
    cout << "n= ";
    int n;
    cin >> n;
    if (!cin)
    {cout << "Error, Bad input! \n";
     return 1;
    }
    if (n < 1 || n > 10)
    {cout << "Incorrect input! \n";
     return 1;
    }
    // въвеждане на масива а
    int i, j;
    for (i = 0; i <= n-1; i++)
        for (j = 0; j <= n-1; j++)
       {cout << "a[" << i<< "][" << j  << "]= ";
        cin >> a[i][j];
       }
    // извеждане на елементите на а по редове
```

```
for (i = 0; i <= n-1; i++)
{for (j = 0; j <= n-1; j++)
  cout << setw(5) << a[i][j];
 cout << "\n";
}
// развиване на матрицата а по стълбове
int b[100];
int m = -1;
for (j = 0; j <= n-1; j++)
   for (i = 0; i <= n-1; i++)
   {m++;
    b[m] = a[i][j];
   }
   m++; // m е броя на елементите на редицата b
// извеждане на редицата b
for (i = 0; i <= m-1; i++)
   cout << setw(5) << b[i];
cout << '\n';
// сортиране на b по метода на пряката селекция
for (i = 0; i <= m-2; i++)
{int k = i;
 int min = b[i];
 for (j = i+1; j <= m-1; j++)
   if (b[j] < min)
   {min = b[j];
    k = j;
   }
 int x = b[i]; b[i] = b[k]; b[k] = x;
}
// извеждане на сортираната b
for (i = 0; i <= m-1; i++)
   cout << setw(5) << b[i];
 cout << '\n';
// конструиране на новата матрица а
m = -1;
for (i = 0; i <= n-1; i++)
   for (j = 0; j <= n-1; j++)
   {m++;
   a[i][j] = b[m];
```

```
      }
   // извеждане на матрицата a
   for (i = 0; i <= n-1; i++)
    {for (j = 0; j <= n-1; j++)
         cout << setw(10) << a[i][j];
     cout << '\n';
 }
 return 0;
}
```

5. Символни низове

Структура от данни низ

Логическо описание

Редица от краен брой символи, заградени в кавички, се нарича **символен низ** или само **низ.**

Броят на символите в редицата се нарича **дължина** на низа.

Примери: “xyz” е символен низ с дължина 3,
“This is a string.” е символен низ с дължина 17, а
“” е символен низ с дължина 0. Нарича се **празен низ.**

Низ, който се съдържа в даден низ се нарича негов **подниз.**

Пример: Низът “ is s “ е подниз на низа “This is a string.”, а низът “ is a sing” не е негов подниз.

Конкатенация на два низа е низ, получен като в края на първия низ се запише вторият. Нарича се още **слепване** на низове.

Пример: Конкатенацията на низовете “a+b” и “=b+a” е низът “a+b=b+a”, а конкатенацията на “=b+a” с “a+b” е низът “=b+aa+b”. Забелязваме, че редът на аргументите е от значение.

Два символни низа се сравняват по следния начин: Сравнява се всеки символ от първия низ със символа от съответната позиция на втория низ. Сравнението продължава до намиране на два различни символа или до края на поне един от символните низове. Ако кодът на символ от първия низ е по-малък от кода на съответния символ от втория низ, или първият низ е изчерпен, приема се, че първият низ е по-малък от втория. Ако пък е по-голям или вторият низ е изчерпен – приема се, че първият низ е по-голям от втория. Ако в процеса на

сравнение и двата низа едновременно са изчерпени, те са развни. Това сравнение се нарича **лексикографско**.

Примери: “abbc” е равен на “abbc”
“abbc” е по-малък от “abbcaaa”
“abbc” е по-голям от “aa”
“abbcc” е по-голям от “abbc”.

Физическо представяне

В ОП низовете се представят последователно.

Символни низове в езика C++

Съществуват два начина за разглеждане на низовете в езика C++:

- като масиви от символи и
- като указатели към тип char.

За щастие, те са семантично еквивалентни.

В тази част ще разгледаме символните низове като масиви от символи.

Дефиницията

```
char str1[100];
```

определя променливата str1 за масив от 100 символа, а

```
char str2[5] = {‘a’, ‘b’, ‘c’};
```

дефинира масива от символи str2 и го инициализира. Тъй като при инициализацията са указани по-малко от 5 символа, останалите се допълват с нулевия символ, който се означава със символа \0, а понякога и само с 0. Така последната дефиниция е еквивалентна на дефиниците:

```
char str2[5] = ‘a’, ‘b’, ‘c’, ‘\0’, ‘\0’};
char str2[5] = ‘a’, ‘b’, ‘c’, 0, 0};
```

Всички действия, които описахме, за работа с едномерни масиви, са валидни и за масиви от символи с изключение на извеждането. Операторът

```
cout << str2;
```

няма да изведе адреса на str2 (както беше при масивите от друг тип), а текста

```
abc
```

Има обаче една особеност. Ако инициализацията на променливата str2 е пълна и не завършва със символа \0, т.е. има вида

```
char str2[5] = {‘a’, ‘b’, ‘c’, ‘d’, ‘e’};
```

операторът

```
cout << str2;
```

извежда текста

```
abcde<неопределено>
```

Имайки пред вид всичко това, бихме могли да напишем подходящи програмни фрагменти, които въвеждат, извеждат, копират, сравняват, извличат части, конкатенират низове. Тъй като операциите се извършват над индексираните променливи, налага се да се поддържа целочислена променлива, съдържаща дължината на низа.

В езика са дадени средства, реализиращи низа като скаларна структура. За целта низът се разглежда като редица от символи, завършваща с нулевия символ \0, наречен още **знак за край на низ.** Тази организация има предимството, че не е необходимо с всеки низ да се пази в променлива дължината му, тъй като знакът за край на низ позволява да се определи краят му.

Примери: Дефинициите

```
char m[5] = {‘a, ‘b’, ‘b’, ‘a’, ‘\0’};
char n[10] = {‘x’, ‘y’, ‘z’, ‘1’, ‘2’, ‘+’, ‘\0’};
```

свързват променливте от тип масив от символи m и n с низовете “abba” и “xyz12+” съответно. Знакът за край на низ ‘\0’ не се включва явно в низа.

Този начин за инициализация не е много удобен. Следните дефиниции са еквивалентни на горните.

```
char m[5] = “abba”;
char n[10] = “xyz12+”;
```

Забелязваме, че ако низ съдържащ n символа трябва да се свърже с масив от символи, минималната дължина на масива трябва да бъде n+1, за да се поберат n-те символа плюс символът \0.

Задаване на низове

Типът char[size], където size е константен израз от интегрален или изброен тип, може да бъде използван за задаване на тип низ с максимална дължина size-1.

Пример:

char[5]   може да се използва за задаване на тип низ с максимална дължина 4.

Множество от стойности

Множеството от стойности на типа низ, зададен чрез char[size], се състои от всички низове с дължина 0, 1, 2, ..., size-1

Примери:

1. Множеството от стойности на типа низ, зададен чрез char[5] се състои от всички низове с дължина 0, 1, 2, 3 и 4.

множество от стойности
на тип низ, зададен чрез char[5]

“” “a” “klm” “+)”
“1234” “vsa” “abba” “lkI”

2. Множеството от стойности на типа char[10] се състои от всички низове с дължина 0, 1, 2, ..., 9.

множество от стойности
на тип низ, зададен чрез char[10]

“” “1” “asd” “zxcvbnmop” “\n”
“a+b-c*k^” “a+b=?” “123+23=”

Елементите от множеството от стойности на даден тип низ са неговите **константи.** Например, “a+b=c-a*e” е константа от тип char[10].

Променлива величина, множеството от допустимите стойности на която съвпада с множеството от стойности на даден тип низ, се нарича променлива от този тип низ. Понякога ще я наричаме само низ.

Фиг. 5 определя дефиницията на променлива от тип низ.

```
<дефиниция_на_променлива_от_тип_низ> ::=
   char <променлива>[size]; |
   char <променлива>[size] = “<редица_от_символи>”;
   char <променлива>[size] = {<редица_от_константни_изрази>};|
```

където

<променлива> ::= <идентификатор>

size е константен израз от интегрален или изброен тип със *положителна* стойност;

<редица_от_константни_изрази> ::= <константен_израз>|
<константен_израз>, <редица_от_константни_изрази>

като константните изрази са от тип char.

<редица_от_символи> ::= <празно>| <символ> |
<символ><редица_от_символи>

с максимална дължина size-1.

Фиг. 5.

Примери:

```
char s1[5];
char s2[10] = “x+y”;
char s3[8] = {‘1’, ‘2’, ‘3’, ‘\0’};
```

Ако редицата от константни изрази съдържа по-малко от size израза, може да не завършва със знака за край на низ. Системата автоматично го добавя. А ако съдържа точно size константни израза, задължително трябва да завършва със знака за край на низ \0, или само 0.

При дефиниция на низ с инициализация е възможно size да се пропусне. Тогава инициализацията трябва да съдържа символа ‘\0’ и за стойност на size се подразбира броят на константните изрази, изброени при инициализацията, включително ‘\0’. Ако size е указано и изброените константни изрази в инициализацията са по-малко от size, останалите се инициализират с ‘\0’.

Примери:

Дефиницията

```
char q[5] =  {‘a’, ‘b’};
```

е еквивалентна на

```
char q[5] =  {‘a’, ‘b’, ‘\0’, ‘\0’, ‘\0’};
```

и на

```
char q[5] = “ab”;
```

а

```
char r[] = {'a', 'b', '\0'}; или
char r[] = "ab";
```

са еквивалентни на

```
char r[3] = {'a', 'b', '\0'}; или
char r[3] = "ab";
```

Забележка: Не са възможни конструкции от вида:

```
char q[5];
q = {'a', 'v', 's'}; или
char r[5];
r = "avs";
```

т.е. **на променлива от тип низ не може да бъде присвоявана константа от тип низ.**

Недопустими са също дефиниции от вида:

```
char q[4] = {'a', 's', 'd', 'f', 'g', 'h'}; или
char q[];
```

Инициализацията е един начин за свързване на променлива от тип низ с конкретна константа от множеството от стойности на този тип низ. Друг начин предоставят индексираните променливи.

Примери:

```
q[0] = 'a'; q[1] = 's'; q[2] = 'd';
```

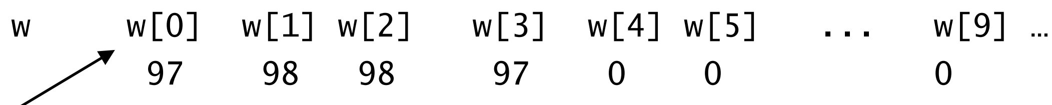
Дефиницията на променлива от тип низ не само свързва променливата с множеството от стойности на указания тип, но и отделя определено количество памет (обикновено 4B), в която записва адреса на първата индексирана променлива, свързана с променливата от тип низ. Останалите индексирани променливи се разполагат последователно след първата. За всяка индексирана променлива се отделя по 1B ОП. Съдържанието на отделената за индексираните променливи памет е неопределено освен ако не е зададена дефиниция с инициализация. Тогава в клетките се записват инициализиращите стойности, допълнени със знака за край на низ.

Пример: След дефиницията

```
char s[4];
char w[10] = "abba";
```

разпределението на паметта има вида:

```
ОП
s  ──►  s[0]  s[1]  s[2]  s[3]
         -     -     -     -
```



Операции и вградени функции

Въвеждане на стойност

Реализира се по стандартния начин - чрез оператора cin.

Пример:

```
char s[5], t[3];
cin >> s >> t;
```

Настъпва пауза в очакване да се въведат два низа с дължина **не по-голяма** от 4 в първия и не по-голяма от 2 във втория случай. Водещите интервали, табулации и преминаване на нов ред се пренебрегват. За разделител на низовете се използват интервалът, табулациите и знака за преминаване на нов ред. Знакът за край на низ автоматично се добавя в края на всяка от въведените знакови комбинации. При въвеждане на низовете не се извършва проверка за достигане на указаната горна граница. Това може да доведе до трудноооткриваеми грешки.

Извеждане на низ

Реализира се също по стандарния начин.

Пример:

Операторът

```
cout << s;
```

извежда низа, който е стойност на s. Не е нужно да се грижим за дължината му. Знакът за край на низ идентифицира края на му.

Дължина на низ

Намира се чрез функцията strlen.

Синтаксис

```
strlen(<str>)
```

където

<str> е произволен низ.

Семантика

Намира дължината на <str>.

Пример:

strlen(“abc”) намира 3, strlen(“”) намира 0.

За използване на тази функция е необходимо да се включи заглавният файл string.h.

Конкатенация на низове

Реализира се чрез функцията strcat.

Синтаксис

strcat(<var_str>, <str>)

където

<var_str> е променлива от тип низ, а

<str> е низ (константа, променлива или по-общо израз).

Семантика

Конкатенира низа, който е стойност на <var_str> с низа <str>. Резултатът от конкатенацията се връща от функцията, а също се съдържа в променливата <var_str>. За използване на тази функция е необходимо да се включи заглавният файл string.h.

Пример:

```
#include <iostream.h>
#include <string.h>
int main()
{char a[10];
 cout << "a= ";
 cin >> a;     // въвеждане на стойност на a
 char b[4];
 cout << "b= ";
 cin >> b;     // въвеждане на стойност на b
 strcat(a, b);    // конкатениране на a и b, резултатът е в a
 cout << a << ‘\n’;  // извеждане на a
 cout << strlen(strcat(a, b)) << '\n'; //повторна конкатенация
 return 0;
}
```

Забележка: Функцията strcat може да се използва и като оператор, и като израз.

Сравняване на низове

Реализира се чрез функцията strcmp.

Синтаксис

strcmp(<str1>, <str2>)

където

<str1> и <str2> са низове (константи, променливи или по-общо изрази).

Семантика

Низовете <str1> и <str2> се сравняват лексикографски. Функцията strcmp е целочислена. Резултатът от обръщение към нея е цяло число с отрицателна стойност (-1 за реализацията Visual C++ 6.0), ако <str1> е по-малък от <str2>, 0 – ако <str1> е равен на <str2> и с положителна стойност (1 за реализацията Visual C++ 6.0), ако <str1> е по-голям от <str2>.

За използване на strcmp е необходимо да се включи заглавният файл string.h.

Примери:

1.
```
char a[10] = "qwerty", b[15] = "qwerty";
if (!strcmp(a, b)) cout << "yes \n"; else cout << "no \n";
```
извежда yes, тъй като strcpm(a, b) връща 0 (низовете са равни), !strcmp(a, b) е 1 (true).

2.
```
char a[10] = "qwe", b[15] = "qwerty";
if (strcmp(a, b)) cout << "yes \n"; else cout << "no \n";
```
извежда yes, тъй като strcpm(a, b) връща -1 (a е по-малък от b).

3.
```
char a[10] = "qwerty", b[15] = "qwer";
if (strcmp(a, b)) cout << "yes \n"; else cout << "no \n";
```
извежда yes, тъй като strcpm(a, b) връща 1 (a е по-голям от b).

Копиране на низ

Реализира се чрез функцията strcpy.

Синтаксис

strcpy(<var_str>, <str>)

където

<var_str> е променлива от тип низ, а

<str> е низ (константа, променлива или по-общо израз).

Семантика

Копира <str1> в <var_str>. Ако <str1> е по-дълъг от допустимата за <val_str> дължина, са възможни трудноооткриваеми грешки. Резултатът от копирането се връща от функцията, а също се съдържа в <var_str>.

За използване на тази функция е необходимо да се включи заглавният файл string.h.

Пример: Програмният фрагмент

```
char a[10];
strcpy(a, "1234567");
cout << a << "\n";
```

извежда

```
1234567
```

Търсене на низ в друг низ

Реализира се чрез функцията strstr.

Синтаксис

strstr(<str1>, <str2>)

където

<str1> и <str2> са произволни низове (константи, променливи или по-общо изрази).

Семантика

Търси <str2> в <str1>. Ако <str2> се съдържа в <str1>, strstr връща подниза на <str1> започващ от първото срещане на <str2> до края на <str1>. Ако <str2> не се съдържа в <str1>, strstr връща "нулев указател". Последното означава, че в позиция на условие, функционалното обръщение ще има стойност false, но при опит за извеждане, ще предизвиква грешка.

За използване на тази функция е необходимо да се включи заглавният файл string.h.

Примери: Програмният фрагмент

```
char str1[15] = "asemadaemada", str2[10]= "ema";
cout << strstr(str1, str2) << "\n";
```

извежда

```
emadaemada
```

а

```
char str1[15] = "asemadaemada", str2[10]= "ema";
cout << strstr(str2, str1) << "\n";
```

предизвиква съобщение за грешка по време на изпълнение.

Преобразуване на низ в цяло число

Реализира се чрез функцията atoi.

Синтаксис

atoi(<str>)

където <str> е произволен низ (константа, променлива или по-общо израз от тип низ).

Семантика

Преобразува символния низ в число от тип int. Водещите интервали, табулации и знака за преминаване на нов ред се пренебрегват. Символният низ се сканира до първия символ различен от цифра. Ако низът започва със символ различен от цифра и знак, функцията връща 0.

За използване на тази функция е необходимо да се включи заглавният файл stdlib.h.

Примери:

Програмният фрагмент

```
char s[15] = "-123a45";
cout << atoi(s) << "\n";
```

извежда –123, а

```
char s[15] = "b123a45";
cout << atoi(s) << "\n";
```

извежда 0.

Преобразуване на низ в реално число

Реализира се чрез функцията atof.

Синтаксис

atof(<str>)

където <str> е произволен низ (константа, променлива или по-общо израз от тип низ).

Семантика

Преобразува символния низ в число от тип double. Водещите интервали, табулации и знака за преминаване на нов ред се пренебрегват. Символният низ се сканира до първия символ различен

от цифра. Ако низът започва със символ различен от цифра, знак или точка, функцията връща 0.

За използване на тази функция е необходимо да се включи заглавният файл stdlib.h.

Примери:

Програмният фрагмент

```
char s[15] = "-123.35a45";
cout << atof(st) << "\n";
```

извежда –123.35, а

```
char st[15] = ".123.34c35a45";
cout << atof(st) << "\n";
```

извежда 0.123.

Допълнение:

Конкатенация на n символа от низ с друг низ

Реализира се чрез функцията strncat.

Синтаксис

```
strncat(<var_str>, <str>, n)
```

където

<var_str> е променлива от тип низ,

<str> е низ (константа, променлива или по-общо израз), а

n е цял израз с *неотрицателна* стойност.

Семантика

Копира първите n символа от <str> в края на низа, който е стойност на <var_str>. Копирането завършва когато са прехвърлени n символа, или е достигнат краят на <str>. Резултатът е в променливата <var_str>. За използване на тази функция е необходимо да се включи заглавният файл string.h.

Пример: Резултатът от изпълнението на фрагмента:

```
chat a[10] = “aaaaa”;
strncat(a, “qwertyqwerty”, 5);
cout << a;
```

е

```
aaaaaqwert
```

а на

```
strncat(a, “qwertyqwerty”, -5);
cout << a;
```
предизвиква съобщение за грешка.

Копиране на n символа в символен низ

Реализира се чрез функцията strncpy.

Синтаксис

```
strncpy(<var_str>, <str>, n)
```

където

<var_str> е променлива от тип низ,

<str> е низ (константа, променлива или по-общо израз), а

n е цял израз с *неотрицателна* стойност.

Семантика

Копира първите n символа на <str1> в <var_str>. Ако <str> има по-малко от n символа, ‘\0’ се копира до тогава докато не се запишат n символа. Параметърът <var_str> трябва да е от вида char[n] и съдържа резултатния низ. За използване на тази функция е необходимо да се включи заглавният файл string.h.

Примери: 1. Програмният фрагмент

```
char a[10];
strncpy(a, "1234567", 8);
cout << a << "\n";
```
извежда

```
1234567
```
Изпълнява се по следния начин: тъй като дължината на низа “1234567” е по-малка от 8, допълва се с един знак ‘\0’ и се свързва с променливата a.

2. Програмният фрагмент

```
char a[10];
strncpy(a, "123456789", 5);
cout << a << "\n";
```
извежда

```
12345<неопределено>
```
Изпълнява се по следния начин: тъй като дължината на низа “123456789” е по-голяма от 5, низът “12345” се свързва с променливата a, но не става допълване с ‘\0’, което личи по резултата.

Сравняване на n символа на низове

Реализира се чрез функцията strncmp.

Синтаксис

strncmp(<str1>, <str2>, n)

където

<str1> и <str2> са низове (константи, променливи или по-общо изрази), а

n е цял израз с *неотрицателна* стойност.

Семантика

Сравнява първите n символа на <str1> със символите от съответната позиция на <str2>. Сравнението продължава до намиране на два различни символа или до края на един от символните низове.

Резултатът от функцията strncmp е цяло число с отрицателна стойност, ако <str1> е по-малък от <str2>, 0 – ако <str1> е равен на <str2> и с положителна стойност, ако <str1> е по-голям от <str2>.

За използване на strncmp е необходимо да се включи заглавният файл string.h.

Примери:

1. char a[10] = “qwer”, b[15] = “qwerty”;
   if (!strncmp(a, b, 3)) cout << “yes \n”;
   else cout << “no \n”;

извежда yes, тъй като strncpm(a, b) връща 0 (низовете са равни), !strncmp(a, b) е 1 (true).

2. char a[10] = “qwer”, b[15] = “qwerty”;
   if (strncmp(a, b, 5)) cout << “yes \n”;
   else cout << “no \n”;

извежда yes, тъй като strncpm(a, b) връща -1 (a е по-малък от b).

3. char a[10] = “qwerty”, b[15] = “qwer”;
   if (strncmp(a, b, 5)) cout << “yes \n”;
   else cout << “no \n”;

извежда yes, тъй като strncpm(a, b) връща 1 (a е по-голям от b).

Търсене на символ в низ

Реализира се чрез функцията strchr, съдържаща се в string.h.

Синтаксис

strchr(<str>, <expr>)

където

<str> е произволен низ, а

<expr> е израз от интегрален или изброен тип с положителна стойност, означаваща ASCII код на символ.

Семантика

Търси първото срещане на символа, чийто ASCII код е равен на стойността на <expr>. Ако символът се среща, функцията връща подниза на <str> започващ от първото срещане на символа и продължаващ до края му. Ако символът не се среща – връща "нулев указател". "нулев указател". Последното означава, че в позиция на условие, функционялното обръщение ще има стойност false, но при опит за извеждане, ще предизвика грешка.

Примери: Операторът

cout << strchr("qwerty", 'e');

извежда

erty

Операторът

cout << strchr("qwerty", 'p');

извежда съобщение за грешка, а

if (strchr("qwerty", 'p')) cout << "yes \n"; else cout << "no \n";

извежда no, тъй като 'p' не се среща в низа "qwerty".

Търсене на първата разлика

Реализира се чрез функцията strspn.

Синтаксис

strspn(<str1>, <str2>)

където

<str1> и <str2> са произволни низове (константи, променливи или по-общо изрази).

Семантика

Проверява до коя позиция <str1> и <str2> съвпадат. Връща дължината на префикса до първия различен символ. За използване на тази функция е необходимо да се включи заглавният файл string.h.

Пример: Програмният фрагмент

char a[10]= "asdndf", b[15] = "asdsdfdhf";

```
cout << strspn(a, b) << "\n";
```

извежда 3 тъй като първият символ, по който се различават a и b е в позиция 4.

Задачи върху тип низ

Задача 61.

Задачи

Задача 1. Да се напише програма, която намира скаларното произведение на реалните вектори $a = (a_0, a_1, ..., a_{n-1})$ и $b = (b_0, b_1, ..., b_{n-1})$, $(1 \leq n \leq 50)$.

Задача 2. Да се напише програма, която въвежда n символа и намира и извежда минималния (максималния) от тях.

Задача 3. Да се напише програма, която:

а) въвежда редицата от n цели числа $a_0, a_1, ..., a_{n-1}$,

б) намира и извежда сумата на тези елементи на редицата, които се явяват удвоени нечетни числа.

Задача 4. Да се напише програма, която намира и извежда сумата от положителните и броя на отрицателните елементи на редицата от реални числа $a_0, a_1, ..., a_{n-1}$ $(1 \leq n \leq 30)$.

Задача 5. Да се напише програма, която изчислява реципрочното число на произведението на тези елементи на редицата $a_0, a_1, ..., a_{n-1}$ $(1 \leq n \leq 50)$, за които е в сила релацията $2 < ai < i!$.

Задача 6. Да се напише програма, която изяснява, има ли в редицата от цели числа $a_0, a_1, ..., a_{n-1}$ $(1 \leq n \leq 100)$ два последователни нулеви елемента.

Задача 7. Дадени са сортираните във възходящ ред числови редици $a_0, a_1, ..., a_{k-1}$ и $b_0, b_1, ..., b_{k-1}$ $(1 \leq k \leq 40)$. Да се напише програма, която намира броя на равенствата от вида $a_i = b_j$ $(i = 0, ..., k-1, j = 0, ..., k-1)$.

Задача 8. Дадени са две редици от числа. Да се напише програма, която определя колко пъти първата редица се съдържа във втората.

Задача 9. Всяка редица от равни числа в едномерен сортиран масив, се нарича площадка. Да се напише програма, която намира началото и дължината на най-дългата площадка в даден сортиран във възходящ ред едномерен масив.

Задача 10. Да се напише програма, която по дадена числова редица a_0, a_1, ..., a_{n-1} ($1 \le n \le 20$) намира дължината на максималната ѝ ненамаляваща подредица a_{i1}, a_{i2}, ..., $a_{iк}$ ($a_{i1} <= a_{i2} <= ... <= a_{iк}$, $i1 < i2 < ... < iк$).

Задача 11. Дадена е редицата от символи s_0, s_1, ..., s_{n-1} ($1 \le n \le 15$). Да се напише програма, която извежда отначало всички символи, които изобразяват цифри, след това всички символи, които изобразяват малки латински букви и накрая всички останали символи от редицата, запазвайки реда им в редицата.

Задача 12. Дадени са две цели числа, представени с масиви от символи. Да се напише програма, която установява дали първото от двете числа е по-голямо от второто.

Задача 13. Да се напише програма, която определя дали редицата от символи s_0, s_1, ..., s_{n-1} ($1 \le n \le 45$) е симетрична, т.е. четена отляво надясно и отдясно наляво е една и съща.

Задача 14. Дадена е редицата от естествени числа a_0, a_1, ..., a_{n-1} ($1 \le n \le 30$). Да се напише програма, която установява има ли сред елементите на редицата не по-малко от два елемента, които са степени на 2.

Задача 15. Дадени са полиномите Pn(x) и Qm(x). Да се напише програма, която намира:

а) сумата им

б) произведението им.

Задача 16. За векторите a = (a_0, a_1, ..., a_{n-1}) и b = (b_0, b_1, ..., b_{n-1}) ($1 \le n \le 20$), да се определи дали са линейно зависими.

Задача 17. Дадена е квадратна целочислена матрица A с размерност nxn, елементите на която са естествени числа. Да се напише програма, която намира:

а) сумата от елементите под главния диагонал, които са прости числа;

б) произведението от елементите над главния диагонал, в записа на цифрите на които се среща цифрата 5;

в) номера на първия неотрицателен елемент върху главния диагонал.

Задача 18. Дадена е квадратната реална матрица A с размерност nxn. Да се напише програма, която намира:

а) сумата от елементите върху вторичния главен диагонал;

б) произведението от елементите под (над) вторичния главен диагонал.

Задача 19. Дадена е реална правоъгълна матрица A с размерност nxm. Да се напише програма, която изтрива k-ти ред (стълб) на A. Изтриването означава да се преместят редовете (стълбовете) с един нагоре (наляво) и намаляване броя на редовете (стълбовете) с един.

Задача 20. Върху равнина са дадени n точки чрез матрицата

$$X = \begin{pmatrix} x_{0,0} & x_{0,1} & \dots & x_{0,n-1} \\ x_{1,0} & x_{1,1} & \dots & x_{1,n-1} \end{pmatrix}$$

така, че $(x_{0,i}, x_{1,i})$ са координатите на i-тата точка. Точките по двойки са съединени с отсечки. Да се напише програма, която намира дължината на най-дългата отсечка.

Задача 21. Дадена е матрицата от цели числа

$$A = \begin{pmatrix} a_{0,0} & a_{0,1} & \dots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \dots & a_{1,n-1} \end{pmatrix}$$

Да се напише програма, която намира сумата на тези елементи $a_{1,i}$, $(0 \le i \le n-1)$, за които $a_{0,i}$ имат стойността на най-голямото сред елементите от първия ред на матрицата.

Задача 22. Дадено е множеството M от двойки

$M = \{<x_0, y_0>, <x_1, y_1>, \dots, <x_{n-1}, y_{n-1}>\}$,

като x_i и y_i $(0 \le i \le n-1)$ са цели числа. Да се напише програма, която проверява дали множеството M дефинира функция.

Упътване: Множеството M дефинира функция, ако от $x_i = x_j$ следва $y_i = y_j$.

Задача 23. Да се напишат програми, която конструират матриците:

$$\begin{pmatrix} 0&0&0&0&0&0&0&0&0&0 \\ 0&1&0&0&0&0&0&0&0&0 \\ 0&0&2&0&0&0&0&0&0&0 \\ 0&0&0&3&0&0&0&0&0&0 \\ 0&0&0&0&4&0&0&0&0&0 \\ 0&0&0&0&0&5&0&0&0&0 \\ 0&0&0&0&0&0&6&0&0&0 \\ 0&0&0&0&0&0&0&7&0&0 \\ 0&0&0&0&0&0&0&0&8&0 \\ 0&0&0&0&0&0&0&0&0&9 \end{pmatrix} \quad \begin{pmatrix} 1&2&3&4&5&6&7&8&9&10 \\ 0&1&2&3&4&5&6&7&8&9 \\ 0&0&1&2&3&4&5&6&7&8 \\ 0&0&0&1&2&3&4&5&6&7 \\ 0&0&0&0&1&2&3&4&5&6 \\ 0&0&0&0&0&1&2&3&4&5 \\ 0&0&0&0&0&0&1&2&3&4 \\ 0&0&0&0&0&0&0&1&2&3 \\ 0&0&0&0&0&0&0&0&1&2 \\ 0&0&0&0&0&0&0&0&0&1 \end{pmatrix}$$

Задача 24. Дадена е целочислената квадратна матрица A от n-ти ред. Да се напише програма, която намира максималното от простите числа на A.

Задача 25. Казваме, че два стълба на една матрица си приличат, ако съвпадат множествата от числата, съставящи стълбовете. Да се напише програма, която намира номерата на всички стълбове на матрицата Anxm, които си приличат.

Задача 26. Матрицата A има седлова точка в a[i, j], ако a[i, j] е минимален елемент в i-я ред и максимален елемент в j-я стълб на A. Да се напише програма, която намира *всички* седлови точки на дадена матрица A.

Задача 27. Матрицата A има седлова точка в a[i, j], ако a[i, j] е минимален елемент в i-я ред и максимален елемент в j-я стълб на A. Да се напише програма, която установява дали *съществува* седлова точка в дадена матрица A.

Задача 28. Дадена е квадратна матрица A от n-ти ред. Да се напише програма, която установява дали съществува k ($0 \le k \le n-1$), така че k-я стълб на A да съвпада с k-я й ред.

Задача 29. Дадена е реалната квадратна матрица Anxn. Да се напише програма, която намира:

а) $\max\limits_{0\le i\le n-1} \{ \min\limits_{0\le j\le n-1} \{a_{ij}\}\}$

б) $\max\limits_{0\le j\le n-1} \{ \min\limits_{0\le i\le n-1} \{a_{ij}\}\}$

в) $\min\limits_{0\le i\le n-1} \{ \max\limits_{0\le j\le n-1} \{a_{ij}\}\}$

г) $\min\limits_{0\le j\le n-1} \{ \max\limits_{0\le i\le n-1} \{a_{ij}\}\}$

Задача 30. Дадена е квадратната матрица Anxn ($2 \le n \le 10$). Да се напише програма, която определя явява ли се A ортонормирана, т.е. такава, че скаларното произведение на всеки два различни реда на A е равно на 0, а скаларното произведение на всеки ред на себе си е равно на 1?

Задача 31. Да се напише програма, която определя явява ли се квадратната матрица Anxn магически квадрат, т.е. такава, че сумата от елементите от всички редове и стълбове е еднаква.

Задача 32. Дадена е система от линейни уравнения от n-ти ред. Да се напише програма, която я решава.

Задача 33. С тройката (i, j, v) се представя елемента v от i-я ред и j-я стълб на матрица. Две матрици с размерност nxn са представени като редици от тройки. Тройките са подредени по редове. Ако тройката (i, j, v) отсъства, приема се, че v = 0. Да се напише

програма, която събира матриците и представя резултата като редица от тройки.

Допълнителна литература

1. Ст. Липман, Езикът C++ в примери, “КОЛХИДА ТРЕЙД” КООП, С. 1993.

2. К. Хорстман, Принципи на програмирането със C++, С., СОФТЕХ, 2000.

3. B. Stroustrup, C++ Programming Language. Third Edition, Addison – Wesley, 1997.

Глава 7

Типове указател и псевдоним

1. Тип указател

Променлива, това е място за съхранение на данни, което може да съдържа различни стойности. Идентифицира се с дадено от потребителя име (идентификатор). Има си и тип. Дефинира се като се указват задължително типът и името й. Типът определя броя на байтовете, в които ще се съхранява променливата, а също и множеството от операциите, които могат да се изпълняват над нея. Освен това, с променливата е свързана и стойност – неопределена или константа от типа, от който е тя. Нарича се още **rvalue**. Мястото в паметта,

в което е записана rvalue се **нарича адрес на променливата** или **lvalue**. По-точно адресът е адреса на първия байт от множеството байтове, отделени за променливата.

Пример: Фрагментът

int i = 1024;

дефинира променлива с име i и тип int. Стойността й (rvalue) е 1024. i именува място от паметта (lvalue) с размери 4 байта, като lvalue е адреса на първия байт.

Намирането на адреса на дефинирана променлива става чрез унарния префиксен дясноасоциативен оператор & (амперсанд). Приоритетът му е същия като на унарните оператори +, -, !, ++, -- и др. Фиг. 1 описва оператора.

Синтаксис

&<променлива>

където <променлива> е вече дефинирана променлива.

Семантика

Намира адреса на <променлива>.

Фиг. 1.

Пример: &i е адреса на променливата i и може да се изведе чрез оператора cout << &i;

Операторът & не може да се прилага върху константи и изрази, т.е. &100 и &(i+5) са недопустими обръщения. Не е възможно също прилагането му и върху променливи от тип масив, тъй като те имат и смисъла на константни указатели.

Адресите могат да се присвояват на специален тип променливи, наречени променливи от тип указател или само указатели.

Задаване на тип указател

Нека T е име или дефиниция на тип. За типа T, T* е тип, наречен указател към T. T се нарича **указван тип** или **тип на указателя**.

Примери:

int* е тип указател към int;

enum {a, b, c}* е тип указател към enum {a, b, c}.

Множество от стойности

Състои се от адресите на данните от тип T, дефинирани в програмата, преди използването на T*. Те са константите на типа T*. Освен тях съществува специална константа с име NULL, наречена **нулев указател**. Тя може да бъде свързвана с всеки указател независимо от неговия тип. Тази константа се интерпретира като "сочи към никъде", а в позиция на предикат е false.

Променлива величина, множеството от допустимите стойности, на която съвпада с множеството от стойности на типа T*, се нарича променлива от тип T* или променлива от тип указател към тип T. Дефинира се по стандартния начин. Фиг. 2 показва синтаксиса на дефиниция на променлива от тип указател.

Дефиниция на променлива от тип указател

T* <променлива> [= <стойност>]; |

T *<променлива> [= <стойност>];

където

T е име или дефиниция на тип;

<променлива> ::= <идентификатор>

<стойност> е шестнадесетично цяло число, представляващо адрес на данна от тип T или NULL.

Фиг. 2.

T определя типа на данните, които указателят адресира, а също и начина на интерпретацията им.

Възможно е фрагментите

<променлива> [=<стойност>] и

*<променлива>[=<стойност>]

да се повтарят. За разделител се използва запетаята. В първия случай обаче има особеност. Дефиницията

T* a, b;

е еквивалентна на

T* a;

T b;

т.е. само променливата a е указател.

Примери: Дефиницията

int *pint1, *pint2;

задава два указателя към тип int, а

int *pint1, pint2;

- указател pint1 към int и променлива pint2 от тип int.

Дефиницията на променлива от тип указател предизвиква в ОП да се отделят 4B, в които се записва някакъв адрес от множеството от стойности на съответния тип, ако дефиницията е с инициализация и неопределено или NULL, ако дефиницията не е с инициализация. (За реализацията Visual C++ 6.0 е неопределено). Този адрес е **стойността** на променливата от тип указател, а записаното на този адрес е **съдържанието** й.

Пример: Дефинициите

int i = 12;

int* p = &i; // p е инициализирано с адреса на i

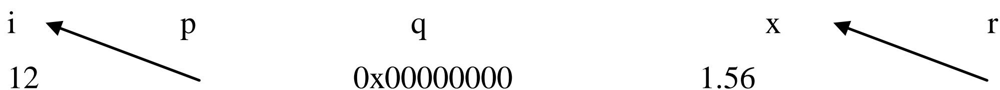
double *q = NULL; // q е инициализирано с нулевия указтел

double x = 1.56;

double *r = &x; // r е инициализирано с адреса на x

предизвикват следното разпределение на паметта

ОП



Съвет: Всеки указател, който не сочи към конкретен адрес, е добре да се свърже с константата NULL. Ако по невнимание се опитате да използвате нулев указател, програмата ви може да извърши нарушение при достъп и да блокира, но това е по-добре, отколкото указателят да сочи към кой знай къде.

Операции и вградени функции

Извличане на съдържанието на указател

Осъществява се чрез префиксния, дясноасоциативен унарен оператор * (Фиг. 3).

Синтаксис

<променлива_от_тип_указател>

Семантика

Извлича стойността на адреса, записан в <променлива_от_тип_ указател>, т.е. съдържанието на <променлива_от_тип_ указател>.

Фиг. 3.

Като използваме дефинициите от примера по-горе, имаме:

*p е 12 // 12 е съдържанието на p

*r е 1.56// 1.56 е съдържанието на r

Освен, че намира съдържанието на променлива от тип указател, обръщението

*<променлива_от_тип_указател>

е данна от тип T (променлива или константа). Всички операции, допустими за типа T, са допустими и за нея.

Като използваме дефинициите от примера по-горе, *p и *r са цяла и реална променливи, съответно. След изпълнение на операторите за присвояване

```
*p = 20;
*r = 2.18;
```

стойността на i се променя на 20, а тази на r – на 2.18.

Аритметични и логически операции

Променливите от тип указател могат да участват като операнди в следните аритметични и логически операции +, -, ++, --, ==, !=, >, >=, < и <=. Изпълнението на аритметични операции върху указатели е свързано с някои особености, заради което аритметиката с указатели се нарича още адресна аритметика. Особеноста се изразява в т. нар. мащабиране. Ще го изясним чрез пример.

Да разгледаме фрагмента

```
int *p;
double *q;
...
p = p + 1;
q = q + 1;
```

Операторът p = p + 1; свързва p не със стойността на p, увеличена с 1, а с p + 1*4, където 4 е броя на байтовете, необходими за записване на данна от тип int (p е указател към int). Аналогично, q = q + 1; увеличава стойността на q не с 1, а с 8, тъй като q е указател към double (8 байта са необходими за записване на данна от този тип).

Общото правило е следното: Ако p е указател от тип T*, p+i е съкратен запис на p + i*sizeof(T), където sizeof(T) е функция, която намира броя на байтовете, необходими за записване на данна от тип T.

Въвеждане

Не е възможно въвеждане на данни от тип указател чрез оператора cin. Свързването на указател със стойност става чрез инициализация или оператора за присвояване.

Извеждане

Осъществява се по стандартния начин - чрез оператора cout.

Допълнение

Типът, който се задава в дефиницията на променлива от тип указател, е информация за компилатора относно начина, по който да се интерпретира съдържанието на указателя. В контекста на горния пример *p са четири байта, които ще се интерпретират като цяло число от тип int. Аналогично, *q са осем байта, които ще се интерпретират като реално число от тип double.

Следващата програма илюстрира дефинирането и операциите за работа с указатели.

```
#include <iostream.h>
int main()
{int n = 10;   // дефинира и инициализира цяла променлива
 int* pn = &n;      // дефинира и инициализира указател pn към n
 // показва, че указателят сочи към n
 cout << "n=  " << n << "  *pn= " << *pn << '\n';
 // показва, че адресът на n е равен на стойността на pn
 cout << "&n= "<< &n << "  pn= " << pn << '\n';
 // намиране на стойността на n чрез pn
 int m = *pn; //  == 10
 // промяна на стойността на n чрез pn
 *pn = 20;
 // извеждане на стойността на n
 cout << "n= " << n << '\n'; // n == 20
 return 0;
}
```

В някои случаи е важна стойността на променливата от тип указател (адресът), а не нейното съдържание. Тогава тя се дефинира като указател към *тип void*. Този тип указатели са предвидени с цел една и съща променлива - указател да може в различни моменти да сочи към данни от различен тип. В този случай, при опит да се използва съдържанието на променливата от тип указател, ще се предизвика грешка. Съдържанието на променлива - указател към тип void може да се извлече само след привеждане на типа на указателя (void*) до типа на съдържанието. Това може да се осъществи чрез операторите за преобразуване на типове.

Пример:

```
int a = 100;
void* p;                    // дефинира указател към void
p = &a;                     // инициализира p
cout << *p;                 // грешка
cout << *((int*) p);        // преобразува p в указател към int
                                    // и тогава извлича съдържанието му.
```

В C++ е възможно да се дефинират указатели, които са константи, а също и указатели, които сочат към константи. И в двата случая се използва запазената дума const, която се поставя пред съответните елементи от дефинициите на указателите. Стойността на елемента, дефиниран като const (указателя или обекта, към който сочи) не може да бъде променяна.

Пример:

```
int i, j = 5;
int *pi;                    // pi е указател към int
int * const b = &i;     // b е константен - указател към int
const int *c = &j; // c е указател към цяла константа.
b = &j;                     // грешка, b е константен указател
*c = 15;                // грешка, *c е константа
```

2. Указатели и масиви

В C++ има интересна и полезна връзка между указателите и масивите. Тя се състои в това, че имената на масивите са указатели към техните "първи" елементи. Това позволява указателите да се разглеждат като алтернативен начин за обхождане на елементите на даден масив.

Указатели и едномерни масиви

Нека a е масив, дефиниран по следния начин:

```
int a[100];
```

Тъй като a е указател към a[0], *a е стойността на a[0], т.е. *a и a[0] са два различни записа на стойността на първия елемент на масива. Тъй като елементите на масива са разположени последователно в паметта, a + 1 е адреса на a[1], a + 2 е адреса на a[2] и т.н. a + n-1 е адреса на a[n-1]. Тогава *(a+i) е друг запис на a[i] (i = 0, 1, ..., n-1).

Има обаче една особеност. Имената на масивите са константни указатели. Заради това, някои от аритметичните операции, приложими над указатели, не могат да се приложат над масиви. Такива са ++, -- и присвояването на стойност.

Следващата програма показва два начина за извеждане на елементите на масив.

```
#include <iostream.h>
int main()
{int a[] = {1, 2, 3, 4, 5, 6};
 for (int i = 0; i <=  5; i++)
        cout << a[i] << '\n';
 for (i = 0; i <=  5; i++)
        cout << *(a+i) << '\n';
 return 0;
}
```

Фрагментът

```
for (i = 0; i <= 5; i++)
{cout << *a << '\n';
 a++;
}
```

съобщава за грешка заради оператора a++ (a е константен указател и не може да бъде променян). Може да се поправи като се използва помощна променлива от тип указател към int, инициализирана с масива a, т.е.

```
int* p = a;
for (i = 0; i <= 5; i++)
{cout << *p << '\n';
 p++;
 }
```

Използването на указатели е по-бърз начин за достъп до елементите на масива и заради това се предпочита. Индексираните променливи правят кода по-ясен и разбираем. В процеса на компилация всички конструкции от вида a[i] се преобразуват в *(a+i), т.е. операторът за индексиране [...] се обработва от компилатора чрез адресна аритметика. Полезно е да отбележим, че операторът [] е лявоасоциативен и с по-висок приоритет от унарните оператори (в частност от оператора за извличане на съдържание *).

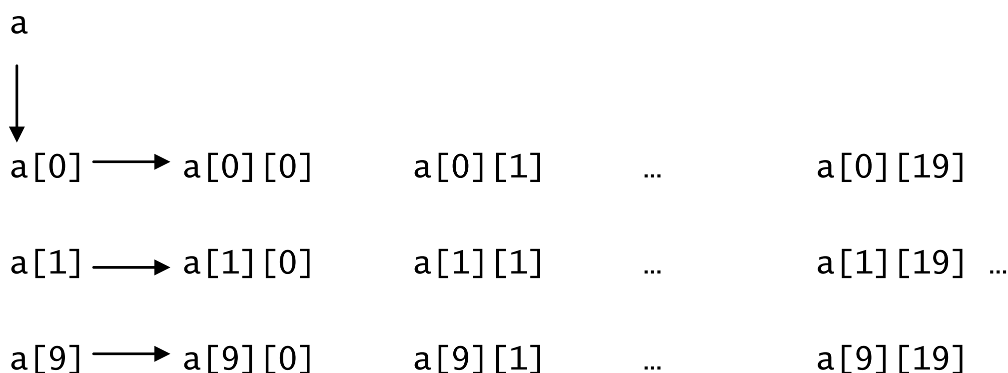
Указатели и двумерни масиви

Името на двумерен масив е константен указател към първия елемент на едномерен масив от константни указатели. Ще изясним с пример казаното.

Нека a е двумерен масив, дефиниран по следния начин:

int a[10][20];

Променливата a е константен указател към първия елемент на едномерния масив a[0], a[1], ..., a[9], като всяко a[i] е константен указател към a[i][0] (i = 0,1, ..., 9), т.е.



Тогава

**a == a[0][0]

a[0] == *a   a[1] == a[0]+1   ...   a[9] == a[0]+9,

т.е.

a[i] == a[0] + i == *a + i

Като използваме, че операторът за индексиране е лявоасоциативен, получаваме:

a[i][j] == (*a + i) [j] == *((*a + i) + j).

Задача 67. Да се напише програма, която въвежда по редове правоъгълна матрица Anxk от реални числа и извежда матрицата, образувана от редовете на A от в четна позиция, като всеки елемент е увеличен с 1, след което извежда матрицата, образувана от редовете от нечетна позиция, като всеки елемент е увеличен с 2 и накрая, ако n е четно, извежда сумата на матриците от редовете от четните и нечетните позиции на A.

Програма Zad67.cpp решава задачата.

Program Zad67.cpp

```
#include <iostream.h>
#include <iomanip.h>
int main()
{int a[20][100];
 int* p[20];
 int* q[20];
 cout << "n, k = ";
```

```
int n, k;
cin >> n >> k;
if (!cin)
{cout << "Error! \n";
 return 0;
}
int i, j;
for (i = 0; i <= n-1; i++)
 for (j = 0; j <= k-1; j++)
 {cout << "a[" << i << "][" << j << "]= ";
 cin >> *(*(a+i)+j);
}
 for (i = 0; i <= n-1; i++)
 {for(j = 0; j <= k-1; j++)
   cout << setw(10) << *(*(a+i)+j);
  cout << '\n';
 }
 cout << "\n\n first new array\n";

 int m = -1;
 for (i = 0; i <= n-1; i = i+2)
 {m++;
  *(p+m) = *(a+i);
 }
 for (i = 0; i <= m; i++)
 {for(j = 0; j <= k-1; j++)
     cout << setw(10) << *(*(p+i)+j)+1;
  cout << '\n';
 }
 int l = -1;
 for (i = 1; i <= n-1; i = i+2)
 {l++;
  q[l] = a[i];
 }
```

```
 cout << "\n\n second new array \n";
 for (i = 0; i <= l; i++)
 {for(j = 0; j <= k-1; j++)
     cout << setw(10) << *(*(q+i)+j)+2;
  cout << '\n';
 }
 cout << "\n\n third new array \n";
 if (n%2 == 0)
      for (i = 0; i <= m; i++)
      {for (j = 0; j <= k-1; j++)
        cout << setw(10) << *(*(p+i)+j) + *(*(q+i)+j);
       cout << '\n';
      }
return 0;
}
```

3. Указатели и низове

Низовете са масиви от символи. Името на променлива от тип низ е константен указател, както и името на всеки друг масив. Така всичко, което казахме за връзката между масив и указател е в сила и за низ – указател.

Следващият пример илюстрира обхождане на низ чрез използване на указател към char. Обхождането продължава до достигане на знака за край на низ.

```
#include <iostream.h>
int main()
{char str[] = "C++Language"; // str е константен указател
 char* pstr = str;
 while (*pstr)
 {cout << *pstr << '\n';
  pstr++;
 }// pstr вече не е свързан с низа "C++Language".
 return 0;
}
```

Тъй като низът е зададен чрез масива от символи str, str е константен указател и не може да бъде променяна стойността му. Затова се налага използването на помощната променлива pstr.

Ако низът е зададен чрез указател към char, както е в следващата програма, не се налага използването на такава.

```
#include <iostream.h>
int main()
{char* str = "C++Language";  // str е променлива
 while (*str)
 {cout << *str << '\n';
  str++;
 }
 return 0;
}
```

Примерите показват, че задаването на низ като указател към char има предимство пред задаването като масив от символи. Ще отбележим обаче, че дефиницията

```
char* str = "C++Language";
```

не може да бъде заменена с

```
char* str;
cin >> str;
```

следвани с въвеждане на низа “C++Language”, докато дефиницията

```
char str[20];
```

позволява въвеждането му чрез cin, т.е.

```
cin >> str;
```

Има още една особеност при дефинирането на низ като указател към char за реализацията Visual C++ 6.0. Ще я илюстрираме с пример.

Нека дефинираме променлива от тип низ по следния начин:

```
char s[] = “abba”;
```

Операторът

```
*s = ‘A’;
```

е еквивалентен на s[0] = ‘A’ и ще замени първото срещане на символа ‘a’ в s с ‘A’. Така

```
cout << s;
```

извежда низа

```
Abba
```

Да разгледаме съответната дефиницията на s чрез указател към char

char* s = “abba”;

Операторът

*s = ‘A’;

би трябвало да замени първото срещане на символа ‘a’ в s с ‘A’, тъй като s съдържа адреса на първото ‘a’. Тук обаче реализацията на Visual C++ съобщава за грешка – нарушение на достъпа. Последното може да се избегне като опцията на компилатора /ZI се замени с /Zi. Това се реализира като в менюто Project се избере Settings, след това C/C++, където Category трябва да има опция General. Накрая, в Project Options се промени /ZI на /Zi. Опцията /Zi се грижи за проблемите при нарушаване на достъпа, едно неудобство, което трябва да се има предвид.

4. Тип псевдоним

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

Задаване на тип псевдоним

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

Множество от стойности

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

Пример: Нека програмата съдържа следните дефиниции

int a, b = 5;

...

int x, y = 9, z = 8;

...

Множеството от стойности на типа int& съдържа имената a, b, x, y, z. Променлива величина, множеството от допустимите стойности на която съвпада с множеството от стойности на даден тип псевдоним, се нарича променлива от този тип псевдоним. Фиг. 4 илюстрира дефиницията.

<дефиниция_на_променлива_от_тип_псевдоним> ::=
T& <var> = <defined_var_of_T>; |
T &<var> = <defined_var_of_T>;
където
T е име тип, а
<defined_var_of_T> е име на вече дефинирана променлива от тип T.
Нарича се инициализатор.

Фиг. 4.

Възможно е фрагментите

<var> = <defined_var_of_T> и

&<var> = <defined_var_of_T>

да се повтарят многократно. За разделител се използва символът запетая. Има обаче една особеност. Дефиницията

T& a = b, c = d;

е еквивалентна на

T& a = b;

T c = d;

Пример: Дефинициите

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

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

Дефинициите задължително са с инициализация – променлива от същия тип като на базовия тип на типа псевдоним. Освен това, след инициализацията, променливата псевдоним не може да се променя като й се присвоява нова променлива или чрез повторна дефиниция. Затова тя е "най-константната" променлива, която може да съществува.

Пример:

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

Операции и вградени функции

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

Примери:

```
1. int ii = 0;
      int& rr = ii;
      rr++;
      int* pp = &rr;
```

Резултатът от изпълнението на първите два оператора е следния:

ii, rr

... 0	...	

Операторът rr++; не променя адреса на rr, а стойността на ii и тя от 0 става 1. В случая rr++ е еквивалентен на ii++. Адресът на rr е адреса на ii. Намира се чрез &rr. Чрез дефиницията

int* pp = &rr;

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

2. int a = 5;

```
int &syn = a;
cout << syn << "  " << a << '\n';
int b = 10;
syn = b;
cout << b << "  " << a << "  " << syn << '\n';
```

извежда

```
5      5
10     10 10
```

Операторът syn = b; е еквивалентен на a = b;.

3. int i = 1;

```
int& r = i;     // r и i са свързани с едно и също цяло число
cout << r; // извежда 1
int x = r; // x има стойност 1
r = 2;     // еквивалентно е на i = 2;
```

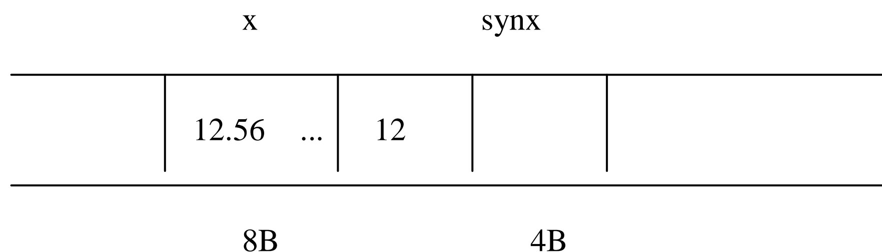
Допълнение: Възможно е типът на инициализатора да е различен от този на псевдонима. В този случай се създава нова, наречена **временна**, променлива от типа на псевдонима, която се инициализира със зададената от инициализатора стойност, преобразувана до типа на псевдонима.

Например, след дефиницията

double x = 12.56;

int& synx = x;

имаме



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

Константни псевдоними

В C++ е възможно да се дефинират псевдоними, които са константи. За целта се използва запазената дума const, която се поставя пред дефиницията на променливата от тип псевдоним. По такъв начин псевдонимът не може да променя стойността си, но ако е псевдоним на променлива, промяната на стойността му може да стане чрез промяна на променливата.

Пример: Фрагментът

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

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

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

ще изведе

```
125 125
150 150
```

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

Допълнителна литература

1. Ст. Липман, Езикът C++ в примери, “КОЛХИДА ТРЕЙД” КООП, С. 1993.

2. B. Stroustrup, C++ Programming Language. Third Edition, Addison – Wesley, 1997.

Глава 8

Функции

Добавянето на нови оператори и функции в приложенията, реализирани на езика C++, се осъществява чрез функциите. Те са основни структурни единици, от които се изграждат програмите на езика. Всяка функция се състои от множество от оператори, оформени подходящо за да се използват като обобщено действие или операция. След като една функция бъде дефинирана, тя може да бъде изпълнявана многократно за различни входни данни.

Програмите на езика C++ се състоят от една или повече функции. Сред тях задължително трябва да има точно една с име main и наречена **главна функция**. Тя е първата функция, която се изпълнява при стартиране на програмата. Главната функция от своя страна може да се обръща към други функции. Нормалното изпълнение на програмата завършва с изпълнението на главната функция (Възможно е изпълнението да завърши принудително с изпълнението на функция, различна от главната).

Използването на функции има следните предимства:

- Програмите стават ясни и лесни за тестване и модифициране.

- Избягва се многократното повтаряне на едни и същи програмни фрагменти. Те се дефинират еднократно като функции, след което могат да бъдат изпълнявани произволен брой пъти.

- Постига се икономия на памет, тъй като кодът на функцията се съхранява само на едно място в паметта, независимо от броя на нейните изпълнения.

Ще разгледаме най-общо разпределението на оперативната памет за изпълнима програма на C++. Чрез няколко примерни програми ще покажем дефинирането, обръщението и изпълнението на функции, след което ще направим съответните обобщения.

1. Разпределение на ОП за изпълнима програма

Разпределението на ОП зависи от изчислителната система, от типа на операционната система, а също от модела памет. Най-общо се състои от: *програмен код, област на статичните данни, област на динамичните данни* и *програмен стек* (Фиг. 1).

Фиг. 1.

Програмен код

В тази част е записан изпълнимият код на всички функции, изграждащи потребителската програма.

Област на статичните данни

В нея са записани глобалните обекти на програмата.

Област на динамичните данни

За реализиране на динамични структури от данни (списъци, дървета, графи, ...) се използват средства за динамично разпределение на паметта. Чрез тях се заделя и освобождава памет в процеса на изпълнение на програмата, а не преди това (при компилирането й). Тази памет е от областта на динамичните данни.

Програмен стек

Този вид памет съхранява данните на функциите на програмата. Стекът е динамична структура, организирана по правилото "последен влязъл – пръв излязъл". Той е редица от елементи с пряк достъп до елементите от единия си край, наречен **връх**. Достъпът се реализира чрез указател. Операцията включване се осъществява само пред елемента от върха, а операцията изключване – само за елемента от върха.

Елементите на програмния стек са "блокове" от памет, съхраняващи данни, дефинирани в някаква функция. Наричат се **стекови рамки**.

2. Примери за програми, които дефинират и използват функции

Задача 68. Да се напише програма, която въвежда стойности на естествените числа a, b, c и d и намира и извежда най-големият общ делител на числата a и b, след това на c и d и накрая на a, b, c и d.

Програма Zad68.cpp решава задачата. Тя се състои от две функции: gcd и main. Функцията gcd(x, y) намира най-големия общ делител на естествените числа x и y. Тъй като main се обръща към (извиква) gcd, функцията gcd трябва да бъде известна преди функцията main. Най-лесният начин да се постигне това е във файла, съдържащ програмата, първо да се постави дефиницията на gcd, а след това тази на main. Ще бъде показан алтернативен начин по-късно.

Описанието на функцията gcd прилича на това на функцията main. Състои се от заглавие

```
int gcd(int x, int y)
```

и тяло

```
{while (x != y)
 if (x >= y) x = x-y; else y = y-x;
 return x;
```

}

Заглавието определя, че gcd е име на двуаргументна целочислена функция с цели аргументи, т.е.

gcd: int x int ⟶ int

Името е произволен идентификатор. В случая е направен мнемонически избор. Запазената дума int пред името на функцията е типа й (по-точно е типа на резултата на функцията). В кръгли скобки и отделени със запетая са описани параметрите x и y на gcd. Те са различни идентификатори. Предшестват се от типовете си. Наричат се **формални параметри за функцията**.

Тялото на функцията е блок, реализиращ алгоритъма на Евклид за намиране на най-голямия общ делител на естествените числа x и y. Завършва с оператора

return x;

чрез който се прекратява изпълнението на функцията като стойността на израза след return се връща като стойност на gcd в мястото, в случая в main, в което е направено обръщението към нея.

```
Program Zad68.cpp
#include <iostream.h>
int gcd(int x, int y)
{while (x != y)
 if (x >= y) x = x-y; else y = y-x;
 return x;
 }
 int main()
 {cout << "a, b, c, d= ";
  int a, b, c, d;
  cin >> a >> b >> c >> d;
  if (!cin || a < 1 || b < 1 || c < 1 || d < 1)
  {cout << "Error \n";
   return 1;
  }
  int r = gcd(a, b);
```

```
 cout << "gcd{" << a << ", " << b << "}= " << r << "\n";
 int s = gcd(c, d);
 cout << "gcd{" << c << ", " << d << "}= " << s << "\n";
 cout << "gcd{" << a << ", " << b << ", " << c << ", "
        << d << "}= " << gcd(r, s) << "\n";
  return 0;
}
```

Изпълнение на програма Zad68.cpp

Дефинициите на функциите main и gcd се записват в областта на паметта, определена за програмния код. Изпълнението на програмата започва с изпълнение на функцията main. Фрагментът

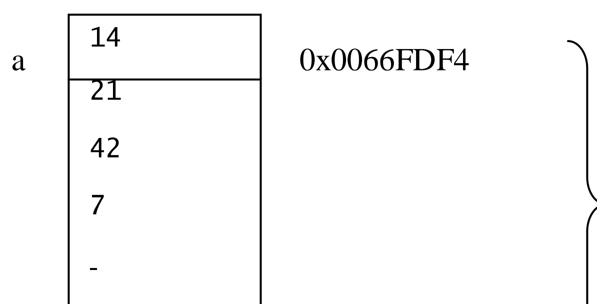
```
 cout << "a, b, c, d= ";
int a, b, c, d;
cin >> a >> b >> c >> d;
if (!cin || a < 1 || b < 1 || c < 1 || d < 1)
{cout << "Error \n";
 return 1;
}
```

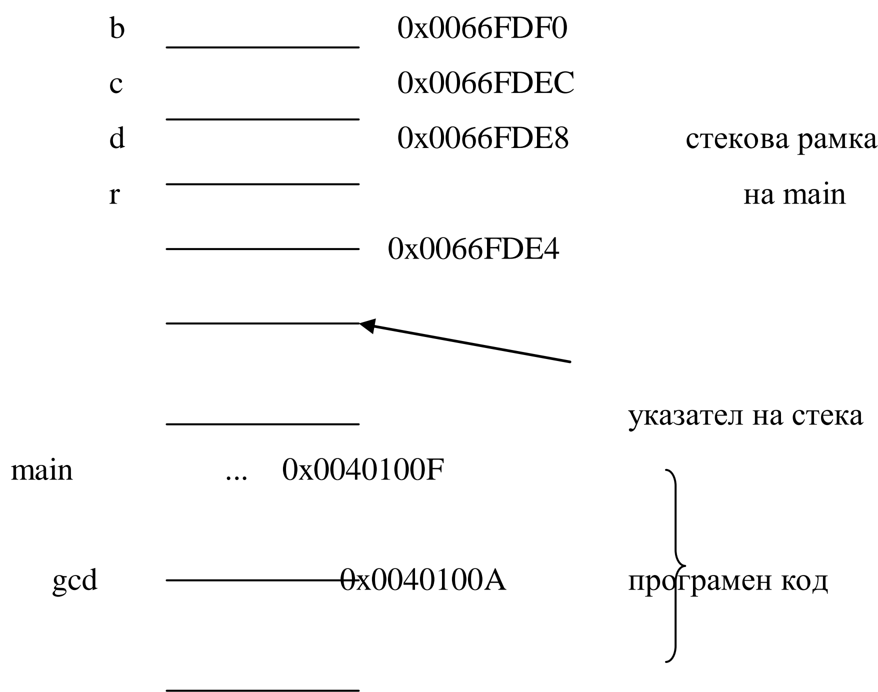
дефинира и въвежда стойности на целите променливи a, b, c и d като осигурява да са естествени числа. Нека за a, b, c и d са въведени 14, 21, 42 и 7 съответно. В тази последователност те се записват в дъното на програмния стек (Фиг. 2.). Така на дъното на стека се оформя “блок” от памет за main с достатъчно големи размери, който освен променливите от main съдържа и някои “вътрешни” данни. Този блок се нарича **стекова рамка на main.**

Операторът

```
int r = gcd(a, b);
```

дефинира цялата променлива r като в стековата рамка на main, веднага след променливата d отделя 4B, в които ще запише резултатът от обръщението gcd(a, b) към функцията gcd. Променливите a и b се наричат **фактически параметри за това обръщение**. Забелязваме, че типът им е същия като на съответните им формални параметри x и y.





Фиг. 2.

Обръщение към gcd(a, b)

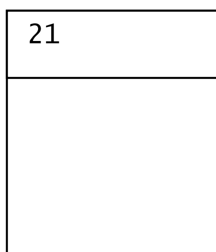
В програмния стек се генерира нов блок памет – стекова рамка за функцията gcd. В него се записват формалните и локалните параметри на gcd, а също и някои "вътрешни" данни като return-адреса и адреса на стековата рамка на main.

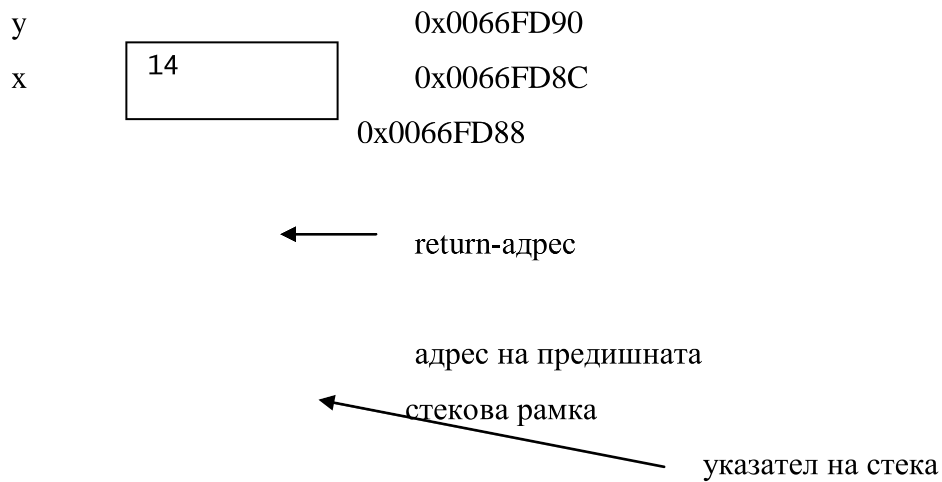
Обръщението се осъществява на два етапа:

а) Свързване на формалните с фактическите параметри

В стековата рамка на gcd, се отделят по 4 байта за формалните параметри x и y в обратен ред на реда, в които са записани в заглавието. В тази памет се **откопирват стойностите** на съответните им фактически параметри. Отделят се също 4B за т. нар. return-адрес, адреса на мястото в main, където ще се върне резултатът, а също се отделя памет, в която се записва адресът на предишната стекова рамка, т.е.

памет за gcd (I-во обръщение към него)

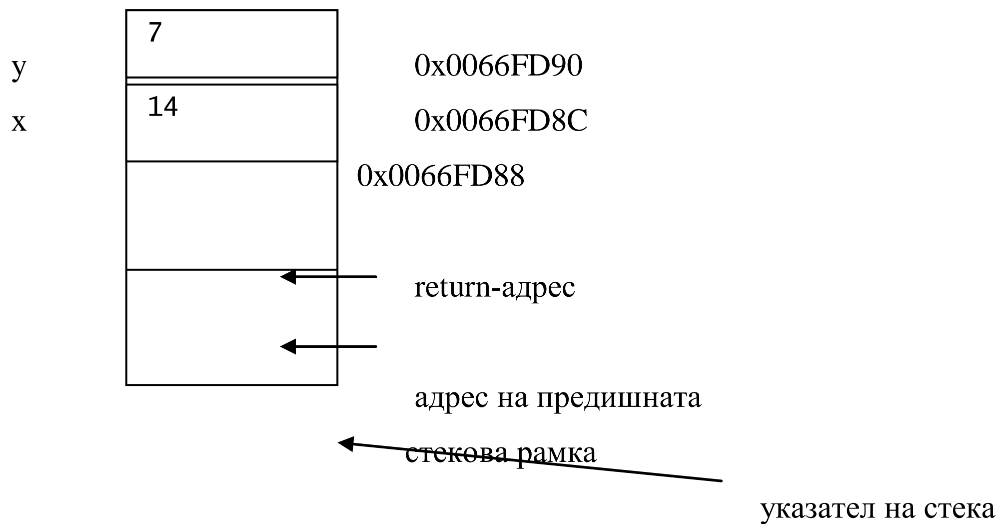




б) Изпълнение на тялото на gcd

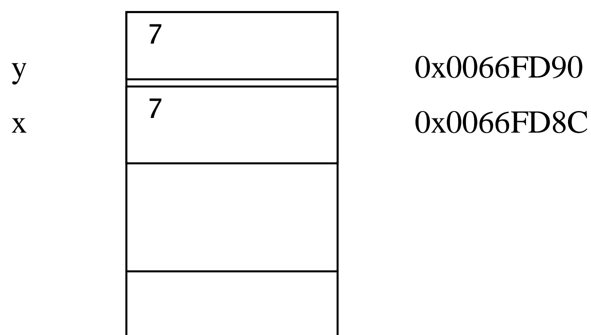
Тъй като е в сила y > x, стойността на y се променя на 7, т.е.

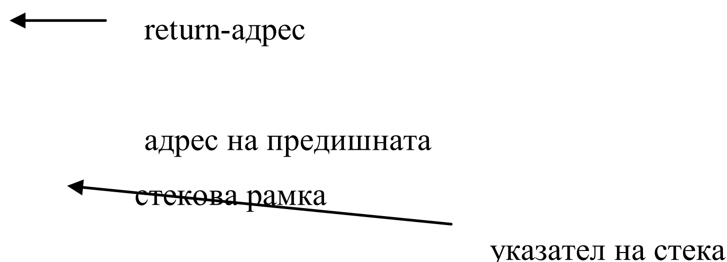
памет в стека за gcd



Сега пък е в сила x > y, което води до промяна стойността му на 7, т.е.

памет в стека за gcd





Операторът за цикъл завършва изпълнението си. Изпълнението на оператора

return x;

преустановява изпълнението на gcd като връща в main в мястото на прекъсването (return-адреса) стойността 7 на обръщението gcd(a, b). Отделената за gcd стекова рамка се освобождава. Указателят на стека сочи края на стековата рамка на main. Изпълнението на програмата продължава с инициализацията на r. Резултатът от обръщението gcd(14, 21) се записва в отделената за r памет.

Операторът

cout << "gcd{" << a << ", " << b << "}= " << r << "\n";

извежда получения резултат.

Изпълнението на останалите обръщения към gcd се реализира по същия начин. При обръщението към всяко от тях в стека се създава стекова рамка на gcd, а след завършване на обръщението, рамката се освобождава. При достигане до оператора return 0; от main, се освобождава и стековата рамка на main.

Функцията gcd реализира най-простото и “чисто” дефиниране и използване на функции – получава входните си стойности единствено чрез формалните си параметри и връща резултата си чрез оператора return. Забелязваме, че обръщението gcd(a, b) работи с копия на стойностите на a и b, запомнени в x и y, а не със самите a и b. В процеса на изпълнение на тялото на gcd, стойностите на x и y се променят, но това не оказва влияние на стойностите на фактическите параметри a и b.

Такова свързване на формалните с фактическите параметри се нарича **свързване по стойност** или още **предаване на параметрите по стойност.** При него фактическите параметри магат да бъдат не само променливи, но и изрази от типове, съвместими с типовете на

съответните формални параметри. Обръщението gcd(gcd(a, b), gcd(c, d)) е коректно и намира най-големия общ делител на a, b, c и d.

В редица случаи се налага функцията да получи входа си чрез някои от формалните си параметри и да върне резултат не по обичайния начин – чрез оператора return, а чрез същите или други параметри. Задача 69 дава пример за това.

Задача 69. Да се напише програма, която въвежда стойности на реалните числа a, b, c и d, след което разменя стойностите на a и b и на c и d съответно.

Ако дефинираме функция swapi(double* x, double* y), която разменя стойностите на реалните числа към които сочат указателите x и y, обръщението swapi(&a, &b) ще размени стойностите на a и b, а обръщението swapi(&c, &d) ще размени стойностите на c и d. Програма Zad69.cpp решава задачата. Тя се състои от функциите: swapi и main. Тъ0й като main се обръща към (извиква) swapi, функцията swapi трябва да бъде известна преди функцията main. Затова във файла, съдържащ програмата, първо се поставя swapi, а след това main.

```
Program Zad69.cpp
#include <iostream.h>
#include <iomanip.h>
void swapi(double* x, double* y)
{double work = *x;
 *x = *y;
 *y = work;
 return;
}
0int main()
{cout << "a, b, c, d= ";
 double a, b, c, d;
 cin >> a >> b >> c >> d;
 cout << setprecision(2) << setiosflags(ios :: fixed);
 cout << setw(10) << a << setw(10) << b
         << setw(10) << c << setw(10) << d << '\n';
```

```
swapi(&a, &b);
swapi(&c, &d);
cout << setw(10) << a << setw(10) << b
         << setw(10) << c << setw(10) << d << '\n';
return 0;
}
```

Функцията swapi има подобна структура като на gcd. Но и заглавието, и тялото й са по-различни. Типът на swapi е указан чрез запазената дума void. Това означава, че функцията не връща стойност чрез оператора return. Затова в тялото на swapi е пропуснат изразът след return. Формалните параметри x и y са указатели към double, а в тялото се работи със съдържанията на указателите.

Забелязваме също, че обръщенията към swapi в main

```
swapi(&a, &b);
swapi(&c, &d);
```

не участват като аргументи на операции, а са оператори.

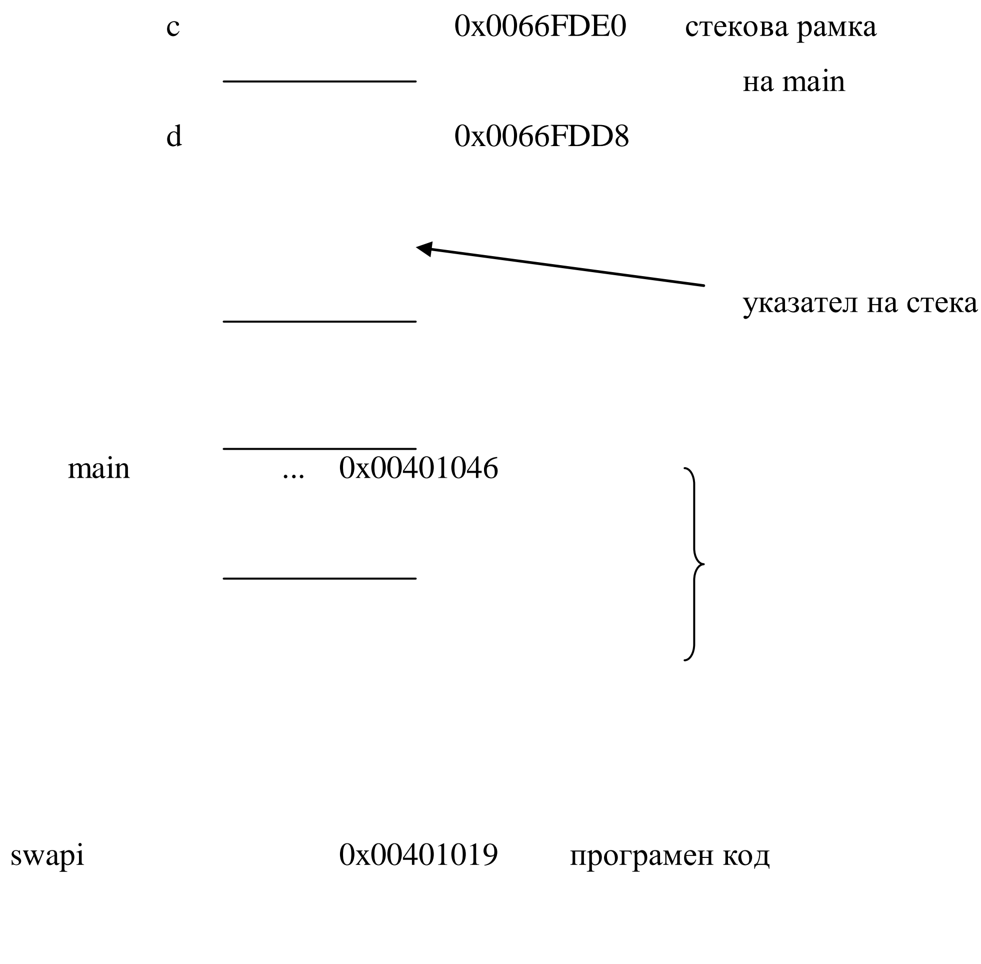
Изпълнение на програма Zad69.cpp

Дефинициите на функциите main и swapi се записват в областта на паметта, определена за програмния код. Изпълнението на програмата започва с изпълнение на функцията main. Фрагментът

```
cout << "a, b, c, d= ";
double a, b, c, d;
cin >> a >> b  >> c >> d;
cout << setprecision(2) << setiosflags(ios :: fixed);
cout << setw(10) << a << setw(10) << b
         << setw(10) << c << setw(10) << d << '\n';
```

дефинира и въвежда стойности за реалните променливи a, b, c и d и ги извежда върху екрана според дефинираното форматиране. Нека за стойности на a, b, c и d са въведени 1.5, 2.75, 3.25 и 8.2 съответно (Фиг. 3).

a	1.5	0x0066FDF0
b	2.75	0x0066FDE8
	3.25	
	8.2	



Фиг. 3.

Обръщението

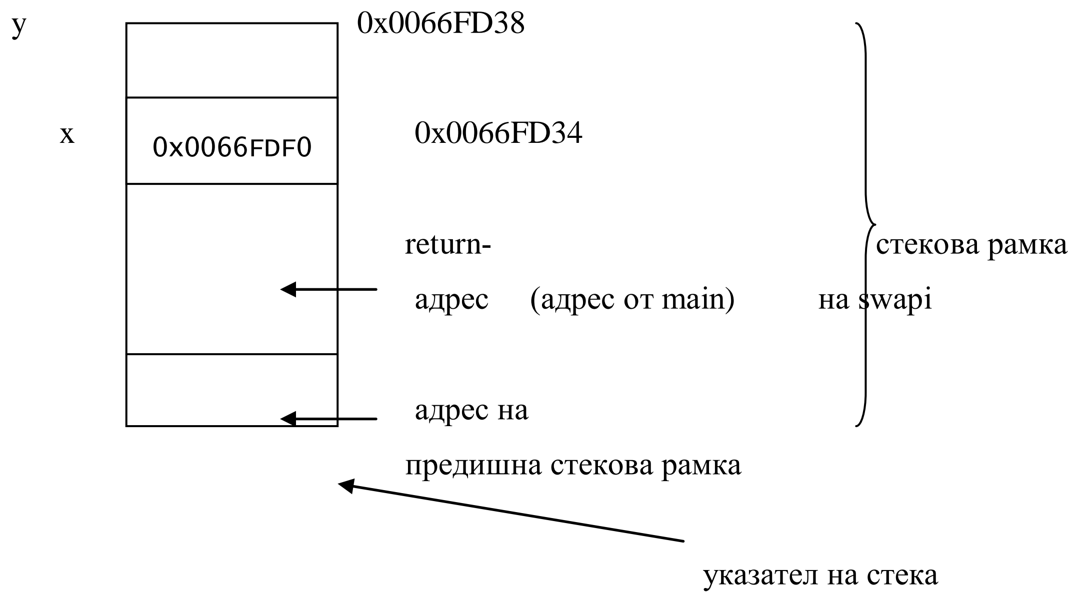
swapi(&a, &b);

се изпълнява на два етапа по следния начин:

а) Свързване на формалните с фактическите параметри

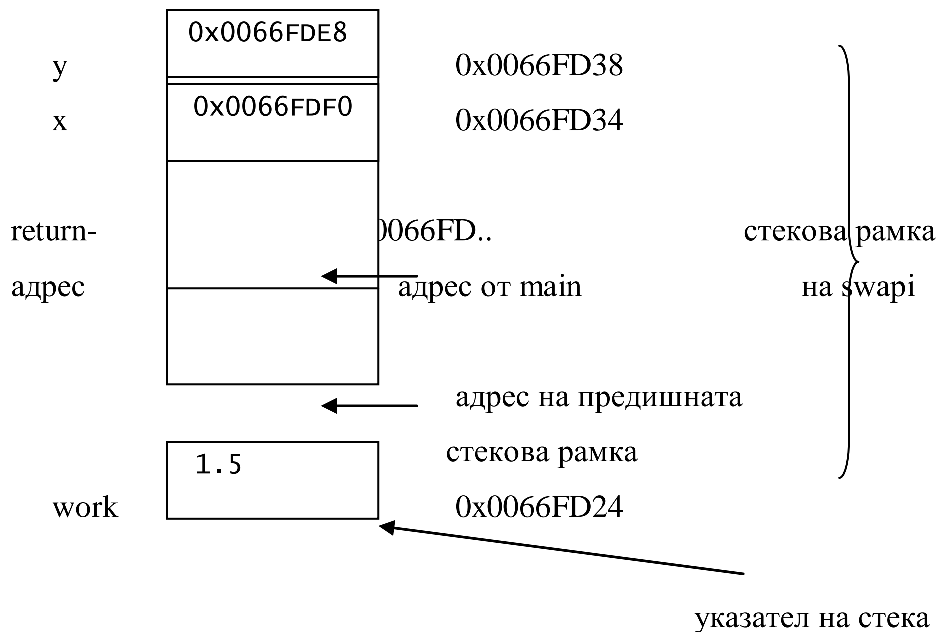
В стека се конструира нова рамка – рамката на swapi. Отделят се по 4 байта за формалните параметри x и y, в която памет се **записват адресите** на съответните им фактически параметри, още 4B, в които се записва адресът на swapi(c, d), от където трябва да се продължи изпълнението на main (return-адреса), а също и памет, в която се записва адресът на предишната стекова рамка (в случая на main).

0x0066FDE8



б) Изпълнение на тялото на swapi

Изпълнява се като блок. За реалната променлива work се отделят 8 байта в стековата рамка на swapi, в които се записва съдържанието на x, в случая 1.5, т.е.



Операторът

*x = *y;

променя съдържанието на x с това на y, а операторът

*y = work;

променя съдържанието на y като го свързва със стойността на work, т.е.

стекова рамка на main

	2.75	
a	1.5	0x0066FDF0
b	3.25	0x0066FDE8
c	8.2	0x0066FDE0
d		0x0066FDD8
	...	

Операторът return; прекъсва работа на на swapi и предава управлението в точката на извикването му в главната функция (return-адреса). Стековата рамка, отделена за swapi се освобождава. Указателят на стека сочи стековата рамка на main. В резултат стойностите на променливите a и b са разменени.

Обръщението swapi(c, d) се изпълнява по аналогичен начин. За нея се генерира нова стекова рамка (на същите адреси), която се освобождава когато изпълнението на swapi завърши.

Функцията swapi получава входните си стойности чрез формалните си параметри и връща резултата си чрез тях. Забелязваме, че обръщението swapi(&a, &b) работи не с копия на стойностите на a и b, а с адресите им. В процеса на изпълнение на тялото се променят стойностите на фактическите параметри a и b при първото обръщение към нея и на c и d – при второто.

Такова свързване на формалните с фактическите параметри се нарича **свързване на параметрите по указател** или още **предаване на параметрите по указател** или **свързване по адрес.** При този вид предаване на параметрите, фактическите параметри задължително са променливи или адреси на променливи.

Освен тези два начина на предаване на параметри, в езика C++ има още един – **предаване на параметри по псевдоним.** Той е сравнително по-удобен от предаването по указател и се предпочита от програмистите.

Ще го илюстрираме чрез същата задача. Програма Zad69_1.cpp реализира функция swapi, в която предаването на параметрите е по псевдоним.

```
Program Zad69_1.cpp
#include <iostream.h>
#include <iomanip.h>
 void swapi(double& x, double& y)
 {double work = x;
  x = y;
  y = work;
  return;
 }
int main()
{cout << "a, b, c, d= ";
 double a, b, c, d;
 cin >> a >> b >> c >> d;
 cout << setprecision(2) << setiosflags(ios :: fixed);
 cout << setw(10) << a << setw(10) << b
         << setw(10) << c << setw(10) << d << '\n';
 swapi(a, b);
 swapi(c, d);
 cout << setw(10) << a << setw(10) << b
         << setw(10) << c << setw(10) << d << '\n';
 return 0;
}
```

Ще проследим изпълнението и на тази модификация.

Изпълнението на програмата започва с изпълнение на функцията main. Фрагментът

```
cout << "a, b, c, d= ";
double a, b, c, d;
cin >> a >> b >> c >> d;
cout << setprecision(2) << setiosflags(ios :: fixed);
cout << setw(10) << a << setw(10) << b
```

```
          << setw(10) << c << setw(10) << d << '\n';
```

дефинира и въвежда стойности за реалните променливи a, b, c и d и ги извежда върху екрана според дефинираното форматиране. Нека за стойности на a, b, c и d отново са въведени 1.5, 2.75, 3.25 и 8.2 съответно. След обработката му в стека се конструира стековата рамка на main.

памет на main

		1.5	
x	a	2.75	0x0066FDF0
y	b	3.25	0x0066FDE8
	c	8.2	0x0066FDE0
	d		0x0066FDD8
		…	

Обръщението

swapi(a, b);

се изпълнява на два етапа по следния начин:

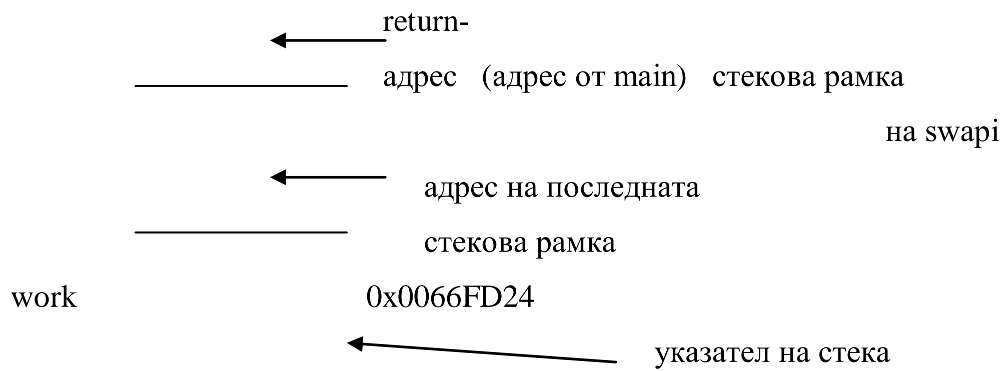
а) Свързване на формалните с фактическите параметри

За целта се генерира нова стекова рамка – рамката на swapi. Указателят на стека сочи тази рамка. Тъй като формалните параметри x и y са псевдоними на променливите a и b, за тях памет в стековата рамка на swapi не се отделя. Параметърът x "прелита" и се "закачва" за фактическия параметър a и аналогично y "прелита" и се "закачва" за фактическия параметър b от стековата рамка на main. Така всички действия с x и y в swapi се изпълняват с фактическите параметри a и b от main съответно.

б) Изпълнение на тялото на swapi

Изпълнява се като блок. В рамката на swapi, за реалната променлива work се отделят 8 байта, в които се записва стойността на x, в случая 1.5, т.е.

стекова рамка на swapi



Операторът

x = y;

присвоява на a стойността на b, а операторът

y = work;

променя стойността на променливата b като й присвоява стойността на work, т.е.

x	a	2.75	0x0066FDF0
y	b	1.5	0x0066FDE8
	c	3.25	0x0066FDE0
	d	8.2	0x0066FDD8
		…	

Операторът return; прекъсва работа на на swapi и предава управлението на return-адреса от главната функция. Стековата рамка на swapi се освобождава. Указателят на стека сочи стековата рамка на main. В резултат, стойностите на променливите a и b са разменени. Променливите a и b са “освободени от“ x и y. Следва изпълнение на обръщението

swapi(c, d);

което се реализира по същия начин (даже на същите адреси в стека).

Забелязваме, че фактическите параметри, съответстващи на формални параметри-псевдоними са променливи.

Тази реализация на swapi е по-ясна от съответната с указатели. Тялото й реализира размяна на стойностите на две реални променливи без да се налага използването на адреси.

Нека в тялото на main на Zad69_1.cpp преди оператора return; включим фрагмента:

int m, n;

```
cin >> m >> n;
swapi(m, n);
cout << setw(10) << m << setw(10) << n << "\n";
```

Някои реализации (visual C++ 6.0) ще сигнализират грешка на третата линия – невъзможност за преобразуване на параметър от int в double &, други обаче ще имат нормално поведение, но няма да разменят стойностите на m и n. Последното е така, тъй като при несъответствие на типа на псевдонима с типа на инициализатора, в стековата рамка на swapi, се създават “временни” променливи x и y, в които се запомнят конвертираните стойности на инициализаторите. Извършва се размяната, но само в стековата рамка на swapi.

При предаване на параметрите по указател или по псевдоним, фактическите параметри са променливи, за разлика от предаването на параметри по стойност, когато фактическите параметри могат да са изрази в общия случай.

Възможно е някои параметри да се подават по стойност, други по псевдоним или указател, а също функцията да връща резултат и чрез оператора return. Примери ще бъдат дадени в следващите части на изложението. Ще бъдат обсъдени също предимствата и недостатъците на всеки от начините за предаване на параметрите.

Ако функция не връща резултат чрез return (типът й е void), се нарича още **процедура**.

Разгледаните програми се състояха от две функции. По-сериозните приложения съдържат повече функции. Подредбата им може да започва с main, след която в произволен ред да се дефинират останалите функции. В този случай, дефиницията на main трябва да се предшества от *декларациите* на останалите функции. Декларацията на една функция се състои от заглавието й, следвано от ;. Имената на формалните параметри могат да се пропуснат. Например, програмата от Zad69_1.cpp може да се запише във вида:

```
Program Zad69_1.cpp
#include <iostream.h>
#include <iomanip.h>
void swapi(double&, double&); // декларация на swapi
int main()
{cout << "a, b, c, d= ";
 double a, b, c, d;
 cin >> a >> b >> c >> d;
 cout << setprecision(2) << setiosflags(ios :: fixed);
 cout << setw(10) << a << setw(10) << b
```

```
        << setw(10) << c << setw(10) << d << '\n';
 swapi(a, b);
 swapi(c, d);
 cout << setw(10) << a << setw(10) << b
        << setw(10) << c << setw(10) << d << '\n';
 return 0;
}
void swapi(double& x, double& y) // дефиниция на swapi
{double work = x;
 x = y;
 y = work;
 return;
}
```

3. Дефиниране на функции

Синтаксис на дефиницията

Дефиницията на функция се състои от две части: заглавие (прототип) и тяло. Синтаксисът й е показан на Фиг. 4.

```
[<модификатор>][<тип_на_функцията>]<име_на_функция>
                    (<формални_параметри>)
{<тяло>
}
където
 <модификатор>::= inline|static| ...
 <тип_на_резултата> ::= <име_на_тип> | <дефиниция_на_тип>
 <име_на_функция> ::= <идентификатор>
 <формални_параметри> :: <празно> | void |
                                        <параметър> {, <параметър>}
 <параметър> ::= <тип>[ & |opc * [const]opc] opc <име_на_параметър>
 <тип> ::= <име_на_тип>
 <име_на_параметър> ::= <идентификатор>
```

<тяло> ::= <редица_от_оператори_и_дефиниции>

Фиг. 4.

Модификаторите са спецификатори, които задават препоръка за компилатора (inline), класа памет (extern или static) и др. Ще дадем примери в следващите разглеждания. Ако е пропуснат, подразбира се extern.

Типът на функцията е произволен без масив и функционален, но се допуска да е указател към такива обекти. Ако е пропуснат, подразбира се int.

Името на функцията е произволен идентификатор. Допуска се нееднозначност.

Списъкът от формални параметри (нарича се още **сигнатура**) може да е празен или void. Например, следната функция извежда текст:

```
void printtext(void)
{cout << "This is text!!!\n"
 cout << " .....";
 return;
}
```

В случай, че е непразен, имената на параметрите трябва да са различни. Те заедно с името определят еднозначно функцията. Формалните параметри са: *параметри – стойности*, *параметри – указатели* и *параметри – псевдоними*. Името на параметъра се предшества от тип.

Примери:

int a, int const& b, double& x, int const * y, const int* a

Засега няма да използваме параметри, специфицирани със const.

Тялото на функцията е редица от дефиниции и оператори. Тя описва алгоритъма, реализиращ функцията. Може да съдържа един или повече оператора return.

Операторът return (Фиг. 5) връща резултата на функцията в мястото на извикването.

Синтаксис

return [<израз>]

където

return е запазена дума

<израз> е произволен израз от тип <тип_на_функцията> или съвместим с него. Ако типът на функцията е void, <израз> се пропуска.

Семантика

Пресмята се стойността на <израз>, конвертира се до типа на функцията (ако е възможно) и връщайки получената стойност в мястото на извикването на функцията, прекратява изпълнението й.

Фиг. 5.

Забележка: Ако функцията не е от тип void, тя задължително трябва да върне стойност. Това означава, че операторът return трябва да се намира във всички клонове на тялото. В противен случай, повечето компилатори ще изведат съобщение или предупреждение за грешка. Възможно е обаче функцията да върне случайна стойност, което е лошо. По-добре е функцията да върне някаква безобидна стойност, отколкото случайна.

Функциите могат да се дефинират в произволно място на програмата, но не и в други функции. Преди да се извика една функция, тя трябва да е "позната" на компилатора. Това става, като дефиницията на функцията се постави пред main или когато функцията се дефинира на произволно място в частта за дефиниране на функции, а преди дефинициите на функциите се постави само нейната декларация.

<декларация_на_функция> ::=
[<модификатор>][<тип_на_резултата>]<име_на_функция>
([<формални_параметри>]);

Възможно е имената на параметрите във <формални_параметри> да се пропуснат.

Семантика на дефиницията

Описанието на функция задава параметрите, които носят входа и изхода, типа на резултата, а също и алгоритъма, за реализиране на действието, което функцията дефинира. Параметрите-стойности най-често задават входа на функцията. Параметрите-указатели и псевдоними са входно-изходните параметри за нея. Алгоритъмът се описва в тялото на функцията. Изпълнението на функцията завършва при достигане на края на тялото или след изпълнение на оператор return [<израз>];.

4. Обръщение към функция

Синтаксис

<обръщение_към_функция> ::=

<име_на_функция>() |

<име_на_функция>(void) |

<име_на_функция>(<фактически_параметри>)

където <фактически_параметри> са толкова на брой, колкото са формалните параметри. Освен по брой, формалните и фактическите параметри трябра да си съответстват по тип, по вид и по смисъл.

Съответствието по тип означава, че типът на i-тия фактически параметър трябва да съвпада (да е съвместим) с типа на i-тия формален параметър. Съответствието по вид се състои в следното: ако формалният параметър е параметър-указател, съответният му фактически параметър задължително е променлива или адрес на променлива, ако е параметър-псевдоним, съответният му фактически параметър задължително е променлива (за реализацията Visual C++, 6.0 от същия тип) и ако е параметър-стойност – съответният му фактически параметър е израз.

Семантика

Обръщението към функция е унарна операция с най-висок приоритет и с операнд - името на функцията. Последното пък е указател със стойност адреса на мястото в паметта където е записан програмният код на функцията. Ако функцията определя процедура, обръщението към нея се оформя като оператор (завършва с ;). Опитът за използването й като израз предизвиква грешка. Ако функцията връща резултат както чрез return, така и чрез някой от формалните си параметри, обръщението към нея може да се разглежда и като оператор, и като израз. И ако функцията връща резултат единствено чрез оператора

return, обръщението към нея има единствено смисъла на израз. Използването му като оператор не води до грешка, но не предизвиква видим резултат.

Обръщението към функция предизвиква генериране на нова стекова рамка и се осъществява на следните два етапа:

1. Свързване на формалните с фактическите параметри

За целта първият формален параметър се свързва с първия фактически, вторият формален параметър се свързва с втория фактически и т.н. последният формален параметър се свързва с последния фактически параметър. Свързването се реализира по различни начини в зависимост от вида на формалния параметър.

а) формален параметър – стойност

В този случай се намира стойността на съответния му фактически параметър. В стековата рамка на функцията за формалния параметър се отделя толкова памет, колкото типът му изисква и в нея се откопирва стойността на фактическия параметър.

б) формален параметър – указател

В този случай в стековата рамка на функцията за формалния параметър се отделят 4B, в които се записва стойността на фактическия параметър, която е адрес на променлива. Действията, описани в тялото се изпълняват със съдържанието на формалния параметър - указател. По такъв начин е възможна промяна на стойността на променливата, чийто адреа е предаден като фактически параметър.

в) формален параметър – псевдоним

Формалният параметър-псевдоним се свързва с адреса на фактическия. За него в стековата рамка на функцията памет не се отделя. Той просто “прелита” и се “закачва” за фактическия си параметър. Действията с него се извършват над фактическия параметър.

2. Изпълнение на тялото на функцията

Аналогично е на изпълнението на блок.

При всяко обръщение към функция в програмния стек се включва нов “блок” от данни. В него се съхраняват формалните параметри на функцията, нейните локални променливи, а също и някои “вътрешни” данни като return-адреса и др. Този блок се нарича **стекова рамка на функцията.**

В дъното на стека е стековата рамка на main. На върха на стека е стековата рамка на функцията, която се обработва в момента. Под нея е стековата рамка на функцията, извикала функцията, обработваща се в момента. Ако изпълнението на една функция завършва, нейната стекова рамка се отстранява от стека.

Видът на стековата рамка зависи от реализацията. С точност до наредба, тя има вида:

Формални параметри
Адрес за връщане
Адрес на предходна рамка на стека
Локални параметри

Област на идентификаторите в програмата на C++

Идентификаторите означават имена на константи, променливи, формални параметри, функции, класове. Най-общо казано, има три вида области на идентификаторите: *глобална*, *локална* и *област за клас*. Областите се задават *неявно* – чрез позицията на идентификатора в програмата и *явно* – чрез декларация. Отново разглеждането ще е непълно, заради пропускането на класовете и явното задаване на област.

Глобални идентификатори

Дефинираните пред всички функции константи и променливи могат да се използват във всички функции на модула, освен ако не е дефиниран локален идентификатор със същото име в някоя функция на модула. Наричат се **глобални идентификатори**, а областта им – **глобална.**

Локални идентификатори

Повечето константи и променливи имат локална област. Те са дефинирани вътре във функциите и не са достъпни за кода в другите функции на модула. Областта им се определя според общото правило – започва от мястото на дефинирането и завършва в края на оператора (блока), в който идентификаторът е дефиниран. Формалните параметри на функциите също имат локална видимост. Областта им е тялото на функцията.

В различните области могат да се използват еднакви идентификатори. Ако областта на един идентификатор се съдържа в областта на друг, последният се нарича нелокален за първоначалния. В този случай е в сила правилото: Локалният идентификатор "скрива" нелокалния в областта си.

Областта на функция започва от нейното дефиниране и продължава до края на модула, в който функцията е дефинирана. Ако дефинициите на функциите са предшествани от техните декларации, редът на дефиниране на функциите в модула не е от значение – функциите са видими в целия модул. Препоръчва се също дефинирането на заглавен файл с прототипите (декларациите) на функциите.

5. Масивите като формални параметри

едномерни масиви

Съществуват различни начини за задаване на формални параметри от тип едномерен масив.

а) традиционен

Дефиницията

T a[]

където T е скаларен тип, задава параметър a от тип едномерен масив с базов тип T. Може да се укаже горна граница на масива, но компилаторът я пренебрегва.

Примери:

```
int a[]      - a е параметър от тип масив от цели числа,
int a[10]    - еквивалентна е на int a[],
double b[]   - b е параметър от тип масив от реални числа,
char c[]     - c е параметър от тип масив от символи.
```

б) чрез указател

Дефиницията

T* p

където T е скаларен тип, задава параметър p от тип указател към тип T. От връзката между масив и указател следва, че тази дефиниция може да се използва и за дефиниране на формален параметър от тип масив.

Примери: Следните дефиниции на формални параметри са еквивалентни на тези от примера по-горе:

int* a   - a е параметър от тип указател към int

double* b   - b е параметър от тип указател към double.

char* c   - c е параметър от тип указател към char.

И в двата случая фактическият параметър се указва с името на едномерен масив от същия тип. Необходимо е също на функцията да се подаде като параметър и размерът на масива.

Задача 70. Да се напишат функции, които въвеждат и извеждат елементите на едномерен масив от цели числа. Като се използват тези функции да се напише програма, която въвежда редица от естествени числа, след което я извежда, а също извежда най-големия общ делител на елементите на редицата.

Програма Zad70.cpp решава задачата.

```
Program Zad70.cpp
#include <iostream.h>
int gcd(int, int);
void readarr(int, int[]);
void writearr(int, int[]);
int main()
{cout << "n= ";
 int n;
 cin >> n;
 int a[20];
 readarr(n, a);
 writearr(n, a);
 int x = a[0];
 for (int i = 1; i <= n-1; i++)
  x = gcd(x, a[i]);
 cout << "gcd = " << x << '\n';
 return 0;
}
```

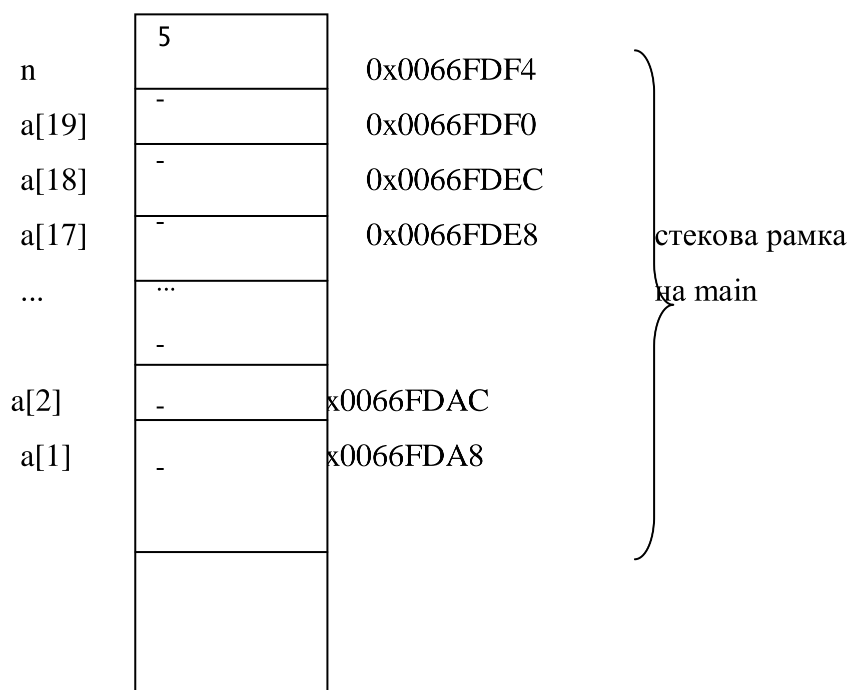
```
int gcd(int a, int b)
{while (a != b)
 if (a >= b) a = a-b; else b = b-a;
 return a;
}
void readarr(int m, int arr[])
// m е размерността на масива
// arr е едномерен масив
{for (int i = 0; i <= m-1; i++)
{cout << "arr[" << i << "]= ";
 cin >> arr[i];
}
}
void writearr(int m, int arr[])
// m е размерността на масива
// arr е едномерен масив
{for (int i = 0; i <= m-1; i++)
 cout << "arr[" << i << "]= " << arr[i] << '\n';
}
```

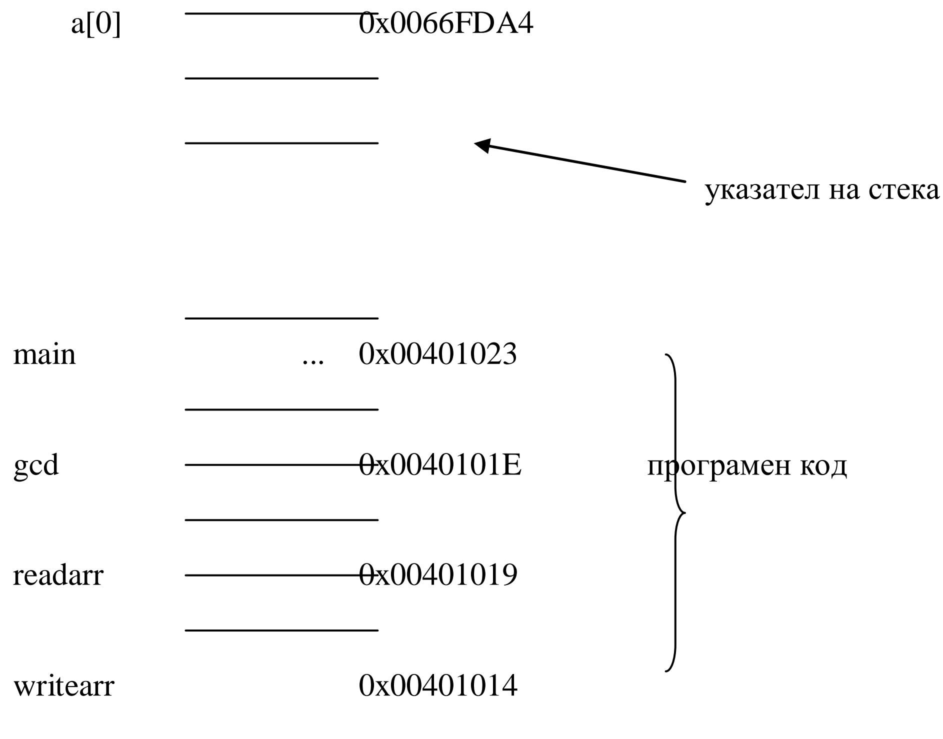
Изпълнение на програмата

Фрагментът

```
cout << "n= ";
int n;
cin >> n;
int a[20];
```

дефинира и въвежда стойност на n, а също дефинира променлива a от тип масив. Нека за n е въведено 5. В резултат е създадена стековата рамка на main. ОП до този момент има вида:





Обръщението

```
readarr(n, a);
```

се реализира като се свързват формалните с фактическите параметри и се изпълни тялото. За целта се формира нова стекова рамка – тази на readarr, в която за формалния параметър arr се отделят 4B, в която памет се откопирва стойността на фактическия параметър a (адресът на a[0]), за m се отделят също 4B, в които се откопирва 5 – стойността на фактическия параметър n. Тялото на функцията се изпълнява като блок. Операторът за цикъл

```
for (int i = 0; i <= m-1; i++)
{cout << "arr[" << i << "]= ";
 cin >> arr[i];
 }
```

е еквивалентен на

```
for (int i = 0; i <= m-1; i++)
{cout << "arr[" << i << "]= ";
 cin >> *(arr + i);
 }
```

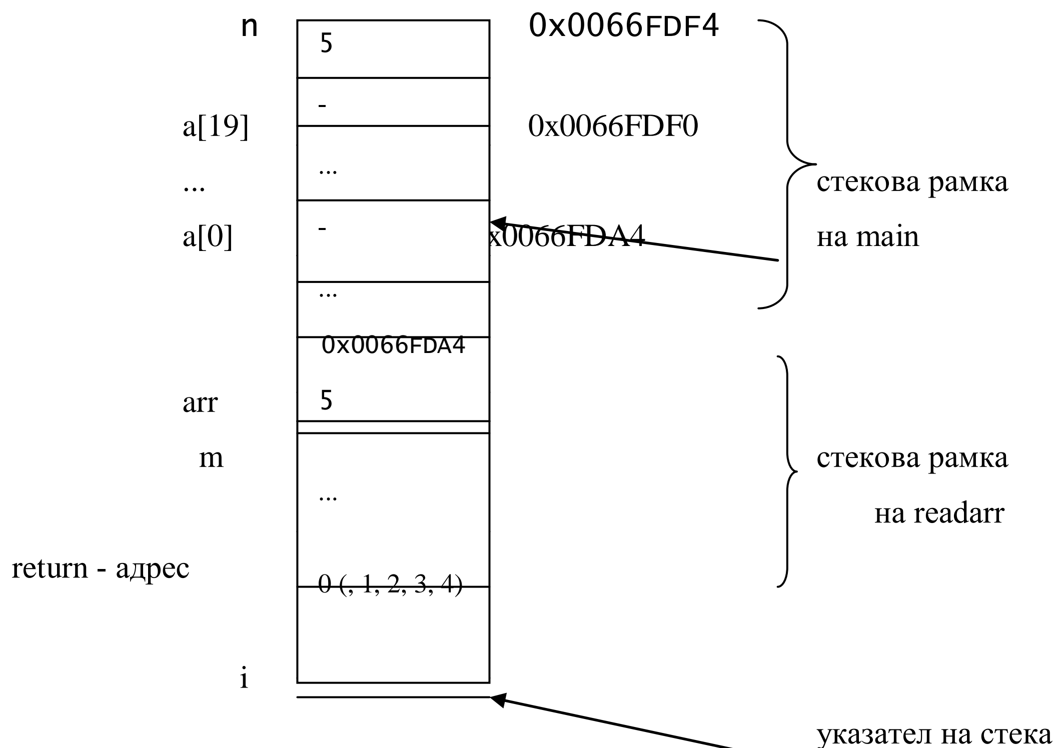
и се изпълнява по следния начин: За цялата променлива i се отделят 4B в стековата рамка на readarr. i последователно приема стойностите 0, 1, ..., 4 и за всяка стойност се изпълнява блокът

```
{cout << "arr[" << i << "]= ";
 cin >> *(arr + i);
 }
```

Операторът

```
cin >> *(arr + i);
```

въвежда стойност на индексираната променлива a[i], тъй като arr + i е адреса на i-тия елемент на a, а *(arr+i) е неговата стойност. Така във функцията се работи с формалния параметър arr, а в действителност действията се изпълняват с фактическия параметър – едномерния масив a. Функцията readarr работи с масива a, а не с негово копие.



След достигане на края на функцията изпълнението й завършва и се освобождава стековата й рамка. В резултат, първите 5 елемента на масива а получават текущи стойности. Операторът

```
writearr(n, a);
```

се изпълнява по аналогичен начин. Отново се работи с фактическия параметър - масива a, а не с негово копие. В този случай, елементите a[0], a[1], ..., a[n-1] на a само се сканират и извеждат. Не се извършват промени над тях. За да ги защитим от неправомерен достъп, е добре формалният параметър arr да дефинираме като указател към цяла константа, т.е. като const int arr[]. Тогава всеки опит да се променя arr[i] (i = 0, 1, ..., n-1) в writearr ще предизвика грешка.

Фрагментът

```
int x = a[0];
for (int i = 1; i <= n-1; i++)
   x = gcd(x, a[i]);
```

намира най-големия общ делител на елементите на редицата.

В заглавията на последните две процедури горните граници на индексите могат явно да се укажат. Например

```
void writearr(int m, int arr[20])
```

и

```
void readarr(int m, int arr[20])
```

са валидни заглавия, но компилаторът не се нуждае от горната граница. Трябват му само скобките [], за да разпознае параметър от тип масив. Може също да се използва второто представяне на формален параметър от тип масив, т.е.

```
void writearr(int m, int* arr)
```

и

```
void readarr(int m, int* arr)
```

Тези представяния на формалните параметри са напълно еквивлентни.

Забележки:

1. Функциите readarr и writearr работят с направо с масива a, а не с негови копия. Промените на елементите на масива се запазват след излизане от функцията.
2. Размерът на масивът не може да се разбере от неговото описание. Затова се налага използването на допълнителния параметър m в списъка от аргументи на функциите. Последното

не се отнася за масивите, представляващи символни низове, тъй като те завършват със знака за край на низ ‘\0’.

Задача 71. Да се напише функция len(char* s), която намира дължината на символен низ, а също функция eqstrs(char*, char*), която сравнява два символни низа за лексикографско равно.

Функциите Function Zad71_1 и Function Zad71_2 решават задачата.

```
Function Zad71_1
int len(char* s)
{int k = 0;
 while(*s)
 {k++;
  s++;}
 return k;
}

Function Zad71_2
bool eqstrs(char* str1, char* str2)
{while (*str1 == *str2 && * str1)
  {str1++; str2++;}
 return *str1 == *str2;
 }
```

Обърнете внимание, че тъй като всеки низ завършва със символа ‘\0’, който се интерпретира като false, изразът *s в оператора за цикъл while на функцията len, ще бъде истина и тялото ще се изпълнява до достигане на края на низа. Аналогична конструкция имаме и при дефиницията на функцията eqstrs.

Задача 72. Да се напише булева функция, която проверява дали цялото число x е елемент на редицата от цели числа a_0, a_1, ..., a_{n-1}.

Функцията Zad72.cpp решава задачата.

```
Function Zad72
bool search(int n, int a[], int x)
{int i = 0;
 while (a[i] != x && i < n-1)i++;
 return a[i]==x;
```

```
}
```

Обръщението

```
search(m, b, y)
```

проверява дали елементът y се съдържа в редицата b_0, b_1, ..., b_{m-1}, а

```
search(k, b + m, y)
```

проверява дали y се съдържа в подредицата b_m, b_{m+1}, ..., b_{m+k-1}.

Еквивалентна дефиниция на тази функция е:

```
bool search(int n, int* a, int x)
{int i = 0;
 while (*(a+i) != x && i < n-1)i++;
 return *(a+i)==x;
}
```

Задача 73. Да се напише функция, която проверява дали редицата от цели числа a_0, a_1, ..., a_{n-1} е монотонно намаляваща.

Функцията Zad73 решава задачата.

```
Function Zad73
bool monnam(int n, int a[])
{int i = 0;
 while (a[i] >= a[i+1] && i < n-2)i++;
 return a[i] >= a[i+1];
}
```

или

```
bool monnam(int n, int* a)
{int i = 0;
 while (*(a+i) >= *(a+i+1) && i < n-2)i++;
 return *(a+i) >= *(a+i+1);
}
```

Обръщението

```
monnam(m, b)
```

проберява дали редицата b_0, b_1, ..., b_{m-1} е монотонно намаляваща, а

```
monnam(k, b + m)
```

- дали подредицата b_m, b_{m+1}, ..., b_{m+k-1} на b_0, b_1, ..., b_{m-1} е монотонно намаляваща.

Задача 74. Да се напише функция, която проверява дали редицата от цели числа a_0, a_1, ..., a_{n-1} се състои от различни елементи.

Функцията Zad74 решава задачата.

```
Function Zad74
bool differ(int n, int a[])
{int i = -1;
 bool b; int j;
 do
 {i++; j = i;
  do
  {j++;
   b = a[i] != a[j];
  }while (b && j < n-1);
  }while (b && i < n-2);
  return b;
 }
```

Обръщението

```
differ(m, b, y)
```

проберява дали редицата b_0, b_1, ..., b_{m-1} се състои от различни елементи, а

```
differ(k, b + m)
```

- дали подредицата b_m, b_{m+1}, ..., b_{m+k-1} на b_0, b_1, ..., b_{m-1} се състои от различни елементи.

Задача 75. Да се напише програма, която въвежда две числови редици, сортира ги във възходящ ред, слива ги и извежда получената редица.

Програма Zad75.cpp решава задачата. За целта са дефинирани следните функции:

readarr – въвежда числова редица
writearr – извежда числова редица върху екрана
sortarr - сортира във възходящ ред елементите на числова редица
mergearrs - слива числови редици.

Program Zad75.cpp

```
#include <iostream.h>
#include <iomanip.h>
void writearr(int, double[]);
void readarr(int, double[]);
void sortarr(int, double[]);
void mergearrs(int, double[], int, double[], int&, double[]);

int main()
{cout << "n= ";
 int n;
 cin >> n;
 double a[20];
 readarr(n, a);
 cout << endl;
 writearr(n, a);
 cout << endl;
 sortarr(n, a);
 cout << endl;
 writearr(n, a);
 cout << "m= ";
 int m;
 cin >> m;
 double b[30];
 readarr(m, b);
 cout << endl;
 writearr(m, b);
 cout << endl;
 sortarr(m, b);
 cout << endl;
 writearr(m, b);
 cout << endl;
 int p;
 double c[50];
 mergearrs(n, a, m, b, p, c);
 writearr(p, c);
 return 0;
}
void writearr(int m, double arr[])
{cout << setprecision(3) << setiosflags(ios::fixed);
```

```
 for (int i = 0; i <= m-1; i++)
   cout << setw(10) << arr[i];
 cout << "\n";
}
void readarr(int m, double arr[])
{for (int i = 0; i <= m-1; i++)
{cout << "arr[" << i << "]= ";
 cin >> arr[i];
}
}
 void sortarr(int n, double a[])
{for (int i = 0; i <= n-2; i++)
{int k = i;
 double min = a[i];
 for (int j = i+1; j <= n-1; j++)
   if (a[j] < min)
   {min = a[j];
    k = j;
   }
 double x = a[i]; a[i] = a[k]; a[k] = x;
}
}
void mergearrs(int n, double a[], int m, double b[],
               int& k, double c[])
{int i = 0, j = 0;
 k = -1;
 while (i <= n-1 && j <= m-1)
 if (a[i] <= b[j])
 {k++;
  c[k] = a[i];
  i++;
 }
 else
 {k++;
  c[k] = b[j];
  j++;
 }
 int l;
 if (i > n-1)
```

```
    for (l = j; l <= m-1; l++)
    {k++;
     c[k] = b[l];
    }
  else
    for (l = i; l <= n-1; l++)
    {k++;
     c[k] = a[l];
    }
    k++;
 }
```

многомерни масиви

Когато многомерен масив трябва да е формален параметър на функция, в описанието му трябва да присъстват като константи всички размери с изключение на първият. Например, декларацията

void readarr2(int n, int matr[][20]);

определя matr като двумерен масив (редица от двадесеторки от цели числа). Описанието

int (*matr)[20]

е еквивалентно на

int matr[][20]

Скобките, ограждащи *matr, са задължителни. В противен случай, тъй като [] е с по-висок приоритет от *, int *matr[20] ще се интерпретира като “matr е масив с 20 елемента от тип *int”.

Задача 76. Да се напише програма, която въвежда квадратна матрица от цели числа, след което я извежда като увеличава всеки от елементите на матрицата над главния диагонал с 5 и намалява всеки от елементите под главния диагонал с 5.

Програма Zad76.cpp решава задачата. Тя дефинира функциите:

readarr2 – въвежда квадратна матрица

writearr2 – извежда квадратна матрица

transff - увеличава всеки от елементите на матрицата над главния диагонал с 5 и намалява всеки от елементите под главния диагонал с 5.

Program Zad76.cpp

```
#include <iostream.h>
#include <iomanip.h>
void readarr2(int, int[][10]);
void writearr2(int, int[][10]);
void transff(int, int[][10]);
int main()
{int a[10][10];
 cout << "n= ";
 int n;
 cin >> n;
 if (!cin)
 {cout << "Error, Bad input! \n";
  return 1;
 }
 if (n < 1 || n > 10)
 {cout << "Incorrect input! \n";
  return 1;
 }
 readarr2(n, a);
 cout << '\n';
 writearr2(n, a);
 cout << '\n';
 transff(n, a);
 writearr2(n, a);
 return 0;
}
void readarr2(int n, int arr[][10])
{for (int i = 0; i <= n-1; i++)
  for (int j = 0; j <= n-1; j++)
   cin >> arr[i][j];
}
void writearr2(int n, int arr[][10])
{for (int i = 0; i <= n-1; i++)
 {for (int j = 0; j <= n-1; j++)
   cout << setw(5) << arr[i][j];
  cout << "\n";
 }
}
void transff(int n, int arr[][10])
```

```
{int i, j;
 for (i = 1; i <= n-1; i++)
  for (j = 0; j <= i-1; j++)
     arr[i][j] = arr[i][j] - 5;
 for(i = 0; i <= n-2; i++)
  for(j = i+1; j <= n-1; j++)
     arr[i][j] = arr[i][j] + 5;
}
```

Обръщението

```
transff(k, a + m);
```

ще извърши същото действие над квадратната подматрица на дадената матрица:

$$\begin{pmatrix} a_{m,0} & a_{m,1} & ... & a_{m,\ k-1} \\ a_{m+1,\ 0} & a_{m+1,1} & ... & a_{m+1,k-1} \\ ... & ... & ... & ... \\ a_{m+k-1,0} & a_{m+k-1,1} & ... & a_{m+k-1,k-1} \end{pmatrix}$$

Задача 77. Да се напише програма, която въвежда редица от думи не по-дълги от 14 знака и дума, също не по-дълга от 14 знака. Програмата да проверява дали думата се среща в редицата. За целта да се оформят подходящи функции.

Програма Zad77.cpp решава задачата. В нея са дефинирани функциите:

void readarrstr(int n, char s[][15]); - въвежда редица от n думи,

bool search(int n, char s[][15], char* x); - търси думата x в редицата s от n думи. За целта използва помощната функция

bool eqstrs(char* str1, char* str2);

от Задача 71.

```
Program Zad77.cpp
#include <iostream.h>
#include <string.h>
void readarrstr(int, char [][15]);
bool eqstrs(char*, char*);
bool search(int, char [][15], char*);
```

```
int main()
{char a[20][15];
 cout << "n= ";
 int n;
 cin >> n;
 readarrstr(n, a);
 cout << "word: ";
 char word[15];
 cin >> word;
 if (search(n, a, word)) cout << "yes \n";
 else cout << "no \n";
 return 0;
}
void readarrstr(int n, char s[][15])
{for(int i = 0; i <= n-1; i++)
 {cout << "s[" << i << "]= ";
  cin >> s[i];
 }
}
bool eqstrs(char* str1, char* str2)
 {while (*str1 && *str1 == *str2)
 {str1++; str2++;}
  if(*str1 != *str2) return false;
  else return true;
 }
bool search(int n, char s[][15], char* x)
{int i = 0;
 while (!eqstrs(s[i], x) && i < n-1) i++;
 return eqstrs(s[i], x);
}
```

Задача 78. Да се напише програма, която умножава две матрици.

Програма Zad78.cpp решава задачата. Тя дефинира следните функции:

readarr2 – въвежда матрица,

writearr2 – извежда матрица,

multmatr – умножава матрици.

Program Zad78.cpp

```
#include <iostream.h>
#include <iomanip.h>
void readarr2(int n, int m, double [][30]);
void writearr2(int n, int m, double [][30]);
void multmatr(int, int, int, double [][30],
           double [][30], double [][30]);

int main()
{double a[10][30], b[20][30], c[10][30];
 cout << "n= ";
 int n;
 cin >> n;
 if (!cin)
 {cout << "Error, Bad input! \n";
  return 1;
 }
 if (n < 1 || n > 10)
 {cout << "Incorrect input! \n";
  return 1;
 }
 cout << "m= ";
 int m;
 cin >> m;
 if (!cin)
 {cout << "Error, Bad input! \n";
  return 1;
 }
 if (m < 1 || m > 30)
 {cout << "Incorrect input! \n";
  return 1;
 }
  cout << "k= ";
 int k;
 cin >> k;
 if (!cin)
 {cout << "Error, Bad input! \n";
  return 1;
 }
```

```
 if (k < 1 || k > 30)
 {cout << "Incorrect input! \n";
  return 1;
 }
 readarr2(n, m, a);
 writearr2(n, m, a);
 cout << "\n";
 readarr2(m, k, b);
 cout << "\n";
 writearr2(m, k, b);
 cout << "\n";
 multmatr(n, m, k, a, b, c);
 writearr2(n, k, c);
 return 0;
}
void readarr2(int n, int m, double arr [][30])
{for (int i = 0; i <= n-1; i++)
   for (int j = 0; j <= m-1; j++)
      cin >> arr[i][j];
 return;
}
void writearr2(int n, int m, double arr[][30])
{cout << setprecision(3) << setiosflags(ios::fixed);
 for (int i = 0; i <= n-1; i++)
 {for (int j = 0; j <= m-1; j++)
   cout << setw(10) << arr[i][j];
  cout << "\n";
 }
 return;
}
void multmatr(int n, int m, int k, double a[][30],
              double b[][30], double c[][30])
{for (int i = 0; i <= n-1; i++)
  for (int j = 0; j <= m-1; j++)
  {c[i][j] = 0;
   for (int p = 0; p <= m-1; p++)
      c[i][j] += a[i][p] * b[p][j];

      //c[I][j] = c[I][j]+a[I][j]* b[p][j]
```

```
  }
}
```

6. Масивите като върнати оценки

Въпреки, че масивите могат да са параметри на функции, функциите **не могат** да са от тип масив. Възможно е обаче да са от тип указател. Това позвлява дефинирането на функции, които връщват масиви.

Пример: В следващата програма е дефинирана функцията readarr, която въвежда стойности на едномерен масив. Тя връща резултат не само чрез променливата от тип масив arr, но и чрез оператора return. Това позволява обръщенията към нея да са както като оператор, така и като израз.

```
#include <iostream.h>
void writearr(int, int[]);
int* readarr(int, int[]);
int main()
{cout << "n= ";
 int n;
 cin >> n;
 int a[20];
 int* p = readarr(n, a);
 writearr(n, p);
 cout << endl;
 return 0;
}
void writearr(int m, int arr[])
{for (int i = 0; i <= m-1; i++)
 cout << "arr[" << i << "]= " << arr[i] << '\n';
return;
}
int* readarr(int m, int arr[])
{for (int i = 0; i <= m-1; i++)
 {cout << "arr[" << i << "]= ";
  cin >> arr[i];
 }
```

```
return arr;
}
```

Въпрос: Допустима ли е конструкцията: readarr(n, a)[i], където i е цяло число от 0 до n-1? Ако това е така, какъв е резултатът от изпълнението му?

Задача 79. Да се напише функция, която намира и връща като резултат конкатенацията на два низа. Функцията да променя първия си аргумент като в резултат той да съдържа конкатенацията на низовете.

Програма Zad79.cpp решава задачата.

```
Program Zad79.cpp
#include <iostream.h>
int len(char*);
char *cat(char*, char*);
int main()
{char s1[100];
 cout << "s1= ";
 cin >> s1;
 cout << "s2= ";
 char s2[100];
 cin >> s2;
 cout << cat(s1, s2) << "  " << s1 << '\n';
 return 0;
}
int len(char* s)
{int k = 0;
 while (*s)
 {k++; s++;
 }
 return k;
}
char* cat(char *s1, char *s2)
{int i = len(s1);
 while (*s2)
 {s1[i] = *s2;
  i++;
  s2++;
```

```
}
 s1[i] = '\0';
 return s1;
}
```

Задачи

Задача 1. Въпреки многото й недостатъци, следващата програма е доста поучителна. Тя дефинирана функцията readarr, която има за формален параметър броя на елементите на масива и връща едномерен масив, определен чрез указател към първия му елемент.

```
#include <iostream.h>
int a[20];
void writearr(int, int[]);
int* readarr(int);
int main()
{cout << "n= ";
 int n;
 cin >> n;
 int* p = readarr(n);
 writearr(n, p);
 cout << ‘\n’;
 return 0;
}
void writearr(int m, int arr[])
{for (int i = 0; i <= m-1; i++)
 cout << "arr[" << i << "]= " << arr[i] << '\n';
}
int* readarr(int m)
{for (int i = 0; i <= m-1; i++)
{cout << "a[" << i << "]= ";
 cin >> a[i];
}
return a;
}
```

извършете експерименти с тази програма.

Задача 2. Да се напише програма, която въвежда полиномите:

$$P_n(x) = a_o x^{n-1} + a_1 x^{n-2} + \ldots + a_{n-2} x^1 + a_{n-1},$$

$$Q_m(x) = b_o x^{m-1} + b_1 x^{m-2} + \ldots + b_{m-2} x^1 + b_{m-1},$$

$$R_k(x) = r_o x^{k-1} + r_1 x^{k-2} + \ldots + r_{k-2} x^1 + r_{k-1}$$

и реалната променлива x и намира и извежда стойностите на полиномите в x.

Задача 3. Да се напише програма, която въвежда стойности на редиците:

$a_0, a_1, \ldots, a_{n-1}$,
$b_0, b_1, \ldots, b_{m-1}$,
$c_0, c_1, \ldots, c_{p-1}$

и намира и извежда AR1, AR2, BR1, BR2, CR1 и CR2, където за дадена редица $x_0, x_1, \ldots, x_{k-1}$

$$XR1 = \frac{1}{k}\sum_{i=0}^{k-1} x_i, \qquad XR2 = \frac{1}{k}\sqrt{\sum_{i=0}^{k-1}(x_i - XR1)^2}.$$

Задача 4. Да се напише функция, която намира разстоянието между две точки в равнината, зададени чрез координатите си (x1, y1) и (x2, y2). Като се използва тази функция да се напише програма, която чете координатите на n точки ($n \geq 1$) от равнината и намира и извежда разстоянието между всеки две от тях.

Задача 5. да се напише функция, която връща стойност true, ако a, b и c са страни на триъгълник и false - в противен случай. Като се използва тази функция, да се напише програма, която въвежда стойности на елементите на матрицата A_{3xn} и определя кои от тройките (a[0][i], a[1][i], a[2][i]), i = 0, 1, ..., n-1 могат да служат за страни на триъгълник.

Задача 6. Да се напише функция, която връща стойност true ако редицата от цели числа $x_0, x_1, \ldots, x_{k-1}$ има поне два последователни нулеви елемента. Като се използва тази функция, да се напише програма, която намира и извежда номерата на редовете на матрицата A [n x n], от цели числа, които имат поне два последователни нулеви елемента.

Задача 7. Да се напише функция, която намира сумата на два полинома. Като се използва тази функция, да се напише програма, която намира сумата на всеки два от полиномите:

$$Q_m(x) = b_{n-1} x^{m-1} + b_{n-2} x^{m-2} + \ldots + b_1 x^1 + b_0,$$

$$P_n(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + ... + a_1x^1 + a_0,$$

Задача 8. Даден е триъгълник със страни a, b и c. Да се напише

$$R_k(x) = r_{k-1}x^{k-1} + r_{k-2}x^{k-2} + ... + r_1x^1 + r_0,$$

програма, която намира медианите на триъгълник, страните на който са медианите на дадения триъгълник.

Упътване: Медианата към страната a на триъгълника е равна на

$$\frac{1}{2}\sqrt{2b^2 + 2c^2 - a^2}.$$

Задача 9. Дадени са координатите на върховете на n триъгълника. Да се напише програма, която определя, кой от триъгълниците е с по-голямо лице.

Задача 10. Дадени са естественото число p > 1 и реалните квадратни матрици с размерности n x n – A, B и C. Да се напише програма, която намира матрицата

$$(A . B . C)^p.$$

Допълнителна литература

1. Ст. Липман, Езикът C++ в примери, “КОЛХИДА ТРЕЙД” КООП, С. 1993.
2. И. Момчев, К. Чакъров, Програмиране III, C и C++, ТУ, С. 1996.
3. Д. Луис, C/C++ бърз справочник, ИнфоДАР, С. 1998.
4. B. Stroustrup, C++ Programming Language. Third Edition, Addison – Wesley, 1997.

Глава 9

Програмиране от по-висок ред в C++

Функция, някои формални параметри на която са функции, се нарича **функция от по-висок ред.**

В езика C++ е възможно формален параметър на функция да е указател към функция, а също е възможно резултатът от изпълнението на функция да е указател към функция. Това позволява да се реализират функции от по-висок ред, а също такива които връщат функция.

1. Указател към функция

Името на функция е константен указател, сочещ към първата машинна инструкция от изпълнимия й машинен код. В езика C++ е възможно да се дефинират променливи, които са указатели към функции (Фиг. 1).

<дефиниция_на_променлива_указател_към_функция> ::=
<тип_на_функция>(*<указател_към_функция>)(<формални_параметри>)
[= <име_на_функция>];

където

- <указател_към_функция> е идентификатор;
- <име_на_функция> е идентификатор, означаващ име на функция от тип <тип_на_функция> и параметри - <формални_параметри>;
- <тип_на_функция> и <формални_параметри> са аналогични на съответните от заглавието на дефиниция функция. Имената на параметрите могат да се пропуснат.

Фиг. 1.

Забележка: Скобите, ограждащи *<указател_към_функция>, са задължителни. В противен случай дефиницията ще се изтълкува от компилатора като декларация на функция с име < указател_към_функция>, с параметри - <формални_параметри> и тип – указател към <тип_на_функция>.

В резултат на дефиницията на променлива от тип указател към функция, за променливата се отделят 4B ОП, която е с неопределена стойност, ако дефиницията е без инициализация, и съдържа адреса на първата машинна команда от изпълнимия код на функцията, чрез която е направена инициализацията, ако дефиницията е с инициализация.

Примери:

1. double (*p)(double, double);

е дефиниция на променлива p от тип указател към функция от тип double с два аргумента също от тип double. В резултат за p се отделят 4B ОП, които са с неопределена стойност.

2. int (*q)(int, int*);

дефинира променлива q от тип указател към функция от тип int, с два аргумента, единият от които цял, а другият – указател към int. За q се отделят 4B ОП, които са с неопределена стойност.

3. Нека са дефинирани следните функции за сортиране на числови редици:

```
void bubblesort(int, int*);   // метод на мехурчето
void mergesort(int, int*);    // сортиране чрез сливане
void heapsort(int, int*);   // пирамидално сортиране
```

Променливата r може да е указател към тези функции ако е дефинирана по следния начин:

```
void (*r)(int, int*);
```

r не може да е указател към функциите:

```
int f1(int, int*);
int f2(int, int*);
```

Указател към последните може да е променливата s, където:

```
int (*s)(int, int*);
```

Горните дефиниции на p, q, r и s са без инициализации.

Дефиниците на променливите x и y

```
void (*x)(int, int*) = bubbesort;
int (*y)(int, int*) = f2;
```

са с инициализация. За всяка от тях се отделят 4B ОП, в която памет се записват адресите на първите команди на изпълнимите кодове на bubblesort и f2 съответно.

Присвояването се извършва по стандартния начин.

Пример:

```
r = mergesort;
x = heapsort;
```

След инициализация на променлива от тип указател към функция, чрез променливата може да се осъществи обръщение към конкретна функция. Така се предоставя ефективен способ за предаване на управлението към потребителски и библиотечни функции.

Обръщението към функция освен директно може да се осъществява и индиректно – чрез указател към нея.

Пример:

```
void (*r)(int, int*) = bubblesort;
bubblesort(n, a);       // директно обръщение
(*r)(n, a);            // индиректно обръщение, чрез r.
```

Забележка: Някой компилатори, в това число и на Visual C++ 6.0, допускат извикването на функция чрез указател да се осъществява и само чрез името на указателя.

Пример:

```
void (*r)(int, int*) = bubblesort;
r(n, a);                // индиректно обръщение към bubblessort,
                        // чрез r.
```

2. Функциите като формални параметри

Указател към функция може да е формален параметър на функция. Ще илюстрираме тази възможност с няколко примери.

Задача 80. Да се напише функция, която реализира математическата абстракция:

$$\sum_{\substack{i=a \\ i\to next(i)}}^{b} f(i).$$

където a и b са дадени реални числа (a ≤ b), f е реална едноаргументна функция, задаваща терма, а next е реална едноаргументна функция, задаваща стъпката за промяна на управляващия параметър на сумата.

За да решим задачата ще предложим няколко частни решения.

а) Да се дефинира функция, която намира стойността на сумата:

sin(a) + sin(a+1) + sin(a+2) + ... + sin(b),

където a и b са дадени реални числа. Функцията sum_sin решава задачата.

```
double sum_sin(double a, double b)
{double s = 0;
 for (double i = a; i <= b + 1E-14; i = i + 1)
    s = s + sin(i);
 return s;
}
```

б) Да се дефинира функция, която намира стойността на сумата:

cos(a) + cos(a + 0.2) + cos(a + 0.4) + ... + sin(b)

където a и b са дадени реални числа. Функцията sum_cos решава задачата.

```
double sum_cos(double a, double b)
{double s = 0;
 for (double i = a; i <= b + 1e-14; i = i + 0.2)
     s = s + cos(i);
 return s;
}
```

Забелязваме, че тези две функции се "приличат". Написани са по следния общ шаблон:

```
double <name>(double a, double b)
{double s = 0;
 for (double i = a; i <= b + 1e-14; i = <next>(i))
    s = s + <f>(i);
 return s;
}
```

Елементите, по които функциите sum_sin и sum_cos се различават са означени с <...> в шаблона. Това са две функции – f, означаваща терма и next - стъпката на сумата. Като използваме възможността формален параметър на функция да е указател към функция, можем да изнесем <f> и <next> като формални параметри на функцията и да обобщим тези частни случаи. Така стигаме до функцията sum:

```
Function Zad80.cpp
double sum(double a, double b, double (*f)(double),
           double (*next)(double))
{double s = 0;
 for (double i = a; i <= b + 1e-14; i = next(i))
    s = s + f(i);
 return s;
}
```

Обръщенията към sum:

```
sum(a, b, sin, next1)
sum(a, b, cos, next2)
```

където

```
int next1(double x)
{return x + 1;
}
int next2(double x)
{return x + 0.2;
}
```

реализират горните частни случаи.

sum е функция от по-висок ред. В нея третият и четвъртият параметри са указатели към функции.

Като използваме sum, може да дефинираме sum_sin и sum_cos по следния начин:

```
double sum_sin(double a, double b)
{return sum(a, b, sin, next1);
}
double sum_cos(double a, double b)
{return sum(a, b, cos, next2);
}
```

Задача 81. Да се напише функция, която реализира математическата абстракция:

$$\prod_{\substack{i=a \\ i->next(i)}}^{b} f(i)$$

където a и b са реални числа, f е реална едноаргументна функция, задаваща терма, а next - реална едноаргументна функция, задаваща стъпката за промяна на управляващия параметър на произведението. Да се включи тази функция в програма и се намерят:

tg(1) * tg(1.5) * tg(2) * tg(2.5) * tg(3)

и

arctg(1) * arctg(1.1) * arctg(1.2) * arctg(1.3).

Програма Zad81.cpp решава задачата.

```
Program Zad81.cpp
#include <iostream.h>
#include <math.h>
double prod(double, double, double (*)(double),
            double (*) (double));
double next1(double);
double next2(double);
int main()
{cout << prod(1, 3, tan, next1) << '\n';
 cout << prod(1, 1.3, atan, next2) << '\n';
 return 0;
}
double prod(double a, double b, double (*f)(double),
            double (*next)(double))
{double s = 1.0;
 for (double i = a; i <= b + 1e-14; i = next(i))
    s = s * f(i);
 return s;
}
double next1(double x)
{return x + 0.5;
}
```

```
double next2(double x)
{return x + 0.1;
}
```

В тази програма е дефинирана функцията от по-висок ред prod, реализираща исканата абстракция. В нея третият и четвъртият параметри са указатели към функции. В главната програма са направени две обръщения към нея

```
prod(1, 3, tan, next1)
```

и

```
prod(1, 1.3, atan, next2),
```

намиращи търсените произведения.

Забелязваме, че функциите sum и prod си “приличат”. Написани са по следния общ шаблон.

```
double <name>(double a, double b, double (*f)(double),
              double (*next)(double))
{double s = <null_val>;
 for (double i = a; i <= b + 1e-14; i = next(i))
    s = s <op> f(i);
 return s;
}
```

И в този случай, елементите, по които sum и prod се различават са оградени с <...>. Това са операцията op и нулата на операцията - null_val. Отново ще изнесем op и null_val като формални параметри на функцията. Тъй като op е бинарна инфиксна операция, а не име на функция, ще дефинираме помощна реална функция с име op, с два реални параметъра и връщаща резултата от прилагането на операцията op към аргументите на функцията op. Така получаваме още едно обобщение на горните абстракции – функцията от по-висок ред accumulate (Задача 82).

Задача 82. Да се напише програма, която реализира следната математическа абстракция:

$$f(a) \otimes f(next(a)) \otimes f(next(next(a)) \otimes \ldots \otimes f(b)$$

където с ⊗ е означена произволна бинарна целочислена операция, а f и next имат смисъла, определен в предходните две задачи.

Програма Zad82.cpp решава задачата.

```
Program Zad82.cpp
#include <iostream.h>
#include <math.h>
double accumulate(double (*) (double, double),
                  double, double, double,
                  double (*)(double), double (*) (double));
double plus(double, double);
double mult(double, double);
double next1(double);
double next2(double);
int main()
{cout << "a, b= ";
 double a, b;
 cin >> a >> b;
 if (!cin)
 {cout << "Error! \n";
  return 1;
 }
 cout << accumulate(plus, 0, a, b, cos, next1) << '\n';
 cout << accumulate(mult, 1, a, b, sin, next2) << '\n';
 return 0;
}

double accumulate(double (*op)(double, double),
                  double null_val, double a, double b,
                  double (*f)(double), double (*next)(double))
{double s = null_val;
 for (double i = a; i <= b +1e-14; i = next(i))
    s = op(s, f(i));
 return s;
```

```
}

double next1(double x)
{return x + 1;
}
double next2(double x)
{return x + 2;
}
double plus(double x, double y)
{return x + y;
}
double mult(double x, double y)
{return x * y;
}
```

Функцията accumulate е функция от по-висок ред. Нейните първи, пети и шести формални параметри са указатели към функции, задаващи съответно операцията op, терма f и стъпката next.

В тази програма са направени две обръщения към функцията accumulate, които намират:

cos(a) + cos(a+1) + cos(a+2) + ... + cos(b)

и

sin(a) * sin(a+2) * sin(a+4) * ... * sin(b)

съответно.

Чрез accumulate, функциите sum и prod могат да се дефинират по следния начин:

```
double sum(double a, double b, double (*f)(double),
           double (*next)(double))
{return accumulate(plus, 0, a, b, f, next);
}
double prod(double a, double b, double (*f)(double),
            double (*next)(double))
{return accumulate(mult, 1, a, b, f, next);
}
```

където plus и mult са дефинирани в програмата.

Използването на променливи, които са указатели към функции, усложнява записа на дефиницията на функция. Добре би било да дадем имена на типовете указател към функция и вместо дефиницията да използваме името на типа. Задаването на имена на типове може да се

осъществи чрез оператора typedef. На Фиг. 2 са дадени синтаксисът и семантиката на този оператор.

Оператор typedef

Синтаксис

typedef <тип> <име>;

където

<тип> е дефиниция на тип;

<име> е идентификатор, определящ името на новия тип.

Семантика

Определя <име> за синоним на типа от <тип>.

Фиг. 2

Примери:

```
typedef unsigned char BYTE; // BYTE е синоним на unsigned char
typedef double REAL;  // REAL е синоним на double
```

Задаването на алтернативно име на тип указател към функция чрез typedef се осъществява по аналогичен начин на дефиниране на променлива от тип указател към функция като новото име на типа заема мястото на променливата.

Примери:

1. typedef double(*mytype)(double);

определя mytype като синоним на типа double (*)(double);

2. typedef double(*newtype)(double, double)

определя newtype като синоним на типа double (*)(double, double).

Като използваме оператора typedef и дефинираме:

```
typedef double (*type1) (double, double);
typedef double (*type2) (double);
```

декларацията на функцията accumulate от Zad82.cpp

```
double accumulate(double (*) (double, double),
                  double, double, double,
                  double (*)(double), double (*) (double));
```

може да се запише по следния начин:

```
double accumulate(type1, double, double, double, type2, type2);
```

Задача 83. Като се направи подходяща модификация на accumulate, да се напише програма, която по дадени естествено число n и реално число x, намира сумата:

$$\sum_{i=0}^{n} \frac{x^i}{i!}$$

Програма Zad83.cpp решава задачата. В нея a и b са 0 и n съответно. Термът f е:

$$f: i \longrightarrow \frac{x^i}{i!},$$

а стъпката се задава от:

$$next: i \longrightarrow i + 1.$$

```
Program Zad83.cpp
Program Zad83.cpp
#include <iostream.h>
#include <math.h>
typedef double (*type1) (double, double);
typedef double (*type2)(int);
typedef int (*type3) (int);
double x;
double accumulate(type1, double, int, int, type2, type3);
double f(int);
int next(int);
double sum(double, double);
int main()
{cout << "n= ";
 int n;
 cin >> n;
 if (!cin || n < 0)
 {cout << "Error! \n";
  return 1;
 }
 cout << "x= ";
 cin >> x;
```

```
 if (!cin)
 {cout << "Error! \n";
  return 1;
 }
 cout << accumulate(sum, 0, 0, n, f, next) << '\n';
 return 0;
}
double f(int i)
{int p = 1;
 for (int j = 1; j <= i; j++) p = p*j;
  return pow(x, i)/p;
}
int next(int x)
{return x + 1;
}
double sum(double x, double y)
{return x + y;
}
double accumulate(type1 op, double null_val,
                  int a, int b, type2 f, type3 next)
{double s = null_val;
 for (int i = a; i <= b; i = next(i))
    s = op(s, f(i));
 return s;
}
```

Направената модификация на accumulate е свързана с промяната на типовете на a, b, на f и next.

3. Функциите като върнати оценки

Функция може да върне като резултат указател към друга функция. Например, декларацията

```
int (*fun(int, int))(int*, int);
```

определя функцията fun с два цели аргумента и връщаща указател към функция от тип

```
int (*)(int*, int).
```

Ако зададем име на този тип чрез typedef, този запис може да се опрости:

```
typedef int (*fun-point)(int*, int);
fun-point fun(int, int);
```

Задача 84. Да се напише програма, която по зададено реално число x и символ (a, b, c или d) избира за изпълнение функция, определена чрез зависимостта:

$$y = \begin{cases} \sin(x) & \longrightarrow a \\ \cos(x) & \longrightarrow b \\ \exp(x) & \longrightarrow c \\ \log(x) & \longrightarrow d. \end{cases}$$

Програма Zad84.cpp решава задачата.

```
Program Zad84.cpp
#include <iostream.h>
#include <math.h>
typedef double (*f_type)(double);
f_type table(char ch)
{switch(ch)
{case 'a': return sin; break;
 case 'b': return cos; break;
 case 'c': return exp; break;
 case 'd': return log; break;
 default: cout << "Error \n"; return tan;
}}
int main()
{char ch;
 cout << "ch= ";
 cin >> ch;
 if (ch < 'a' || ch >'d') cout << "Incorrect input! \n";
 else
 {double x;
 cout << "x= ";
 cin >> x;
 cout << table(ch)(x) << '\n';
 }
```

```
 return 0;
}
```

Илюстрираните в тази част възможности на езика C++ показват, че данните от тип функции съществено не се отличават от другите видове данни. Това показва високата степен на унифицираност в езика и води до увеличаване на изразителната му сила.

Задачи

Задача 1. Като използвате функцията от по-висок ред sum,

$$S = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{3!} + \ldots + (-1)^n \frac{x^{(2n+1)}}{(2n+1)!}$$

намерете:

Задача 2. ***Като използвате функцията от по-висок ред prod, намерете:***

$$S = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots + (-1)^n \frac{x^{2n}}{(2n)!}$$

а) x^n, където x е дадено реално, а n – дадено естествено число.

б) n!, където n е дадено естествено число.

в) броят на вариациите от n елемента от k-ти клас (n и k са дадени естествени числа, $0 \leq k \leq n$).

г) броят на комбинациите от n елемента от k-ти клас (n и k са дадени естествени числа, $0 \leq k \leq n$).

Допълнителна литература

1. Ст. Липман, Езикът C++ в примери, “КОЛХИДА ТРЕЙД” КООП, С. 1993.
2. Д. Луис, C/C++ бърз справочник, ИНФОДАР, С. 1998.
3. М. Тодорова, Езици за функционално и логическо програмиране – функционално програмиране, СОФТЕХ, С., 1998.

4. B. Stroustrup, C++ Programming Language. Third Edition, Addison – Wesley, 1997.

Глава 10

Рекурсия

1. Рекурсивни функции в математиката

Ако в дефиницията на някаква функция се използва самата функция, дефиницията на функцията е рекурсивна.

Примери:

а) Ако n е произволно естествено число, следната дефиниция на функцията факториел

$$n! = \begin{cases} 1, & n = 0 \\ n.(n-1)!, & n > 0 \end{cases}$$

е рекурсивна. Условието при n = 0 не съдържа обръщение към функцията факториел и се нарича **гранично.**

б) Функцията за намиране на най-голям общ делител на две естествени числа a и b може да се дефинира по следния рекурсивен начин:

$$gcd(a, b) = \begin{cases} a, & a = b \\ gcd(a-b, b), & a > b \\ gcd(a, b-a), & a < b \end{cases}$$

Тук граничното условие е условието при a = b.

в) Ако x е реално, а n – цяло число, функцията за степенуване може да се дефинира рекурсивно по следния начин:

$$x^n = \begin{cases} x.x^{n-1}, & n > 0 \\ 1, & n = 0 \\ \dfrac{1}{x^{-n}}, & n < 0 \end{cases}$$

В този случай граничното условие е условието при n = 0.

г) Редицата от числата на Фибоначи

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

може да се дефинира рекурсивно по следния начин:

$$fib(n) = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ fib(n-1) + fib(n-2), & n > 1 \end{cases}$$

В този случай имаме две гранични условия – при n = 0 и при n = 1.

Рекурсивната дефиниция на функция може да се използва за намиране стойността на функцията за даден допустим аргумент.

Примери:

а) Като се използва рекурсивната дефиниция на функцията факториел може да се намери факториелът на 4. Процесът за намирането му преминава през разширение, при което операцията умножение с отлага, до достигане на граничното условие 0! = 1. Следва свиване, при което се изпълняват отложените операции.

4! =

4.3! =

4.3.2! =

4.3.2.1! =

4.3.2.1.0! =

4.3.2.1.1 =

4.3.2.1 =

4.3.2 =

4.6 =

24

б) Като се използва рекурсивната дефиниция на функцията gcd, може да се намери gcd(35, 14).

gcd(35, 14) =

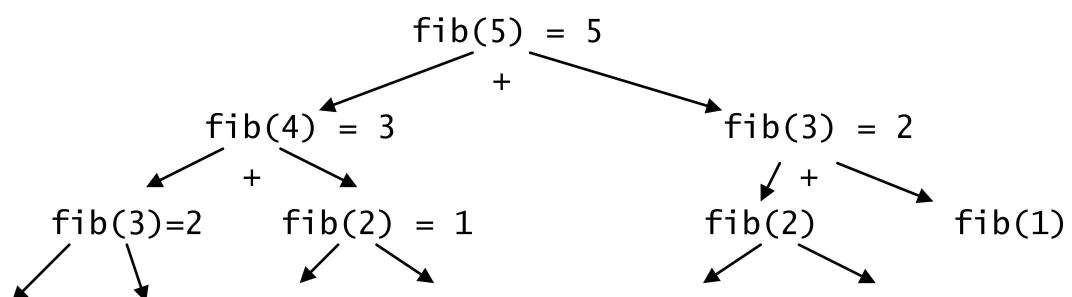
```
gcd(21, 14) =
gcd(7, 14) =
gcd(7, 7) =
7.
```

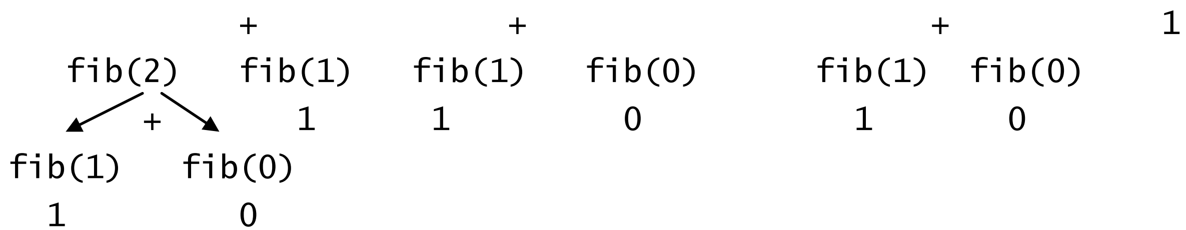
В този случай няма разширяване, нито пък свиване. Двата аргумента на gcd играят ролята на натрупващи параметри. Те се променят докато се достигне граничното условие.

в) Като се използва рекурсивната дефиниция на функцията за степенуване може да се намери стойността на 2^{-4}.

$$2^{-4} =$$
$$\frac{1}{2^4} =$$
$$\frac{1}{2.2^3} =$$
$$\frac{1}{2.2.2^2} =$$
$$\frac{1}{2.2.2.2.2^0} =$$
$$\frac{1}{2.2.2.2.1} =$$
$$\frac{1}{2.2.2.2}$$
$$\frac{1}{2.2.4} =$$
$$\frac{1}{2.8} =$$
$$\frac{1}{16}$$

г) Чрез рекурсивната дефиниция на функцията, генерираща редицата на Фибоначи, може да се намери fib(5).





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

2. Рекурсивни функции в C++

Известно е, че в тялото на всяка функция може да бъде извикана друга функция, която е дефинирана или е декларирана до момента на извикването ѝ. Освен това, в C++ е вграден т. нар. механизъм на рекурсия – разрешено е функция да вика в тялото си самата себе си.

Функция, която се обръща пряко или косвено към себе си, се нарича рекурсивна. Програма, съдържаща рекурсивна функция е **рекурсивна**.

Чрез пример ще илюстрираме описанието, обръщението и изпълнението на рекурсивна функция.

Пример за рекурсивна програма

Задача 85. Да се напише рекурсивна програма за намиране на m! (m е дадено естествено число).

Програма Zad85.cpp решава задачата.

```
Program Zad85.cpp
#include <iostream.h>
int fact(int);
int main()
{cout << "m= ";
 int m;
 cin >> m;
 if(!cin || m < 0)
```

```
 {cout << “Error! \n”;
  return 1;
 }
 cout << m << "!= " << fact(m) << '\n';
 return 0;
}
int fact(int n)
{if (n == 0) return 1;
else return n * fact(n-1);
}
```

В тази програма е описана рекурсивната функция fact, която приложена към естествено число, връща факториела на това число. Стойността на функцията се определя посредством обръщение към самата функция в оператора return n * fact(n-1);.

Изпълнение на програмата

Изпълнението започва с изпълнение на главната функция. Фрагментът

```
cout << "m= ";
int m;
cin >> m;
```

въвежда стойност на променливата m. Нека за стойност на m е въведено 4. В резултат в стековата рамка на main, отделените 4B за променливата m се инициализират с 4. След това се изпълнява операторът

```
cout << m << "!= " << fact(m) << '\n';
```

За целта трябва да се пресметне стойността на функцията fact(m) за m равно на 4, след което получената стойност да се изведе. Обръщението към функцията fact е илюстрирано по-долу:

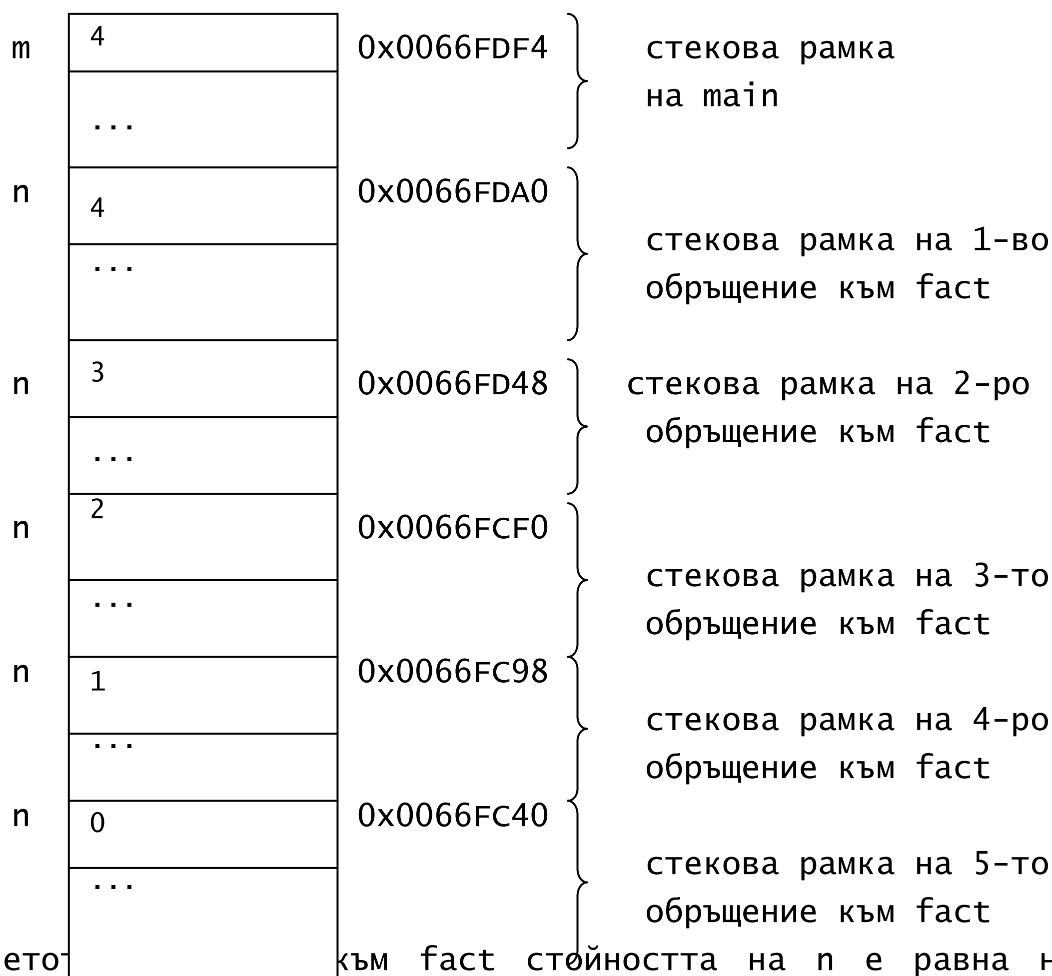
Генерира се стекова рамка за това обръщение към функцията fact. В нея се отделят 4B ОП за фактическия параметър n, в която памет се откопирва стойността на фактическия параметър m и започва изпълнението на тялото на функцията. Тъй като n е различно от 0, изпълнява се операторът

```
return n * fact(n-1);
```

при което трябва да се намери fact(n-1), т.е. fact(3). По такъв начин преди завършването на първото обръщение към fact се прави второ обръщение към тази функция. За целта се генерира нова стекова

рамка на функцията fact, в която за формалния параметър n се откопирва стойност 3. Тялото на функцията fact започва да се изпълнява за втори път (Временно спира изпълнението на тялото на функцията, предизвикано от първото обръщение към нея).

По аналогичен начин възникват още обръщения към функцията fact. При последното от тях, стойността на формалния параметър n е равна на 0 и тялото на fact се изпълнява напълно. Така се получава:



При пето[illegible] към fact стойността на n е равна на 0. В резултат, изпълнението на това обръщение завършва и за стойност на fact се получава 1. След това последователно завършват изпълненията на останалите обръщения към тялото на функцията. При всяко изпълнение на тялото на функцията се определя съответната стойност на функцията fact. След завършването на всяко изпълнение на функцията fact, отделената за fact стекова рамка се освобождава. В крайна сметка в главната програма се връща 24 - стойността на 4!, която се извежда върху екрана.

В този случай, рекурсивното дефиниране на функцията факториел не е подходящо, тъй като съществува лесно итеративно решение.

Препоръка: Ако за решаването на някаква задача може да се използва итеративен алгоритъм, реализирайте го. Не се препоръчва винаги използването на рекурсия, тъй като това води до загуба на памет и време.

Съществуват обаче задачи, които се решават трудно ако не се използува рекурсия.

Задача 86. Да се напише програма, която въвежда от клавиатурата записана без грешка формула от вида

```
<формула> ::=  <цифра> |
               (<формула><знак><формула>)
<знак> ::= + | - | *
<цифра> ::= 0 | 1 | 2 | ...| 9.
```

Програмата да намира и извежда стойността на въведената формула (Например 8 --> 8; ((2-6)*4) --> -16).

Програма Zad86.cpp решава задачата. В нея е дефинирана рекурсивната функция formula, реализираща рекурсивната дефиниция на <формула>.

```
Program Zad86.cpp
#include <iostream.h>
int formula();
int main()
{cout << formula() << "\n";
 return 0;
}
int formula()
{char c, op;
 int x, y;
 cin >> c;                // c е ‘(‘ или цифра
 // <формула> ::= <цифра>
 if (c >= '0' && c <= '9') return (int)c – (int)'0';
 else
 // <формула> ::= (<формула><знак><формула>)
 {x = formula();
  cin >> op;
  y = formula();
  cin >> c;               // прескачане на ‘)’
```

```
  switch (op)
  {case '+': return x + y; break;
   case '-': return x - y; break;
   case '*': return x * y; break;
   default: cout << "error\n"; return 111;
  }
 }
}
```

Забелязваме простотата и компактността на записа на рекурсивните функции. Това проличава особено при работа с динамичните структури: свързан списък, стек, опашка, дърво и граф.

Основен недостатък е намаляването на бързодействието поради загуба на време за копиране на параметрите им в стека. Освен това се изразходва повече памет, особено при дълбока степен на вложеност на рекурсията.

Задачи върху рекурсия

Задача 87. Да се напише рекурсивна програма, която намира най-големия общ делител на две естествени числа.

Програма Zad87.cpp решава задачата.

Program Zad87.cpp

```
#include <iostream.h>
int gcd(int, int);
int main()
{cout << "a, b= ";
 int a, b;
 cin >> a >> b;
 if (!cin || a < 1 || b < 1)
 {cout << "Error! \n";
  return 1;
 }
 cout << "gcd{" << a << ", " << b << "}= " <<  gcd(a, b) <<
"\n";
 return 0;
```

```
}
int gcd(int a, int b)
{if (a == b) return a; else
   if(a > b) return gcd(a-b, b);
   else return gcd(a, b-a);
}
```

Задача 88. Като се използва рекурсивната дефиниция на функцията за степенуване да се напише програма, която по дадени a реално и k – цяло число, намира стойността на a^k.

Програма Zad88.cpp решава задачата.

```
Program Zad88.cpp
#include <iostream.h>
double pow(double, int);
int main()
{cout << "a= ";
 double a;
 cin >> a;
 if (!cin)
 {cout << "Error \n";
  return 1;
 }
 cout << "k= ";
 int k;
 cin >> k;
 if (!cin)
 {cout << "Error \n";
  return 1;
 }
 cout << "pow{" << a << ", " << k << "}= " << pow(a, k) << "\n";
 return 0;
}
double pow(double x, int n)
{if (n == 0) return 1; else
   if (n > 0) return x * pow(x, n-1);
   else return 1.0/pow(x, -n);
}
```

Задача 89. Квадратна мрежа с k реда и k стълба ($1 \le k \le 20$) има два вида квадратчета – бели и черни. В черно квадратче може да се влезе, но не може да се излезе. От бяло квадратче може да се премине във всяко от осемте му съседни, като се прекоси общата им страна или връх. Да се напише програма, която ако са дадени произволна мрежа с бели и черни квадратчета и две произволни квадратчета – начално и крайно, определя дали от началното квадратче може да се премине в крайното.

Анализ на задачата:

а) Ако началното квадратче не е в мрежата, приемаме, че не може да се премине от началното до крайното квадратче.

б) Ако началното квадратче съвпада с крайното, приемаме, че може да се премине от началното до крайното квадратче.

в) Ако началното и крайното квадратчета са различни и началното квадратче е черно, не може да се премине от него до крайното квадратче.

г) Във всички останали случаи, от началното квадратче може да се премине до крайното тогава и само тогава, когато от някое от съседните му квадратчета (в хоризонтално, във вертикално или диагонално направление), може да се премине до крайното квадратче.

Програма Zad89.cpp решава задачата.

Представяне на данните:

Мрежата ще представим чрез квадратна матрица от цели числа mr[k x k] ($1 \le k \le 20$) и ще реализираме чрез съответен двумерен масив. При това mr[i][j] е равно на 0, ако квадратче (i, j) е бяло и е равно на 1, ако е квадратче (i, j) е черно ($0 \le i \le k-1$, $0 \le j \le k-1$). Нека началното квадратче е от ред x и стълб y, а крайното квадратче е от ред m и стълб n.

В програмата Zad89.cpp е дефинирана рекурсивната функция way. Тя връща стойност true, ако от квадратче (x, y) може да се премине до квадратче (m, n) и false – в противен случай. За да се избегне зацикляне (връщане в началното квадратче от всяко съседно), се налага преди рекурсивните обръщения към way, да се промени стойността на mr[x][y] като квадратчето (x, y) се направи черно.

```
Program Zad89.cpp
#include <iostream.h>
```

```
int mr[20][20];
int k, m, n;
bool way(int, int);
void writemr(int, int [][20]);
int main()
{cout << "k from [1, 20] = ";
 do
 {cin >> k;
 }while (k < 1 || k > 20);
 int x, y;
 do
 {cout << "x, y = ";
  cin >> x >> y;
 } while (x < 0 || x > k-1 || y < 0 || y > k-1);
 do
 {cout << "m, n = ";
  cin >> m >> n;
 } while (m < 0 || m > k-1 || n < 0 || n > k-1);
 for (int i = 0; i <= k-1; i++)
 for (int j = 0; j <= k-1; j++)
   cin >> mr[i][j];
 cout << "\n";
 writemr(k, mr);
 if (way(x, y)) cout << "yes \n";
 else cout << "no \n";
 return 0;
}

bool way(int x, int y)
{if (x < 0 || x > k-1 || y < 0 || y > k-1) return false; else
  if (x == m && y == n) return true; else
    if (mr[x][y] == 1) return false; else
      {mr[x][y] = 1;
       bool b = way(x+1,  y) || way(x-1, y)  ||
                way(x,  y+1) || way(x, y-1)  ||
                way(x-1,y-1) || way(x-1, y+1)||
                way(x+1,y-1) || way(x+1, y+1);
       mr[x][y] = 0;
       return b;
```

```
      }
    }
   void writemr(int k, int mr[][20])
   {for (int i = 0; i <= k-1; i++)
   {for (int j = 0; j <= k-1; j++)
        cout << mr[i][j];
    cout << '\n';
   }
   }
```

Задача 90. Да се напише рекурсивен вариант на функцията от по-висок ред accumulate.

Функцията accumulate, дефинирана по-долу, решава задачата. Тя използва дефинициите на типовете:

```
typedef double (*type1) (double, double);
typedef double (*type2) (double);

double accumulate(type1 op, double null_val, double a, double b,
                  type2 f, type2 next)
{if (a > b + 1e-14) return null_val;
 else return op(f(a),
                accumulate(op, null_val, next(a), b, f, next));
}
```

Задача 91. Да се напише рекурсивна функция double min(int n, double* x), която намира минималния елемент на редицата x_0, x_1, ..., x_{n-1}.

първо решение:

```
double min(int n, double* x)
{double b;
 if (n == 1) return x[0]; else
 {b =  min(n-1, x);
  if (b < x[n-1]) return b; else return x[n-1];
 }
}
```

второ решение:

```
double min(int n, double* x)
{double b;
 if (n == 1) return x[0]; else
 {b =  min(n-1, x+1);
  if (b < x[0]) return b; else return x[0];
 }
}
```

Задача 92. Да се напише рекурсивна функция, която проверява дали елементът x принадлежи на редицата a_0, a_1, ..., a_{n-1}.

първо решение:

```
bool member(int x, int n, int* a)
{if (n == 1 ) return a[0] == x;
 return x == a[0] || member(x, n-1, a+1);
}
```

второ решение:

```
bool member(int x, int n, int* a)
{if (n == 1 ) return a[0] == x;
 return x == a[n-1] || member(x, n-1, a);
}
```

Задача 93. Да се напише рекурсивна функция, която проверява дали редицата x_0, x_1, ..., x_{n-1} е монотонно растяща.

първо решение:

```
bool monincr(int n, double* x)
{if (n == 1) return true; else
 return x[n-2] <= x[n-1] && monincr(n-1, x);
}
```

второ решение:

```
bool monincr(int n, double* x)
{if (n == 1) return true; else
 return x[0] <= x[1] && monincr(n-1, x+1);
}
```

Задача 94. Да се напише рекурсивна функция, която проверява дали редицата a_0, a_1, ..., a_{n-1} се състои от различни елементи.

първо решение:

```
bool differ(int n, int* a)
{if (n == 1) return true; else
 return !member(a[0], n-1, a+1) && differ(n-1, a+1);
}
```

второ решение:

```
bool differ(int n, int* a)
{if (n == 1) return true; else
 return !member(a[n-1], n-1, a) && differ(n-1, a);
}
```

Задача 95. Дадена е квадратна мрежа от клетки, всяка от които е празна или запълнена. Запълнените клетки, които са свързани, т.е. имат съседни в хоризонтално, вертикално или диагонално направление, образуват област. Да се напише програма, която намира броя на областите и размера (в брой клетки) на всяка област.

Програма Zad95.cpp решава задачата.

Представяне на данните:

Мрежата ще представим чрез квадратна матрица от цели числа mr[n x n] ($1 \le n \le 20$) и ще реализираме чрез съответен двумерен масив. При това mr[i][i] е 1 ако квадратче (i, j) е запълнено и 0 – в противен случай ($0 \le i \le n-1$, $0 \le j \le n-1$).

Анализ на задачата:

Ще дефинираме функция broy, която преброява клетките в областта, съдържаща дадена клетка (x, y). Функцията има два параметъра x и y - координатите на точката и реализира следния алгоритъм:

а) Ако клетката с координати (x, y) е извън мрежата, приемаме, че броят на клетките в областта е равен на 0.

б) В противен случай, ако клетката с координати (x, y) е празна, приемаме, че броят е равен на 0.

в) В останалите случаи, броят на клетките в областта е равен на сумата от 1 и броя на клетките на всяка област, на която принадлежат осемте съседни клетки на клетката (x, y).

От подточка в) следва, че функцията broy е рекурсивна. За да избегнем зацикляне и многократно преброяване, трябва преди рекурсивното обръщение на направим клетката (x, y) празна.

```
Program Zad95.cpp;
#include <iostream.h>
int mr[20][20];
int n;
int broy(int x, int y)
{if (x < 0 || x > n-1 || y < 0 || y > n-1) return 0;
 else if (mr[x][y] == 0) return 0;
 else {mr[x][y] = 0;
       return 1 + broy(x-1, y+1) + broy(x, y+1)
                + broy(x+1, y+1) + broy(x+1,y)
                + broy(x+1, y-1) + broy(x, y-1)
                + broy(x-1, y-1) + broy(x-1, y);
    }
}
int main()
{cout << "mreja: \n";
 do
 {cout << "n= ";
  cin >> n;
 }while (n < 1 && n > 20);
 for (int i = 0; i <= n-1; i++)
   for (int j = 0; j <= n-1; j++)
    cin >> mr[i][j];
int br = 0;
for (i = 0; i <= n-1; i++)
 for (int j = 0; j <= n-1; j++)
  if (mr[i][j] == 1)
  {br++;
   cout << "size of the " <<  br << " th location is equal to "
        << broy(i, j) << endl;
  }
return 0;
}
```

Задача 96. Дадени са n града (n е естествено число, $1 \le n \le 10$) и целочислена матрица Anxn, така че a_{ij} е равно на 1, ако има пряк път от град i до град j и е 0 в противен случай ($0 \le i, j \le n-1$). Да се напише програма, която установява дали съществува път между два произволно зададени града (Приемаме, че ако от град i до град j има път, то има път и от град j до град i).

Анализ на задачата:

Ще дефинираме рекурсивната булева функция way, която зависи от два параметъра i и j, показващи номерата на градовете, между които се търси дали съществува път. Функцията реализира следния алгоритъм:

а) Ако i = j, съществува път от град i до град j.

б) Ако i ≠ j и има пряк път от град i до град j, има път между двата града.

в) В останалите случаи има път от град i до град j, тогава и само тогава, когато съществува град k, с който град i е свързан с пряк път и от който до град j има път.

Програма Zad96.cpp решава задачата.

```
Program Zad96.cpp;
#include <iostream.h>
int arr[10][10];
int n;
bool way(int i, int j)
{if (i == j) return true; else
   if (arr[i][j] == 1) return true; else
   {bool b = false;
    int k = -1;
    do
    {k++;
     if (arr[i][k] == 1)
     {arr[i][k] = 0; arr[k][i] = 0;
      b = way(k, j);
      arr[i][k] = 1; arr[k][i] = 1;
      }
     }while (!b && k <= n-2);
     return b;
    }
```

```
}
int main()
{do
 {cout << "n= ";
  cin >> n;
 }while (n < 1 || n > 10);
 for (int i = 0; i <= n-1; i++)
   for (int j = 0; j <= n-1; j++)
      arr[i][j] = 0;
 for (i = 0; i <= n-2; i++)
   for (int j = i+1; j <= n-1; j++)
   {cout << "connection between " <<  i
         << " and " << j << " 0/1? ";
    cin >> arr[i][j];
    arr[j][i] = arr[i][j];
   }
 int j;
 do
 {cout << "start and final towns: ";
  cin >> i >> j;
 }while (i < 0 || i > 9 || j < 0 ||  j > 9);
 if (way (i, j)) cout << "yes \n";
 else cout << "no\n";
 return 0;
}
```

Задача 97. Дадени са n града (n е естествено число, $1 \leq n \leq 10$) и целочислена матрица Anxn, така че a_{ij} е равно на 1, ако има пряк път от град i до град j и е 0 в противен случай ($0 \leq i, j \leq n-1$). Да се напише програма, която намира пътя между два произволно зададени града в случай, че път между тях съществува.

Анализ на задачата:

Процедурата foundway намира пътя от град i до град j в случай, че съществува, т.е. ако way(i, j) има стойност true. Пътят се записва в едномерния масив int x[100]. Дължината на пътя се записва в променливата s.

Програма Zad97.cpp решава задачата. В нея е пропусната дефиницията на функцията way. При обръщение към функцията foundway

параметърът-псевдоним s трябва да се свърже с параметър, инициализиран с –1.

```
Program Zad97.cpp
#include <iostream.h>
int arr[10][10];
int n;

bool way(int i, int j)
...
void foundway(int i, int j, int& s, int x[])
{s++;
 x[s] = i;
 if (i != j)
  if (arr[i][j] == 1)
  {s++;
   x[s] = j;
  }
  else
  {bool b = false;
   int k = -1;
   do
   {k++;
    if (arr[i][k] == 1) b = way(k, j);
   }while (!b);
   arr[i][k] = 0; arr[k][i] = 0;
   foundway(k, j, s, x);
   arr[i][k] = 1; arr[k][i] = 1;
  }
}

int main()
{int p = -1;
 int a[100];
 do
 {cout << "n= ";
  cin >> n;
 }while (n < 1 || n > 10);
 for (int i = 0; i <= n-1; i++)
```

```
      for (int j = 0; j <= n-1; j++)
         arr[i][j] = 0;
    for (i = 0; i <= n-2; i++)
      for (int j = i+1; j <= n-1; j++)
      {cout << "connection between " <<  i << " and "
          << j << " 0/1? ";
      cin >> arr[i][j];
      arr[j][i] = arr[i][j];
      }
    int j;
    do
    {cout << "start and final towns: ";
     cin >> i >> j;
    }while (i < 0 || i > 9 || j < 0 ||  j > 9);
    if(way (i, j))
    {foundway(i, j, p, a);
     p++;
     for (int l = 0; l <= p-1; l++) cout << a[l] << "  " ;
     cout << endl;
    }
    else cout << "no\n";
    return 0;
   }
```

Задача 98. Дадени са n града (n е естествено число, $1 \leq n \leq 10$) и целочислена матрица Anxn, така че a_{ij} е равно на 1, ако има пряк път от град i до град j и е 0 в противен случай ($0 \leq i, j \leq n-1$). Да се напише булева функция fixway(int i, int j, int p), която установява дали съществува път с дължина p от град i до град j ($p \geq 1$).

Анализ на задачата:

Булевата функция fixway реализира следния алгоритъм:

а) Ако p = 1, има път от град i до град j с дължина p ако има пряк път между двата града.

б) Ако p > 1, има път от град i до град j с дължина p тогава и само тогава, когато съществува град k, с който град i е свързан с пряк път и от който до град j има път с дължина p-1.

Следващият програмен фрагмент дефинира само функцията fixway.

```
bool fixway(int i, int j, int p)
{if (p == 1) return arr[i][j] == 1; else
 {bool b = false;
  int k = -1;
  do
  {k++;
   if (arr[i][k] == 1) b = fixway(k, j, p-1);
  }while (!b && k <= n-2);
  return b;
 }
}
```

Внимателното анализиране на функцията показва, че тя извършва проверка за съществуване на цикличен път от един до друг връх с указана дължина.

Пример: Ако имаме само два града, означени с 0 и 1 и те са свързни с пряк път, fixway(0, 1, 3) ще отговори с true.

Ако търсим съществуването само на ациклични пътища, ще използваме модификацията на горната функция, дадена по-долу.

```
bool fixway(int i, int j, int p)
{bool b;
 int k;
 if (p == 1) return arr[i][j] == 1; else
 {b = false;
  k = -1;
  do
  {k++;
   if (arr[i][k] == 1)
   {arr[i][k] = 0; arr[k][i] = 0;
    b = fixway(k, j, p-1);
    arr[i][k] = 1; arr[k][i] = 1;
   }
  }while (!b && k <= n-2);
  return b;
 }
}
```

Задача 99. Дадени са n града (n е естествено число, $1 \le n \le 10$) и целочислена матрица Anxn, така че a_{ij} е равно на 1, ако има пряк път от град i до град j и е 0 в противен случай ($0 \le i, j \le n-1$). Да се напише процедура foundfixway(int i, int j, int p, int s&, int* x), която намира пътя от град i до град j с дължина p в случай, че път с дължина p от град i до град j съществува. Пътят да се запомни в масива x, а параметърът s да съдържа текущата дължина на пътя.

Програма Zad99.cpp решава задачата. За краткост, дефиницията на функцията fixway е пропусната.

```
Program Zad99.cpp
#include <iostream.h>
int arr[10][10];
int n;

bool fixway(int i, int j, int p)
...
void foundfixway(int i, int j, int p, int& s, int* x)
{s++;
 x[s] = i;
 if(p == 1)
 {s++;
  x[s] = j;
 }
 else
 {bool b = false;
  int k = -1;
  do
  {k++;
   if (arr[i][k] == 1) b = fixway(k, j, p-1);
  }while (!b);
  foundfixway(k, j, p-1, s, x);
 }
}

int main()
{int p = -1;
```

```
 int a[100];
 do
 {cout << "n= ";
  cin >> n;
 }while (n < 1 || n > 10);
 for (int i = 0; i <= n-1; i++)
   for (int j = 0; j <= n-1; j++)
      arr[i][j] = 0;
 for (i = 0; i <= n-2; i++)
   for (int j = i+1; j <= n-1; j++)
   {cout << "connection between " <<  i << " and "
         << j << " 0/1? ";
    cin >> arr[i][j];
    arr[j][i] = arr[i][j];
   }
 int j, l;
 do
 {cout << "start and final towns, and len between them: ";
  cin >> i >> j >> l;
 }while (i < 0 || i > 9 || j < 0 ||  j > 9);
 if (fixway (i, j, l))
 {foundfixway(i, j, l, p, a);
  for (int m = 0; m <= l; m++) cout << a[m] << "  " ;
  cout << endl;
 }
 else cout << "no\n";
 return 0;
}
```

Задачи

Задача 1. Да се напише рекурсивна функция, която намира стойността на функцията на Акерман Ack(m, n), дефинирана за $m \geq 0$ и $n \geq 0$ по следния начин:

Ack(0, n) = n+1

Ack(m, 0) = Ack(m-1, 1), m > 0

$Ack(m, n) = Ack(m-1, Ack(m, n-1)), m > 0, n > 0$.

Задача 2. Да се напише рекурсивна функция, която установява, дали в записа на естественото числа n се съдържа цифрата k.

Задача 3. Да се напише рекурсивна функция, която установява, дали редицата от цели числа $a_0, a_1, ..., a_{n-1}$ е монотонно намаляваща.

Задача 4. Да се напише рекурсивна функция, която установява, дали редицата от цели числа $a_0, a_1, ..., a_{n-1}$ се състои от различни елементи.

Задача 5. Да се напише рекурсивна програма, която намира стойността на полинома

$$P_n(x) = a_0x^{n-1} + a_1x^{n-2} + ... + a_{n-2}x + a_{n-1}$$

където x и a_i $(0 \le i \le n)$ са дадени реални числа.

Задача 6. Да се напише рекурсивна функция, която пресмята корен квадратен от x, $x \ge 0$, по метода на Нютон.

Задача 7. Да се напише рекурсивна функция, която установява, дали цялото число p се съдържа в редицата от цели числа a_0, a_1, ..., a_{n-1} (n е естествено число, $n \ge 1$).

Задача 8. Да се напише рекурсивна функция, която добавя елемент в сортиран масив, като запазва наредбата на елементите.

Задача 9. Да се напише рекурсивна функция, която изключва елемент от сортиран масив, като запазва наредбата на елементите.

Задача 10. Дадена е мрежа от m x n квадратчета, като за всяко квадратче е определен цвят - бял или черен. Път ще наричаме редица от съседни във вертикално или хоризонтално направление квадратчета с един и същ цвят. Област ще наричаме множество от квадратчета с един и същ цвят между всеки две, от които има път. Дадено е квадратче. Да се определи:

а) броят на квадратчетата от областта, в която се съдържа даденото квадратче.

б) броят на областите с цвят, съвпадащ с цвета на даденото квадратче.

в) броят на областите с цвят, различен от цвета на даденото квадратче.

г) броят на квадратчетата с цвят, съвпадащ с цвета на даденото квадратче, които не са в една област с него.

Задача 11. Дадено е множество от n града и за всеки два от тях е определено дали са свързани с път или не. Едно множество от градове

ще наричаме пълно, ако всеки два различни града, принадлежащи на множеството, са свързани с път. Да се напише програма, която по дадено k, k < n, извежда всички пълни множества, състоящи се от k на брой града.

Задача 12. Дадено е множество от n града и за всеки два от тях е определено дали са свързани с път или не. Едно множество от градове ще наричаме независимо, ако всеки два различни града, принадлежащи на множеството, не са свързани с път. Да се напише програма, която по дадено k, k < n, извежда всички независими множества, състоящи се от k на брой града.

Задача 13. Да се напише програма, която намира стойността на произволен израз от вида

```
израз ::= цяло_число |
      (израз * израз).
```

Задача 14. Да се напише програма, която намира стойността на произволен израз от вида

```
израз ::= цяло_число |
          (израз ^ израз),
```

където с ^ е означена операцията степенуване.

Задача 15. Да се напише програма, която въвежда от клавиатурата без грешка булев израз от вида

```
<булев_израз> ::= true | false |
                  <операция>(<операнди>)
<операция> ::= not | and | or
<операнди> ::= <операнд> |
               <операнд>, <операнди>
<операнд> ::= <булев_израз>,
```

където not има само един операнд, а and и or могат да имат произволен брой операнди. Програмата намира и извежда стойността на булевия израз.

Допълнителна литература

2. Ст. Липман, Езикът C++ в примери, “КОЛХИДА ТРЕЙД” КООП, С. 1993.
3. М. Тодорова, Програмиране на Паскал, Полипринт Враца, С., 1993.

4. М. Тодорова, Езици за функционално и логическо програмиране – функционално програмиране, СОФТЕХ, С., 1998.

Глава 11

Структури

2. Структура от данни запис

Логическо описание

Записът е съставна статична структура от данни, която се определя като крайна редица от фиксиран брой елементи, които могат да са от различни типове. Достъпът до всеки елемент от редицата е пряк и се осъществява чрез име, наречено **поле на записа.**

Физическо представяне

Полетата на записа се представят последователно в паметта.

Примери:

1. Данните за студент от една група (име, адрес, факултетен номер, оценки по изучаваните предмети) могат да се зададат чрез запис с четири полета.

2. Данните за книга от библиотека (заглавие, автор, година на издаване, издателство, цена) могат да се зададат чрез запис с пет полета.

3. Комплексно число може да се зададе чрез запис с две реални полета.

В езика C++ записите се реализират чрез структури. Ще разгледаме последните в развитие. Отначало ще опишем възможностите им на ниво -език C.

3. Дефиниране и използване на структури

Една структура се определя чрез имената и типовете на съставящите я полета. Фигура 1 дава непълна дефиниция на структура.

```
<дефиниция_на_структура> ::= struct <име_на_структурата>
            {<дефиниция_на_полета>;
             {<дефиниция_на_полета>;}opc
            };
<име_на_структура> ::= <идентификатор>
<дефиниция_на_полета> ::= <тип> <име_на_поле>{, <име_на_поле>}opc
<име_на_поле> ::= <идентификатор>
<тип> ::= <име_на_тип> | <дефиниция_на_тип>
```

Фиг. 1

Структурите, дефинирани по този начин, могат да се използват като типове данни. Имената на полетата в рамките на една дефиниция на структура трябва да са *различни* идентификатори.

Примери:

1. struct complex
   {double re, im;};

задава структура с име complex с две полета с имена re и im от тип double. Чрез нея се задават комплексните числа.

2. struct book
   {char name[41], author[31];
    int year;
    double price;

```
};
```

задава структура с име book с четири полета: *първо поле* с име name от тип символен низ с максимална дължина 40 и определящо името на книгата; *второ поле* с име author от тип символен низ с максимална дължина 30, определящо името на автора на книгата; *трето поле* с име year от тип int, определящо годината на издаване и *четвърто поле* а име price от тип double и определящо цената на книгата. Чрез тази структура се задава информация за книга.

3.
```
struct student
{int facnum;
 char name[36];
 double marks[30];
};
```

задава структура с име student и с три полета: *първо поле* с име facnum от тип int, означаващо факултетния номер на студента; *второ поле* с име name от тип символен низ с максимална дължина 35, определящо името на студента и *трето поле* с име marks от тип реален масив с 30 компоненти и означаващо оценките от положените изпити.

Възможно е за име на структура, на нейно поле и на произволна променлива на програмата да се използва един и същ идентификатор. Тъй като това намалява читаемостта на програмата, засега не препоръчваме използването му.

Допуска се влагане на структури, т.е. поле на структура да е структура.

Пример: Допустими са дефинициите:

```
struct xx
{int a, b, c;
}
struct pom
{int a;
 double b;
 char c;
 xx d;
};
```

Не е възможно обаче поле на структура да е от тип, съвпадащ с името на структурата.

Пример: Не е допустима дефиниция от вида

```
struct xxx{
   xxx member;      // опит за рекурсивна дефиниция
```

```
};
```

тъй като компилаторът не може да определи размера на xxx. Допустима е обаче дефиницията:

```
struct xxx{
    xxx* member;
};
```

Допълнение: За да е възможно две дефиниции на структури да се обръщат една към друга, е необходимо пред дефинициите им да се постави декларацията на втората структура. Например, ако искаме дефиницията на структурата list да използва дефиницията на структурата link и обратно, ще трябва да ги подредим по следния начин

```
struct list;        // декларация на втората структура
struct link{
    link* pred;
    link* succ;
    list* member;
};
struct list{
    link* head;
};
```

Дефиницията на структура не предизвиква отделянето на памет за съхраняване на компонентите й. Може да се постави извън функция, в началото на функция или в началото на блок. Местоположението на дефиницията определя областта на името на структурата – съответно за всички функции след дефиницията, в рамките на функцията и в рамките на блока. Най-често дефиницията се задава пред първата функция на програмата и така става достъпна за всички функции.

Тъй като дефинирането на структура чрез задаване на името на структурата определя нов тип данни, ще определим множеството от стойности и операциите и вградените функции, свързани с него.

Множество от стойности

Множеството от стойностите на една структура се състои от всички крайни редици от по толкова елемента, колкото са полетата й, като всеки елемент е от тип, съвместим с типа на съответното поле.

Примери:

1. Множеството от стойности на структурата complex се състои от всички двойки реални числа.

2. Множеството от стойности на структурата book се състои от всички четворки от вида:

   {char[41], char[31], int, double}.

3. Множеството от стойности на структурата student се състои от всички тройки от вида:

   {int, char[36], double[30]}.

Променлива величина, множеството от допустимите стойности, на която съвпада с множеството от стойности на дадена структура, е променлива от дадения тип структура. Променлива от тип структура се дефинира в областта на структурата по следния начин (Фиг. 2):

```
<дефиниция_на_променлива_от_тип_структура> ::=
[struct]opc <име_на_структура>
   <променлива> [={<редица_от_изрази>}]opc
   {,<променлива> [={<редица_от_изрази>}]opc}opc ;|
struct {<дефиниция_на_полета>;
        {<дефиниция_на_полета>;}opc
       }<променлива> [={<редица_от_изрази>}]opc
        {,<променлива> [={<редица_от_изрази>}]opc}opc ;
<променлива> ::= <идентификатор>
```

Фиг. 2.

В C++ използването на запазената дума struct не е задължително. Някои програмисти го използват заради яснотата на кода. Конструкцията {<редица_от_изрази>} предизвиква инициализация на дефинирана променлива. Изразите, изредени във фигурните скоби, се разделят със запетая. Всеки израз инициализира поле на структурата и трябва да е от тип, съвместим с типа на съответното поле. Ако са записани по-малко изрази от броя на полетата, компилаторът допълва останалите с нулевите стойности за съответния тип на поле.

Примери:

```
complex z1, z2 = {5.6, -8.3}, z3;
book b1, b2, b3;
struct student s1 = {44505, “Ivan Ivanov”, {5.5, 6, 5, 6}};
struct{
   int x;
   double y;
}p, q = {-2, -1.6};
pom y;
```

Последната дефиниция от примера по-горе дефинира променливите p и q като променливи от тип структурата

```
struct{
   int x;
   double y;
}
```

която участва с дефиницията, а не с името си.

Достъпът до полетата на структура е пряк. Един начин за неговото осъществяване е чрез променлива от тип структурата, като променливата и името на полето на структурата се разделят с оператора точка (Фиг. 3). Получените конструкции са *променливи* от типа на полето и се наричат **полета на променливата** от тип структура или **член-данни на структурата**, свързани с променливата.

Операторът . е ляво-асоциативен и има приоритет еднакъв с този на () и [].

```
<поле_на_променлива_структура> ::=
     <променлива_структура>.<име_на_поле>
```

Фиг. 3.

Примери:

С променливите z1, z2, z3 се свързват променливите от тип double:

```
z1.re, z1.im, z2.re, z2.im, z3.re и z3.im
```

а с b1, s1 и y –

```
b1.name   - от тип char [41],    b1.author – от тип char [31],
b1.year   - от тип int,          b1.price  - от тип double,
s1.facnum – от тип int,          s1.name   - от тип char [36],
```

```
s1.marks  - от тип double [30],  y.a       - от тип int,
y.b       - от тип double,       y.c      - от тип char,
y.d      -  от тип xx и
            с полета y.d.a, y.d.b и y.d.c от тип int.
```

Дефинирането на променлива от тип структура свързва променливата с множеството от стойности на съответната структура. След дефинициите от примера по-горе, променливите z1, z2 и z3 се свързват с множеството от стойностите на типа complex. При това свързване z2 е свързано с комплексното число 5.6 – i8.3 чрез инициализация. Свързването на z1 и z3 със съответни комплексни числа може да стане чрез инициализация, подобно на z2, чрез присвояване, например z1 = z2;, или чрез задаване на стойности на полетата на променливата, например

```
z3.re = 3.4;
z3.im = -30.5;
```

свързва z3 с комплексното число 3.4 –i30.5.

Освен това, дефинирането на променлива от тип структура предизвиква отделяне на определено количество памет за всяко поле на променливата. Последното се определя от типа на полето. Полетата се разполагат последователно в паметта. Обикновено всяко поле се разполага от началото на машинна дума. Полетата, които не изискват цяло число на брой машинни думи, не използват напълно отредената им памет. Този начин за подреждане на полетата на структура се нарича **изравняване на границата на машинна дума.** За реализацията Visual C++ 6.0 размерът на 1 машинна дума е 8B.

Пример: За променливите p и q, дефинирани по-горе, ще се отделят по 16, а не по 12 байта

ОП

p.x	p.y	q.x	q.y
-	-	-2	-1.6
8B	8B	8B	8B

Допълнение: Две структури, дефинирани по един и същ начин, са различни. Например, дефинициите

```
struct str1{
   int a;
   int b;
};
```

и

```
struct str2{
   int a;
   int b;
};
```

определят два различни типа. Дефинициите

```
str1 x;
str2 y = x;
```

предизвикват грешка заради смесване на типовете (x и y са от различни типове).

Операции и вградени функции

Операциите над структури зависят от реализацията на езика. По стандарт за всяка реализация са определени следните операции и вградени функции:

а) над полетата на променливи от тип структура

Всяко поле на променлива от тип структура е от някакъв тип. Всички операции и вградени функции, допустими над данните от този тип, са допустими и за полето на променливата.

б) над променливи от тип структура

- Възможно е на променлива от тип структура да се присвои стойността на вече инициализирана променлива от същия тип структура или стойността на израз от същия тип.

Пример:

```
z3 = z2;
p = q;
```

- Възможно е формален параметър на функция, а също резултатът от изпълнението й, да са структури. Структури с големи размери обикновено се предават чрез указатели или псевдоними на структури. Така се спестяват ресурси. Освен това, тези начини за предаване са по-сигурни.

Ще илюстрираме използването на структурите чрез следния пример.

Задача 100. Да се напише програма, която:

а) въвежда факултетните номера, имената и оценките по 5 предмета на студентите от една група;

б) извежда в табличен вид въведените данни;

в) сортира в низходящ ред по среден успех данните;

г) извежда сортираните данни, като за всеки студент извежда и средния му успех.

Програма Zad100.cpp решава задачата. Тя реализира следното представяне на данните:

Данните за студент се дефинират чрез структурата

```
struct student
{int facnom;
 char name[26];
 double marks[10];
};
```

а данните за групата чрез масива

```
student table[30];
```

Броят на предметите е дефиниран като константа с име NUM.

Реализирани са следните процедури и функции:

void read_student(student&);

Въвежда стойности на полетата на структура от тип student.

void print_student(student const &);

Извежда върху екрана полетата на структура от тип student.

void sorttable(int n, student[]);

Сортира компонентите на масив от структури от тип student в низходящ ред по среден успех. Резултатът от сортирането е в същия масив от структури.

double average(double);*

Намира средно-аритметичното на елементите на масив от NUM реални числа.

Program Zad100.cpp

```
#include <iostream.h>
#include <iomanip.h>
#include <string.h>
const NUM = 5;
struct student
{int facnom;
 char name[26];
```

```
 double marks[NUM];
};
void read_student(student&);
void print_student(const student&);
void sorttable(int n, student[]);
double average(double*);

int main()
{cout << setprecision(2) << setiosflags(ios::fixed);
 student table[30];
 int n;
 do
 {cout << "number of students? ";
  cin >> n;
 }while (n < 1 || n > 30);
 int i;
 for (i = 0; i <= n-1; i++)
   read_student(table[i]);
 cout << "Table: \n";
 for (i = 0; i <= n-1; i++)
 {print_student(table[i]);
  cout << endl;
 }
 sorttable(n, table);
 cout << "\n New Table: \n";
 for (i = 0; i <= n-1; i++)
 {print_student(table[i]);
  cout << setw(7) << average(table[i].marks) << endl;
 }
 return 0;
}
void read_student(student& s)
{cout << "fak. nomer: ";
 cin >> s.facnom;
 char p[100];
 cin.getline(p, 100);
 cout << "name: ";
 cin.getline(s.name, 40);
 for (int i = 0; i <= NUM-1; i++)
```

```
 {cout << i << " -th mark: ";
  cin >> s.marks[i];
 }
}

void print_student(const student& stud)
{cout << setw(6) << stud.facnom << setw(30) << stud.name;
 for (int i = 0; i <= NUM-1; i++)
 cout << setw(6) << stud.marks[i];
}

void sorttable(int n, student a[])
{for (int i = 0; i <= n-2; i++)
 {int k = i;
  double max = average(a[i].marks);
  for (int j = i+1; j <= n-1; j++)
    if (average(a[j].marks) > max)
    {max = average(a[j].marks);
     k = j;
    }
  student x = a[i]; a[i] = a[k]; a[k] = x;
 }
}

double average(double* a)
{double s = 0;
 for (int j = 0; j <= NUM-1; j++)
    s += a[j];
 return s/NUM;
}
```

Процедурата за сортиране void sorttable(int n, student a[]) е реализирана неефективно тъй като се разместват структури. При структури с големи размери сортирането е много бавно. Реализацията може да се подобри като се създаде масив от **указатели към структурите** – елементи на table. При необходимост от размяна, тя се осъществява не със структурите, а с адресите на съответните им указатели.

3. Указатели към структури

Дефинират се по общоприетия начин (фиг. 4).

<указател_към_структура> ::=
        struct <име_на_структура> * <променлива_указател>
                                [= & <променлива>]$_{opc}$;
<променлива> е от тип <име_на_структура>.

Фиг. 4.

В C++ запазената дума struct може да се пропусне.

Пример:

```
student st1, st2;
...
student *pst = &st1;
...
pst = &st2;
...
```

В резултат за променливата-указател pst се отделят 4B ОП, в които отначало се записва адресът на st1, след което – адресът на st2.

Достъпът до полетата на променлива от тип структура чрез указател към нея се осъществява чрез обръщението:

(*<променлива_указател>).<име_на_поле>

което е еквивалентно на

<променлива_указател> -> <име_на_поле>

За разделител са използвани знаците – и >, записани последователно.

Пример: Достъпът до полетата на stud2 чрез указателя pst се реализира чрез обръщенията:

```
pst -> facnom
pst -> name
pst -> marks
```

Задача 101. Да се модифицира функцията за сортиране sorttable от задача 100, като за сортирането се използва помощен масив от указатели към структурата student.

Пред вид промени и в main ще дадем програмен фрагмент, решаващ задачата. Функциите read_student(), print_student() и average() са пропуснати, тъй като са същите като в задача 100.

```
Program Zad101.cpp
#include <iostream.h>
#include <iomanip.h>
#include <string.h>
const NUM = 5;
struct student
{int facnom;
 char name[26];
 double marks[NUM];
};
void read_student(student&);
void print_student(const student&);
void sorttable(int n, student* []);
double average(double*);

int main()
{cout << setprecision(2) << setiosflags(ios::fixed);
 student table[30];
 student* ptable[30];
 int n;
 do
 {cout << "number of students? ";
  cin >> n;
 }while (n < 1 || n > 30);
 int i;
 for (i = 0; i <= n-1; i++)
 {read_student(table[i]);
  ptable[i] = &table[i];
 }
 cout << "Table: \n";
```

```
  for (i = 0; i <= n-1; i++)
  {print_student(table[i]);
   cout << endl;
  }
  sorttable(n, ptable);
  cout << "\n New Table: \n";
  for (i = 0; i <= n-1; i++)
  {print_student(*ptable[i]);
   cout << setw(7) << average(ptable[i]->marks) << endl;
  }
  return 0;
 }
 ...
 void sorttable(int n, student* a[])
 {for (int i = 0; i <= n-2; i++)
  {int k = i;
   double max = average(a[i]->marks);
   for (int j = i+1; j <= n-1; j++)
     if (average(a[j]->marks) > max)
     {max = average(a[j]->marks);
      k = j;
     }
   student* x;
   x = a[i]; a[i] = a[k]; a[k] = x;
  }
 }
 ...
```

За работа със структури от данни се използва подходът абстракция със структури от данни.

4. Абстракция със структури от данни

При този подход методите за използване на данните са разделени от методите за тяхното представяне. Програмите се конструират така, че да работят с "абстрактни данни" – данни с неуточнено представяне. След това представянето се конкретизира с помощта на множество

функции, наречени **конструктори, селектори** и **предикати**, които реализират “абстрактните данни” по конкретен начин.

Ще го илюстрираме чрез следната задача.

Задача 102. Да се напише програма, която реализира основните рационално-числови операции – събиране, изваждане, умножение и деление на рационални числа.

Програма Zad102.cpp решава задачата. Тя дефинира функции за събиране, изваждане, умножение и деление на рационални числа като реализира следните общоизвестни правила:

$$\frac{n1}{d1} + \frac{n2}{d2} = \frac{n1.d2 + n2.d1}{d1.d2}$$

$$\frac{n1}{d1} - \frac{n2}{d2} = \frac{n1.d2 - n2.d1}{d1.d2}$$

$$\frac{n1}{d1} * \frac{n2}{d2} = \frac{n1.n2}{d1.d2}$$

$$\frac{n1}{d1} / \frac{n2}{d2} = \frac{n1.d2}{d1.n2}.$$

Тези операции лесно могат да се реализират ако има начин за конструиране на рационално число по зададени две цели числа, представящи съответно неговите числител и знаменател и ако има начини, които по дадено рационално число извличат неговите числител и знаменател. Затова в програмата Zad102.cpp са дефинирани функциите:

`void makerat(rat& r, int a, int b)` – която конструира рационално число r по дадени числител a и знаменател b;

`int numer(rat& r)` - която намира числителя на рационалното число r;

`int denom(rat& r)` - която намира знаменателя на рационалното число r,

където с rat означаваме типа рационално число.

Все още не знаем как точно да реализираме тези функции, нито как се представя рационално число, но ако ги имаме, функциите за рационално-числова аритметика и процедурата за извеждане на рационално число могат да се реализират по следния начин:

```
rat sumrat(rat& r1, rat& r2)        //събира рационални числа
{rat r;
 makerat(r, numer(r1)*denom(r2)+numer(r2)*denom(r1),
            denom(r1)*denom(r2));
 return r;
}
rat subrat(rat& r1, rat& r2)        // изважда рационални числа
{rat r;
 makerat(r, numer(r1)*denom(r2)-numer(r2)*denom(r1),
            denom(r1)*denom(r2));
 return r;
}
 rat multrat(rat& r1, rat& r2)      // умножава рационални числа
{rat r;
 makerat(r, numer(r1)*numer(r2),
            denom(r1)*denom(r2));
 return r;
}
rat quotrat(rat& r1, rat& r2)       // дели рационални числа
{rat r;
 makerat(r, numer(r1)*denom(r2),
            denom(r1)*numer(r2));
 return r;
}
void printrat(rat& r)            // извежда рационално число
{cout << numer(r) << "/" << denom(r) << '\n';
}
```

Сега да се върнем към представянето на рационалните числа, а също към реализацията на примитивните операции: конструктора makerat и селекторите numer и denom. Тъй като рационалните числа са частни на две цели числа, удобно представяне на рационално число е структура от вида:

```
struct rat{
   int num, den;
```

```
   };
```
Тогава примитивните функции, реализиращи конструктора makerat и двата селектора numer и denom, имат вида:
```
void makerat(rat& r, int a, int b)
{r.num = a;
 r.den = b;
}
int numer(rat& r)
{return r.num;
}
int denom(rat& r)
{return r.den;
}
```
Тези функции са включени в Zad102.cpp и са използвани за намиране на сумата, разликата, произведението и делението на рационалните числа ½ и ¾.

```
Program Zad102.cpp
#include <iostream.h>
struct rat{
int num, den;
   };
void makerat(rat& r, int a, int b)
{r.num = a;
 r.den = b;
}
int numer(rat& r)
{return r.num;
}
int denom(rat& r)
{return r.den;
}
rat sumrat(rat& r1, rat& r2)
{rat r;
 makerat(r, numer(r1)*denom(r2)+numer(r2)*denom(r1),
            denom(r1)*denom(r2));
 return r;
}
rat subrat(rat& r1, rat& r2)
```

```
{rat r;
 makerat(r, numer(r1)*denom(r2)-numer(r2)*denom(r1),
            denom(r1)*denom(r2));
 return r;
}
rat multrat(rat& r1, rat& r2)
{rat r;
 makerat(r, numer(r1)*numer(r2),
            denom(r1)*denom(r2));
 return r;
}
rat quotrat(rat& r1, rat& r2)
{rat r;
 makerat(r, numer(r1)*denom(r2),
            denom(r1)*numer(r2));
 return r;
}
void printrat(rat& r)
{cout << numer(r) << "/" << denom(r)<< endl;
}
int main()
{rat r1, r2;
 makerat(r1, 1, 2);
 makerat(r2, 3, 4);
 printrat(sumrat(r1, r2));
 printrat(subrat(r1, r2));
 printrat(multrat(r1, r2));
 printrat(quotrat(r1, r2));
 return 0;
}
```

Реализирането на подхода абстракция със структури от данни в Задача 102, показва следните четири нива на абстракция:

- Използване на рационалните числа в проблемна област (във функцията main);
- Реализиране на правилата за рационално-числова аритметика (sumrat, subrat, multrat, quotrat, printrat);
- Избор на представяне на рационалните числа и реализиране на примитивни конструктори и селектори (makerat, numer и denom);

- Работа на ниво структура.

Използването на подхода прави програмите по-лесни за описание и модификация. Наистина, ако разгледаме по-внимателно изпълнението на горната програма, забелязваме, че тя има редица недостатъци, но основният е, че не съкращава рационални числа. За да поправим този недостатък, се налага да променим единствено функцията makerat. За целта ще използваме помощната функция gcd, дефинирана в глава 8. Новата makerat има вида:

```
void makerat(rat& r, int a, int b)
{if (a == 0) {r.num = 0; r.den = b;}
 else
 {int g = gcd(abs(a), abs(b));
  if (a > 0 && b > 0 || a < 0 && b < 0)
  {r.num = abs(a)/g; r.den = abs(b)/g;}
  else {r.num = - abs(a)/g; r.den = abs(b)/g;}
 }
}
```

С това проблемът със съкращаването на рационални числа е решен. За разрешаването му се наложи малка модификация на програмата, която засегна само премитивния конструктор makerat. Последното илюстрира лесната модифицируемост на програмите, реализиращи горния подход.

5. От структури към класове

В тази задача дефинирахме структурата rat, определяща рационални числа, и реализирахме някой основни функции за работа с такива числа. *Възниква въпросът:* Може ли да използваме тази структура като тип данни рационално число? Отговорът е не, защото при типовете данни представянето на данните е скрито за потребителя. На последния са известни множеството от стойности и операциите и вградените функции, допустими за типа. Така възниква усещането, че трябва да се обедини представянето на рационално число като запис с две полета с примитивните операции (makerat, numer и denom), пряко използващи представянето. Последното е възможно, тъй като в езика C++ се допуска полетата на структура да са функции, разбира се от тип, различен от типа на структурата.

В следващото описание примитивните операции, реализирани чрез функциите makerat, numer и denom, а също функцията за извеждане на рационално число, ще направим полета на структутара rat. За целта реализираме следните две стъпки:

1. Включване във фигурните скоби на дефиницията на rat декларациите им

```
void makerat(rat& r, int a, int b);
int numer(rat& r);
int denom(rat& r);
void printrat(rat& r);
```

от които елеминираме участието на формалния параметър rat& r. Получаваме структурата:

```
struct rat
{int num;
 int den;
void makerat(int a, int b);
 int numer();
 int denom();
 void printrat();
};
```

представяща запис с две полета num и den, над които могат да се изпълняват функциите:

- makerat, която конструира рационалното число a/b;
- numer, която намира числителя на рационално число;
- denom, която намира знаменателя на рационално число;
- printrat, която извежда рационално число.

2. Отразяване на тези промени в дефинициите на функциите. За целта ще изтрием участията на rat& r и r., а между типа и името на всяка функция ще поставим името на структурата rat, следвано от оператора ::.

Получаваме:

```
void rat::makerat(int a, int b)
{if (a == 0) {num = 0; den = b;}
else
{int g = gcd(abs(a), abs(b));
 if (a > 0 && b > 0 || a < 0 && b < 0)
 {num = abs(a)/g; den = abs(b)/g;}
```

```
else {num = - abs(a)/g; den = abs(b)/g;}
}
}
int rat::numer()
{return num;
}
int rat::denom()
{return den;
}
void rat::printrat()
{cout << num << "/" << den << endl;
}
```

Тези функции се наричат **член-функции на структурата rat.** Извикването им се осъществява като полета на структура.

Пример:

```
rat r;              // дефиниция на променлива от тип rat
r.makerat(1, 5)     // r се свързва с рационалното число 1/5
r.numer()           // намира числителя на r, в случая 1
r.denom()           // връща знаменателя на r, в случая 5
r.printrat()        // извежда върху екрана r.
```

Обръщението r.numer() е еквивалентно на изпълнение на оператора return r.num;

Програма Zad102_1.cpp реализира последните промени.

Program Zad102_1.cpp

```
#include <iostream.h>
#include <math.h>
struct rat
{int num;
 int den;
 void makerat(int, int);
 int numer();
 int denom();
 void printrat();
};
void rat::printrat()
```

```
{cout << num << "/" << den << endl;
}
int gcd(int a, int b)
{while (a!=b)
 if (a > b) a = a-b; else b = b-a;
 return a;
}
void rat::makerat(int a, int b)
{if (a == 0) {num = 0; den = b;}
 else
 {int g = gcd(abs(a), abs(b));
  if (a>0 && b>0 || a<0 && b < 0)
  {num = abs(a)/g; den = abs(b)/g;}
  else {num = - abs(a)/g; den = abs(b)/g;}
 }
}

int rat::numer()
{return num;
}
int rat::denom()
{return den;
}
rat sumrat(rat& r1, rat& r2)
{rat r; r.makerat(r1.numer() * r2.denom() +
                  r2.numer() * r1.denom(),
                  r1.denom() * r2.denom());
 return r;
}
rat subrat(rat& r1, rat& r2)
{rat r; r.makerat(r1.numer() * r2.denom() -
                  r2.numer() * r1.denom(),
                  r1.denom() * r2.denom());
 return r;
}
rat multrat(rat& r1, rat& r2)
{rat r; r.makerat(r1.numer()*r2.numer(),
                  r1.denom()*r2.denom());
 return r;
```

```
}
rat quotrat(rat& r1, rat& r2)
{rat r; r.makerat(r1.numer()*r2.denom(),
                  r1.denom()*r2.numer());
 return r;
}

int main()
{rat r1; r1.makerat(-1,2);
 rat r2; r2.makerat(3,4);
 sumrat(r1, r2).printrat();
 // или rat r = sumrat(r1, r2); r.print();
 subrat(r1, r2).printrat();
 // или r = subrat(r1, r2); r.printrat();
 multrat(r1, r2).printrat();
 // или r = multrat(r1, r2); r.printrat();
 quotrat(r1, r2).printrat();
 // или r = quotrat(r1, r2); r.printrat();
 return 0;
}
```

Забелязваме, че във функциите sumrat, subrat, multrat и quotrat не се използват полетата на записа num и den, но ако направим опит за използването им даже на ниво main, опитът ще бъде успешен. Последното може да се забрани, ако се използва етикетите private: пред дефиницията на полетата num и den и public: пред декларациите на член-функциите. Структурата rat получава вида:

```
struct rat
{private:
 int num;
 int den;
 public:
 void makerat(int, int);
 int numer();
 int denom();
 void printrat();
};
```

Опитът за използването на полетата num и den на структурата rat извън член-функциите предизвиква грешка.

Етикетите private и public се наричат **спецификатори за достъп.** Всички член-данни, следващи спецификатора за достъп private, са достъпни само за член-функциите от дефиницията на структурата. Всички член-данни и член-функции, следващи спецификатора за достъп public, са достъпни за всяка функция, която е в областта на структурата. Ако специфитаторите за достъп с пропуснати, всички членове са public. Един и същ спецификатор за достъп може да се използва повече от веднаж в една и съща дефиниция на структура.

Така специфицирането на num и den като private прави невъзможно използването им извън член-функциите makerat, numer, denom и printrat.

Ако заменим запазената дума struct със class, последната програма не променя поведението си. Така дефинирахме първия си клас с име rat, създадохме два негови обекта – рационалните числа r1 и r2 и работихме с тях.

Тези идеи са в основата на нов подход за програмиране – обектно – ориентирания.

Задачи

Задача 1. Да се напише функция, която намира разстоянието между две точки в равнината. Като се използва тази функция, да се напише програма, която въвежда координатите на n точки от равнината, намира и извежда най-голямото разстояние между тях. За целта да се дефинира структура, определяща точка от равнината с координати (x, y).

Задача 2. Да се напише програма, която въвежда факултетните номера, имената и успеха по k предмета на студентите от една група и извежда следната таблица:

```
N  име      предмет1 ...   предметK    среден успех
==============================================================
.   .       .            .       .
.   .       .            .       .
.   .       .            .       .
==============================================================
         ср. успех    ...  ср. успех      ср. успех
```

Задача 3. Да се напише:

а) булева функция equal(rat x, rat y), която установява дали рационалните числа x и y са равни.

б) булева функция grthen(rat x, rat y), която установява дали рационалното число x е по-голямо от рационалното число y.

в) функция maxrat(int n, rat x[]), която намира най-голямото от рационалните числа на масива x.

г) функция sortrat(int n, rat x[]), която сортира елементите на редицата x_0, x_1, ..., x_{n-1}.

Задача 4. Да се напише програма, която решава системата уравнения

$$a x + b y = e$$
$$c x + d y = f$$

където коефициентите a, b, c, d, e, f, а също и неизвестните x и y са рационални числа.

Задача 5. Нека a_0, a_1, ..., a_{n-1} и x са рационални числа. Да се напише функция, която намира стойността на полинома

$P(x) = a_0x^n + a_1x^{n-1} + ... + a_n$.

Задача 6. Да се дефинира структура, определяща точка от равнината с координати (x, y), където x и y приемат за стойности числата от 1 до 100. Да се напише програма, която чете координатите на четири точки, представляващи върховете A, B, C и D на четириъгълник в цикличен ред и определя дали ABCD е квадрат, правоъгълник или друга фигура.

Задача 7. Да се напише програма, която решава системата уравнения

$$a x + b y = e$$
$$c x + d y = f$$

където коефициентите a, b, c, d, e, f, а също и неизвестните x и y са комплексни числа.

Задача 8. Нека a_0, a_1, ..., a_{n-1} и x са комплексни числа. Да се напише функция, която намира стойността на полинома

$P(x) = a_0x^n + a_1x^{n-1} + ... + a_n$.

Задача 9. Дадени са естественото число n и комплексното число z. Да се напише програма, която пресмята стойността на следната комплексна функция:

$$F1 = 1 + \frac{z}{1!} + \frac{z^2}{2!} + ... + \frac{z^n}{n!}$$

Допълнителна литература

1. B. Stroustrup, C++ Programming Language. Third Edition, Addison – Wesley, 1997.
2. Ал Стивънс, Кл. Уолнъм, C++ библия, АЛЕКС СОФТ, С., 2000.
3. М. Тодорова, Езици за функционално и логическо програмиране – функционално програмиране, СОФТЕХ, С., 1998.
4. Ст. Липман, Езикът C++ в примери, КОЛХИДА ТРЕЙД КООП, С., 1993.
5. Ч. Сфар, Visual C++ 6.0, том 1, СОФТПРЕС, С. 2000.

```
#include <iostream.h>
    #include <iomanip.h>
    #include <string.h>
    const NUM = 5;
    class student
    {private:
     int facnom;
     char name[26];
     public:
     double marks[5];
     void read_student();
     void print_student();
    };

    void sorttable(int n, student[]);
    double average(double*);

    int main()
    {cout << setprecision(2) << setiosflags(ios::fixed);
     student table[30];
     int n;
     do
     {cout << "number of students? ";
      cin >> n;
     }while (n < 1 || n > 30);
     int i;
     for (i = 0; i <= n-1; i++)
       table[i].read_student();
     cout << "table: \n";
     for (i = 0; i <= n-1; i++)
     {table[i].print_student();
      cout << endl;
     }
     sorttable(n, table);
```

```
 cout << "\n New Table: \n";
 for (i = 0; i <= n-1; i++)
 {table[i].print_student();
  cout << setw(7) << average(table[i].marks) << endl;
 }
 return 0;
}
void student::read_student()
{cout << "fak. nomer: ";
 cin >> facnom;
 char p[100];
 cin.getline(p, 100);
 cout << "name: ";
 cin.getline(name, 40);
 for (int i = 0; i <= NUM-1; i++)
 {cout << i << " -th mark: ";
  cin >> marks[i];
 }
}

void student::print_student()
{cout << setw(6) << facnom << setw(30) << name;
 for (int i = 0; i <= NUM-1; i++)
 cout << setw(6) << marks[i];
}

void sorttable(int n, student a[])
{for (int i = 0; i <= n-2; i++)
 {int k = i;
  double max = average(a[i].marks);
  for (int j = i+1; j <= n-1; j++)
   if (average(a[j].marks) > max)
   {max = average(a[j].marks);
    k = j;
```

```
    }
   student x = a[i]; a[i] = a[k]; a[k] = x;
  }
 }

 double average(double* a)
 {double s = 0;
  for (int j = 0; j <= NUM-1; j++)
    s += a[j];
  return s/5;
}
```

Глава 13

Рекурсия
(допълнение)

1. Синтактичен анализ и намиране на стойност на изрази

В много случаи синтаксисът на различни езикови конструкции има рекурсивна структура. За формалното описание на такива конструкции се използва широкоразпространения език на Бекус-Наур. Например, цяло число без знак можем да опишем по следните два начина:

```
<цяло_без_знак> ::=  <цифра>|
                     <цяло_без_знак><цифра>
```

или

```
<цяло_без_знак> ::=  <цифра>|
```

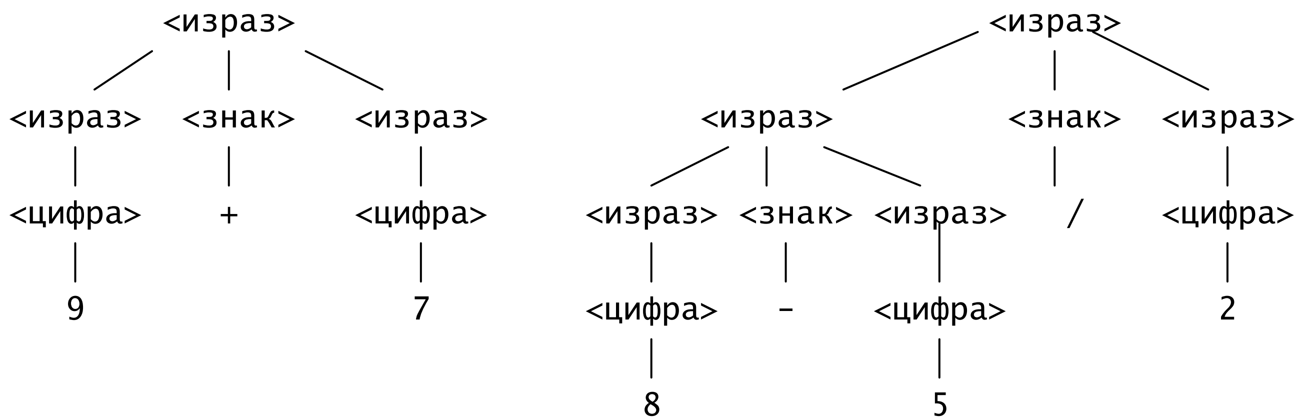
<цифра><цяло_без_знак>

Да се направи синтактичен анализ на символен низ според някакви правила означава да се провери дали низът е получен според тези правила и се определи видът на съставящите го части и връзките между тях. За целта се конструира т. нар. **дърво на синтактичния разбор.**

Например символните низове (9+7) и ((8-5)/2) са изрази според правилата:

<израз> ::= <цифра>|(<израз><знак><израз>)
<знак> ::= +|-|*|/;
<цифра> ::= 0|1|...|9

и съответните дървета на синтактичен разбор са:



Задачата на синтактичния анализ не се състои в рисуване на такива дървета, а в анализ на текста и ако е правилен, да се конструира някаква структура, която определя вида на съставящите низа части и връзките между тях. В горния случай следните структури определят дърветата на синтактичния разбор:

expr(sign(+), expr(digit(9)), expr(digit(7)))
expr(sign(/), expr(sign(-), expr(digit(8)), expr(digit(5))),
expr(digit(2))).

В тази част няма да разгледаме в пълнота задачата за синтактичния анализ на изрази. Ще дискутираме само проверката дали даден символен низ е израз в смисъла на дадени правила. Тъй като дефинициите на тези правила обикновено са рекурсивни, най-естествените решения са рекурсивните.

Задача 110. Да се състави рекурсивна програма, която проверява дали низ, въведен от клавиатурата *започва* с израз, определен от правилата:

<израз> ::= <цифра>|(<израз><знак><израз>)

<знак> ::= +|-|*|/;

<цифра> ::= 0|1|...|9.

Програма Zad110.cpp решава задачата. В нея са дефинирани булевите функции:

bool formula1(); - проверява дали въведеният низ започва с формула;

bool digit(char); - проверява дали символ е цифра;

bool sign(char); - проверява дали символ е знак, според указаното правило.

Program Zad110.cpp

```
#include <iostream.h>
bool formula1();
bool digit(char);
bool sign(char);
int main()
{if (formula1()) cout << "yes \n";
 else cout << "no\n";
 return 0;
}
bool digit(char c)
{return c>='0' && c<='9';
}
bool sign(char c)
{return c=='+' || c== '-' || c=='*' || c== '/';
}
bool formula1()
{char c;
 cin >> c;
 if (c != '(') return digit(c);
 bool yes = formula1();
 if (!yes) return false;
 cin >> c;
 if (!sign(c)) return false;
 yes = formula1();
```

```
 cin >> c;
 return yes && c==')';
}
```

Забележка: Ще отбележим още веднаж, че програмата отговаря положително, както за низа 8, така и за низовете 88 и 8а, както за низа (3+9), така и за низа (3+9)а-5.

Задачата за правилността на символен низ относно дадени правила ще решаваме по аналогичен начин, но чрез индекс ще следим до коя позиция е разпозната търсената форма и отговорът ще е положителен само ако низът е изчерпен.

Задача 111. Да се състави програма, която определя дали символен низ е израз в смисъла на следните правила:

```
<израз> ::= <израз>+<терм>|
            <израз>-<терм>|
            <терм>;
<терм> ::= <терм>*<цифра>|
           <терм>/<цифра>|
           <цифра>;
<цифра> ::= 0|1| ... |9.
```

Във функцията main се въвежда символен низ, в който се “изтриват” интервалите. Четенето на отделните символи от низа се осъществява чрез символната функция getchar:

```
char getchar()
{i++;
 if (i==len) return ' ';
 else return s[i];
}
```

която използва глобалните променливи

```
int i,            // индекс на текущия символ
    len;          // дължина на символния низ s и
char s[100];      // символния низ, анализиран за израз
```

и връща сочения от индекса i символ, ако 0≤i<len и интервал, в противен случай.

Ще дефинираме следните рекурсивни функции:

bool expr() - реализира правилото

```
<израз> ::= <терм>|
            <израз>+<терм>|
            <израз>-<терм>;
bool term() - реализира правилото
<терм> ::= <цифра>|
           <терм>*<цифра>|
           <терм>/<цифра>;
```

Забелязваме, че тези дефиниции са ляво-рекурсивни, заради това че операциите +, -, * и / са ляво-асоциативни. Дословното им реализаране ще доведе до зацикляне в неграничните случаи. Този проблем е известен като **проблем на лявата рекурсия**. Промяната на асоциативността на операциите, т.е.

```
<израз> ::= <терм>+<израз>|
            <терм>-<израз>|
            <терм>;
<терм> ::= <цифра>*<терм>|
           <цифра>/<терм>|
         <цифра>;
```

в случая решава проблема на лявата рекурсия. Програма Zad111.cpp дава едно решение на задачата.

```
Program Zad111.cpp
#include <iostream.h>
#include <string.h>
char c;
int i, len;
char s[100];
char getchar()
{i++;
 if (i==len) return ' ';
 else return s[i];
}
bool expr();
bool term();
bool digit();
int main()
{cout << "Input an expression! ";
 char t[100];
 cin.getline(t, 100);
 len = strlen(t);
```

```
 i = -1;
 for (int j=0; j<=len-1; j++)
   if (t[j]!=' ')
    {i++;
     s[i] = t[j];
    }
 len = i+1;
 i = -1;
 if (expr() && i+1 == len) cout << " yes \n";
 else cout << "no\n";
 return 0;
}
bool digit()
{c = getchar();
 return c>='0' && c<='9';
}
bool term()
{bool yes = digit();
 if (!yes) return false;
 c = getchar();
 if (c!='*' && c!= '/')
 {i--;
  return true;
 }
return term();
}
bool expr()
{bool yes = term();
 if (!yes) return false;
 c = getchar();
 if (c!='+' && c!= '-')
 {i--;
  return true;
 }
return expr();
 }
```

Задача 112. Да се напише програма, която въвежда символен низ и ако той е правилен израз според правилата от предишната задача, пресмята стойността му.

В предишната задача проблема на лявата рекурсия решихме чрез промяна асоциативността на операциите. Това не оказа влияние на правилността на решението. Ако трябва обаче да се намери стойността на символен низ, представящ израз по новите правила, ще се получи грешен резултат (действията ще се извършват отдясно наляво).

В случая с лявата рекурсия ще се справим като запишем правилата по следния нерекурсивен начин:

<израз> ::= <терм>{<знак1><терм>}$_{opc}$

<терм> ::= <цифра>{<знак2><цифра>}$_{opc}$

<знак1> ::= +|-

<знак2> ::= *|/

Програма Zad112.cpp решава задачата.

Program Zad112.cpp

```
#include <iostream.h>
#include <string.h>
char c;
int i, len;
char s[100];
char getchar()
{i++;
 if (i==len) return ' ';
 else return s[i];
}
bool expr(double&);
bool expr1(double, char, double&);
bool term(double&);
bool term1(double, char, double&);
bool digit(double& x)
{c = getchar();
 x = (int)c-48;
 return c>='0' && c<='9';
}
int main()
{cout << "Input an expression! " ;
 char t[100];
```

```
 cin.getline(t, 100);
 len = strlen(t);
 i = -1;
 for (int j=0; j<=len-1; j++)
   if (t[j]!=' ')
    {i++;
     s[i] = t[j];
    }
 len = i+1;
 i = -1;
 double m;
 if (expr(m) && i+1==len) cout << m << " yes \n";
 else cout << "no\n";
 return 0;
}
bool term(double& rez)
{bool yes = digit(rez);
 if (!yes) return false;
 c = getchar();
 if (c==' '|| c== '+' || c== '-') {i--; return true;}
 if (c!='*' && c!= '/') return false;
 return term1(rez, c, rez);
}
 bool term1(double x, char ch, double& rez)
 {double y;
  bool yes = digit(y);
  if (!yes) return false;
  switch (ch)
  {case '*': rez = x*y; break;
   case '/': rez = x/y;
  }
  c = getchar();
  if (c==' ' || c=='+' || c=='-'){i--; return true;}
  if (c!='*' && c!='/') return false;
  return term1(rez, c, rez);
}
bool expr(double& rez)
{bool yes = term(rez);
 if (!yes) return false;
```

```
 c = getchar();
 if (c==' ') {i--; return true;}
 if (c!='+' && c!= '-') return false;
 return expr1(rez, c, rez);
}
 bool expr1(double x, char ch, double& rez)
 {double y;
  bool yes = term(y);
  if (!yes) return false;
  switch (ch)
  {case '+': rez = x+y; break;
   case '-': rez = x-y;
  }
  c=getchar();
  if (c==' '){i--; return true;}
  if (c!='+' && c!='-') return false;
  return expr1(rez, c, rez);
}
```

2. Търсене с връщане назад

При редица задачи се поставят въпроси като "Колко начина за ... съществуват?" или "Има ли начин за ...? или възниква необходимостта от намиране на всички възможни решения, т.е. да се изчерпят всички възможни варианти за решаване на дадена задача. Широкоизползван общ метод за организация на такива търсения е **методът търсене с връщане назад** (backtracking). При него всяко от решенията (вариантите) се строи стъпка по стъпка. Частта от едно решение, построена до даден момент се нарича **частично решение**.

Методът се състои в следното:

- конструира се едно частично решение;
- на всяка стъпка се прави опит текущото частично решение да се продължи;
- ако на някоя стъпка се окаже невъзможно ново разширяване, извършва се връщане назад към предишното частично решение и се прави нов опит то да се разшири (продължи) по друг начин.

- ако се стигне до решение, то се запомня или извежда и процесът на търсене продължава по същата схема, докато бъдат генерирани всички възможни решения.

Търсенето с връщане назад се осъществява като се използва механизмът на рекурсията.

Ще го илюстрираме чрез няколко примера.

Задача 113. **(Пътища в лабиринт) Квадратна мрежа има n реда и n стълба. Всяко квадратче е или проходимо, или е непроходимо. От проходимо квадратче може да се премине във всяко от четирите му съседни като се прекоси общата им страна. В непроходимо квадратче може да се влезе, но от него не може да се излезе. Да се напише програма, която намира всички пътища от квадратче (0,0) до квадратче (n-1,n-1). Пътищата (ако съществуват) да се маркират със '*'.**

Мрежата ще представим чрез квадратна матрица M от символи. Квадратче (i, j) е запълнено със символа 1, ако е непроходимо и с 0 – ако е проходимо.

Програма Zad113.cpp решава задачата. В нея функцията init реализаира въвеждане на мрежата. При това се осигурява квадратче (0,0) да е проходимо. Това не е ограничение, тъй като ако то е непроходимо, задачата няма смисъл. Функцията writelab извежда мрежата с маркирания със * път, ако съществува или съобщава, че път не съществува. Съществената част от програмата е функцията void labyrinth(int i, int j). Тя осъществява търсенето на всички пътища от квадратче (i,j) до квадратче (n-1,n-1). /Предполагаме, че сме намерили частично решение – път от квадратче (0,0) до квадратче (i,j)/. В главната функция main обръщението към нея се осъществява с фактически параметри 0 и 0. В labyrinth е реализиран следният алгоритъм:

- Ако квадратче (i,j) съвпада с (n-1,n-1) е намерен един път от квадратче (i,j) до квадратче (n-1,n-1), извежда се и или се завършва изпълнението, или се продължава търсенето на други пътища;
- Ако горното не е вярно, а квадратче (i,j) е извън мрежата или е непроходимо, labyrinth завършва изпълнението си и накрая
- Ако квадратче (i,j) е вътре в мрежата и е проходимо, частичното решение се продължава като квадратче (i,j) се

включва в пътя /маркира се със символа */ и продължава търсенето на всички възможни пътища от някое от четирите ограждащи (i,j) чрез стена квадратчета до квадратче (n-1,n-1), т.е.

```
labyrinth(i+1,j);
labyrinth(i,j+1);
labyrinth(i-1,j);
labyrinth(i,j-1);
```

Завършването на изпълнението на тези рекурсивни функции означава, че са невъзможни нови разширения, т.е. не съществуват други пътища от квадратче (i,j) до квадратче (n-1,n-1). Осъществява се връщане назад към предишното частично решение, като се отказваме квадратче (i,j) да принадлежи на пътя чрез възстановяване проходимостта му.

Program Zad113.cpp

```
#include <iostream.h>
char m[20][20];
int n;
bool way = false;

void init()
{int i, j;
 do
 {cout << "n= ";
  cin >> n;
 }while(n<1||n>20);
 do
 {cout << "labyrinth:\n";
  for (i=0; i<=n-1; i++)
  for (j=0; j<=n-1; j++)
    cin >> m[i][j];
 }while(m[0][0]!='0');
}

void writelab()
{int k, l;
 cout << endl;
 for (k=0; k<=n-1; k++)
```

```
  {for (l=0; l<=n-1; l++)
     cout << m[k][l] << " ";
   cout << endl;
  }
}
void labyrinth(int i, int j)
{if (i==n-1 && j==n-1){m[i][j]='*'; way = true; writelab();}
 else
   if (i>=0 && i<=n-1 && j>=0 && j<=n-1)
   if(m[i][j] == '0')
   {m[i][j]='*';
    labyrinth(i+1,j);
    labyrinth(i,j+1);
     labyrinth(i-1,j);
     labyrinth(i,j-1);
     m[i][j] = '0';
   }
}

int main()
{init();
 labyrinth(0,0);
 if(!way) cout << "no\n";
 return 0;
}
```

Задача 114. **Квадратна мрежа има n реда и n стълба. Всяко квадратче е или проходимо, или е непроходимо. От проходимо квадратче може да се премине във всяко от четирите му съседни като се прекоси общата им страна. В непроходимо квадратче може да се влезе, но от него не може да се излезе. Да се напише програма, която намира най-кратък път от квадратче (0,0) до квадратче (n-1,n-1). Пътят (ако съществува) да се маркира със '*'.**

Програма Zad114.cpp решава задачата. Чрез алгоритъм, аналогичен на този от предишната задача, се генерират всички възможни пътища от квадратче (0,0) до квадратче (n-1,n-1). От тях се избира един с най-малка дължина (брой квадратчета, включени в него). Освен масива M, програмата поддържа и двумерен масив P, в който пази мрежата с текущия най-

кратък път. Тези структури са описани като глобални променливи. Като глобални са дефинирани и n – размерност на мрежата, bmin – дължина на текущия минимален път, b – дължина на текущо конструирания път и way – булева променлива, индицираща съществуването на път от квадратче (0,0) до квадратче (n-1,n-1).

```
Program Zad114.cpp
#include <iostream.h>
char p[20][20];
char m[20][20];
int n, bmin=0, b=0;
bool way=false;
void init()
{int i, j;
 do
 {cout << "n= ";
  cin >> n;
 }while(n<1||n>20);
 do
 {cout << "labirint:\n";
  for (i=0; i<=n-1; i++)
  for (j=0; j<=n-1; j++)
    cin >> m[i][j];
 }while(m[0][0]!='0');
}
void opt()
{int k, l;
 if (b<bmin || bmin == 0)
 {bmin = b;
  for (k=0; k<=n-1; k++)
     for (l=0; l<=n-1; l++)
        p[k][l]=m[k][l];
 }
}
void writelab()
{int k, l;
 if(!way) cout << "no way\n";
 else
 {cout << "yes \n";
```

```
      for (k=0; k<=n-1; k++)
      {for (l=0; l<=n-1; l++)
         cout << p[k][l] << " ";
       cout << endl;
      }
    }
  }
  void labirint(int i, int j)
  {if (i==n-1 && j==n-1){way=true; m[i][j]='*'; opt();}
   else
     if (i>=0 && i<=n-1 && j>=0 && j<=n-1)
     if(m[i][j]=='0')
     {m[i][j]='*';
       b++;
       labirint(i+1,j);
       labirint(i,j+1);
       labirint(i-1,j);
       labirint(i,j-1);
       m[i][j] = '0';
       b--;
     }
  }
  int main()
  {init();
   labirint(0, 0);
   writelab();
   return 0;
  }
```

Задача 115. Дадени са n града (n е естествено число, $1 \le n \le 10$) и целочислена матрица Anxn, така че a_{ij} е равно на 1, ако има пряк път от град i до град j и е 0 в противен случай ($0 \le i,j \le n-1$). Да се напише програма, която намира и извежда всички ациклични пътища между два произволно зададени града в случай, че път между тях съществува.

Програма Zad115.cpp решава задачата. Ацикличен път е път, състоящ се от различни градове. Процедурата foundallway намира всички ациклични пътища от град i до град j в случай, че път

съществува. Всеки път се записва в глобалния едномерен масив int x[100], а дължината му - в глобалната променлива s. Глобалната булева променлива way получава стойност true, ако съществува път между двата зададени града. Инициализирана е с false.

```
Program Zad115.cpp
#include <iostream.h>
int arr[10][10]={0};
int n, s=-1;
int x[100];
bool way = false;
void writeway()
{for (int i = 0; i<=s; i++)
   cout << x[i] << " ";
  cout << endl;
}

bool member(int x, int n, int* a)
{if (n == 1 ) return a[0] == x;
  return x == a[0] || member(x, n-1, a+1);
}
void foundallway(int i, int j)
{if (i==j){way = true; s++; x[s] = i; writeway();}
  else
    {s++; x[s] = i;
     for(int k = 0; k <= n-1; k++)
     if(arr[i][k] == 1 && !member(k, s+1, x))
     {arr[i][k] = 0; arr[k][i] = 0;
      foundallway(k,j);
      arr[i][k]=1; arr[k][i]=1;
      s--;
     }
    }
}
int main()
{do
  {cout << "n= ";
   cin >> n;
  }while (n < 1 || n > 10);
```

```
  for (i = 0; i <= n-2; i++)
    for (int j = i+1; j <= n-1; j++)
    {cout << "connection between " <<  i << " and "
        << j << " 0/1? ";
    cin >> arr[i][j];
    arr[j][i] = arr[i][j];
    }
  int j;
  do
  {cout << "start and final towns: ";
   cin >> i >> j;
  }while (i<0||i>9||j<0||j>9);
   foundallway(i,j);
   if (!way) cout << "no\n";
   return 0;
 }
```

Задача 116. Дадени са n града (n е естествено число, 1≤n≤10) и целочислена матрица Anxn, така че a_{ij} е равно на 1, ако има пряк път от град i до град j и е 0 в противен случай (0≤i,j≤n-1). Да се напише програма, която намира и извежда ацикличен път с минимална дължина между два произволно зададени града в случай, че път съществува.

Използван е подход аналогичен на този в Задача 114.

```
Program Zad116.cpp
#include <iostream.h>
int arr[10][10]={0};
int n, s=-1, smin = 0;
int x[100], xmin[100];
bool way = false;

void writeway()
{if (!way) cout << "no way\n";
 else
 {cout << "yes\n";
  for (int i = 0; i<=smin-1; i++)
   cout << xmin[i] << " ";
```

```
  cout << endl;
 }
}
void opt()
{if(s+1<smin || smin==0)
 {smin = s+1;
  for(int i = 0; i <= smin-1; i++)
    xmin[i] = x[i];
 }
}
bool member(int x, int n, int* a)
{if (n == 1 ) return a[0] == x;
 return x == a[0] || member(x, n-1, a+1);
}
void foundminway(int i, int j)
{if (i==j){way = true; s++; x[s] = i; opt();}
 else
   {s++; x[s] = i;
    for(int k = 0; k <= n-1; k++)
    if(arr[i][k] == 1 && !member(k, s+1, x))
    {arr[i][k] = 0; arr[k][i] = 0;
      foundminway(k,j);
      s--;
     arr[i][k]=1; arr[k][i]=1;
    }
   }
}
int main()
{do
 {cout << "n= ";
  cin >> n;
 }while (n < 1 || n > 10);
 for (i = 0; i <= n-2; i++)
   for (int j = i+1; j <= n-1; j++)
   {cout << "connection between " <<  i << " and "
       << j << " 0/1? ";
   cin >> arr[i][j];
   arr[j][i] = arr[i][j];
   }
```

```
 int j;
 do
 {cout << "start and final towns: ";
  cin >> i >> j;
 }while (i < 0 || i > 9 || j < 0 || j > 9);
  foundminway(i,j);
  writeway();
  return 0;
}
```

Задачи

Задача 1. Да се направи програма, която анализира символен низ въведен от клавиатурата и определя дали той е израз в смисъла на следната граматика:

<израз> ::= <израз>+<терм>|
<израз>-<терм>|
<терм>;

<терм> ::= <терм>*<буква>|
<терм>/<буква>|
<буква>;

<буква> ::= a|b|c ... |z.

Задача 2. Да се напише програма, която проверява дали символен низ, въведен от клавиатурата е израз съгласно формулата:

<израз> ::= <цифра>|
f(<израз>)|s(<израз>)|p(<израз>)

<цифра> ::= 0|1|...|9

и ако това е така, пресмята стойността на израза, ако f(x), s(x) и p(x) намират съответно x!, x+1 и x-1.

Задача 3. Да се състави програма, която проверява дали низ е израз, определен от формулите:

<израз> ::= <цифра>|(<израз><знак><израз>)

<знак> ::= +|-|*|/;

<цифра> ::= 0|1|...|9

и ако това е така, намира стойността му.

Задача 4. Да се напише програма, която проверява дали символен низ, въведен от клавиатурата е израз съгласно формулата:

```
<израз> ::= <цифра>|
            pow(<израз>,<израз>)|gcd(<израз>,<израз>)
<цифра> ::= 0|1|...|9
```

и ако това е така, пресмята стойността на израза, ако pow(x,y) и gcd(x,y) намират съответно x на степен y и най-големия общ делител на x и y (Ако някое от числата x или y е 0, за gcd(x,y) да се приеме другото или 0, ако и x, и y са 0).

Задача 5. **Квадратна мрежа има n реда и n стълба. Всяко квадратче е или проходимо, или е непроходимо. От проходимо квадратче може да се премине във всяко от четирите му съседни като се прекоси общата им страна. В непроходимо квадратче може да се влезе, но от него не може да се излезе. Да се напише програма, която намира най-краткия път от квадратче (0, 0) до квадратче (n-1,n-1), който минава през произволно зададено проходимо квадратче на мрежата. Пътят да се маркира със '*' (ако съществува).**

Задача 6. Дадени са n града (n е естествено число, $1 \le n \le 10$) и целочислена матрица Anxn, така че a_{ij} е равно на 1, ако има пряк път от град i до град j и е 0 в противен случай ($0 \le i,j \le n-1$). Да се напише програма, която намира и извежда всички ациклични пътища с указана дължина между два произволно зададени града в случай.

Задача 7. Дадени са n града (n е естествено число, $1 \le n \le 10$) и целочислена матрица Anxn, така че a_{ij} е равно на 1, ако има пряк път от град i до град j и е 0 в противен случай ($0 \le i,j \le n-1$). Ацикличен път, в който началния и крайния град съвпадат, се нарича цикъл. Да се напише програма, която намира и извежда всички цикли за произволно зададен град в случай, че цикъл съществува.

Задача 8. Дадени са n града (n е естествено число, $1 \le n \le 10$) и целочислена матрица Anxn, така че a_{ij} е равно на 1, ако има пряк път от град i до град j и е 0 в противен случай ($0 \le i,j \le n-1$). Ацикличен път, в който минава през всички върхове, ще наричаме Хамилтонов цикъл. Да се напише програма, която намира и извежда всички Хамилтонови цикли за зададените градове.

Допълнителна литература

1. Рейнгольд З., Нивергальт Н. Део, Комбинаторные алгоритмы. Теория и практика, М., Мир, 1980.

2. Узерелл Ч., Этюдыдля программистов, М. Мир, 1982.

3. Тодорова М., Езици за функционално и логическо програмиране. Логическо програмиране, С., СОФТЕХ, 1998.

Глава 14

Класове

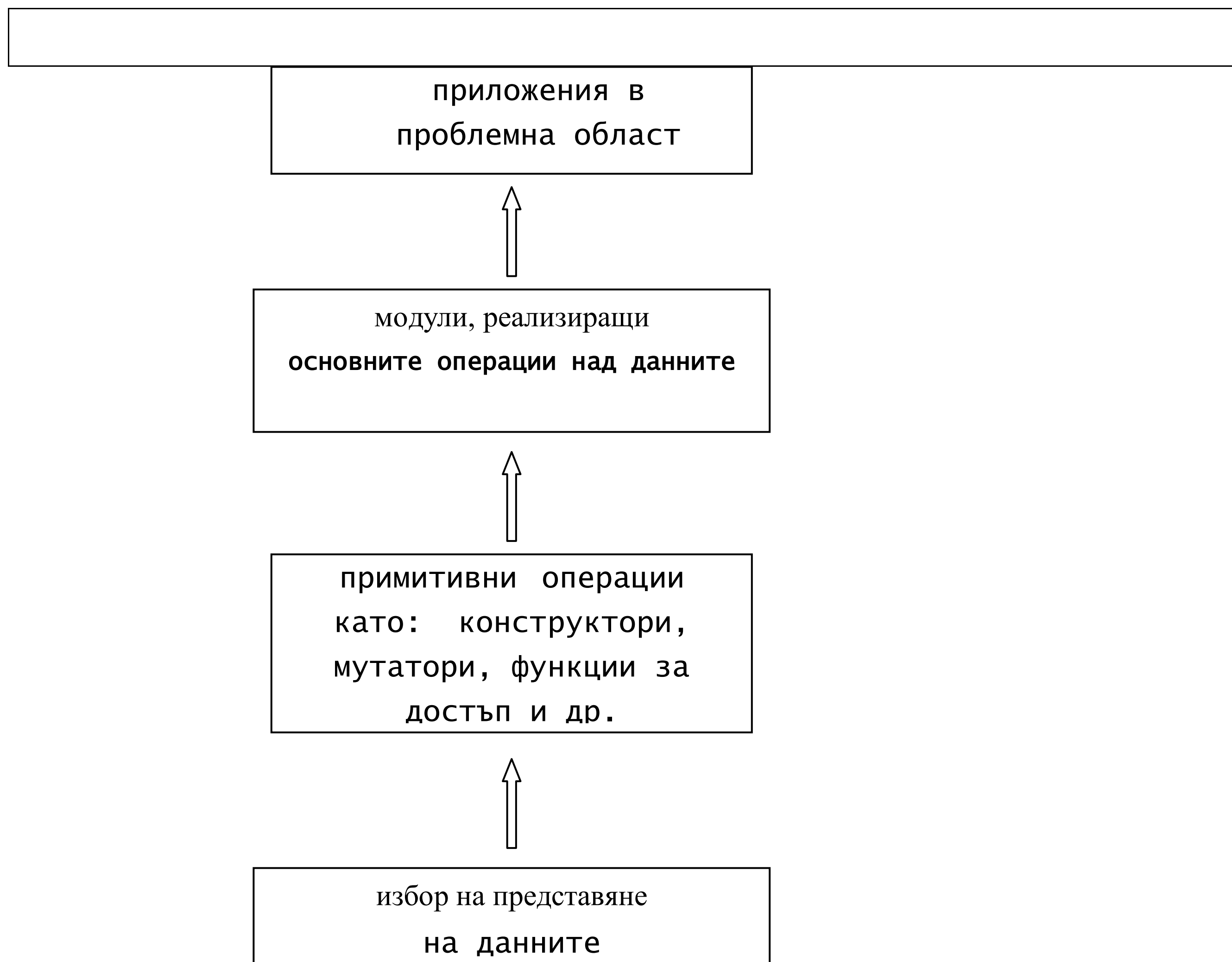
Класовете са нови типове от данни, дефинирани от потребителя. Те могат да обогатяват възможностите на вече съществуващ тип или да представят напълно нов тип.

Класовете са подобни на структурите, даже може да се каже, че в някои отношения са почти идентични. В C++ класът може да се разглежда като структура, на която са наложени някои ограничения по отношение на правата на достъп. Отначало ще разгледаме основното, което е приложимо както за класовете, така и за структурите. Затова ще останем в познатите означения на структурите.

1. Пример за програма, която дефинира и използва клас

Основен принцип на процедурното програмиране е модулния. Програмата се разделя на “подходящи” взаимносвързани части (функции, модули), всяка от които се реализира чрез определени средства. Важен е обаче начинът, по който да стане определянето на частите и връзките помежду им. Целта е, следващи промени в представянето на данните да не променят голям брой от модулите на програмата. Разсъждения в тази посока довеждат до подхода *абстракция със структури от данни*, който вече разгледахме. Ще напомним, че при него методите за използване на данните се разделят от методите за тяхното конкретно представяне. Програмите се конструират така, че да работят с “абстрактни данни” – данни с неуточнено представяне. След това представянето се конкретизира с помощта на множество функции, наречени **конструктори, мутатори** и **функции за достъп**, които реализират “абстрактните

данни” по конкретен начин. Така при решаването на даден проблем се оформят следните нива на абстракцията:



Добра реализация на подхода е тази, при която всяко ниво използва единствено средствата на предходното. Предимствата са, че възникнали промени на едно ниво ще се отразят само на следващото над него. Например, промяна на представянето на данните ще доведе до промени единствено на реализацията на някои от конструкторите, мутаторите или функциите за достъп.

Да се върнем към задачата за рационално-цифрова аритметика. Като използваме подхода абстракция със структури от данни искаме да дефинираме тип данни “рационално число”, след което да го използваме за събиране, изваждане, умножение и деление на рационални числа.

След анализ на правилата, реализиращи тези операции, в глава 11 стигнахме до необходимостта от реализирането на следните примитивни функции за работа с рационални числа:

- конструиране на рационално число по зададени две цели числа, представящи съответно неговите числител и знаменател;
- извличане на числителя на дадено рационално число;
- извличане на знаменателя на дадено рационално число.

Към тях ще добавим и функциите:

- промяна на стойността на рационално число чрез въвеждане, например;
- извеждане на рационално число.

Реализирането на подхода абстракция със структури от данни в този случай показва следните четири нива на абстракция

Ще започнем с реализирането на нивата отдолу нагоре.

избор на представяне на рационално число

Тъй като рационалното число е частно на две цели числа, можем да го определим чрез структурата:

```
struct rat
{int numer;
 int denom;
};
```

където полето numer означава числителя, а полето denom – знаменателя. Тези две полета се наричат **член-данни**, само **данни** или още **абстрактни данни** на структурата. Те определят множеството от стойности на типа rat, който дефинираме. Трябва да добавим и някакви операции и вградени функции, които да могат да се изпълняват над данни от тип rat. Това ще постигнем с реализацията на следващите две нива на абстракция, определени по-горе.

реализиране на примитивните операции

Като компоненти на структурата rat ще добавим набор от примитивни операции: конструктори, мутатори и функции за достъп. Ще ги реализираме като член-функции.

а) конструктори

Конструкторите са член-функции, чрез които се инициализират променливите на структурата. Те винаги имат за име името на структурата. Ще дефинираме два конструктора:

rat() – конструктор без параметри и

rat(int, int) – конструктор с два цели параметъра.

Първият конструктор се нарича още **конструктор по подразбиране.** Използва се за инициализиране на променлива от тип rat, когато при дефиницията й не са зададени параметри.

Ще го дефинираме така:

```
rat::rat()
{numer = 0;
 denom = 1;
}
```

Пример: След дефиницията

```
rat p = rat();
```

или съкратено

```
rat p;
```

p се инициализира с рационалното число 0/1.

Вторият конструктор

```
rat::rat(int x, int y)
{numer = x;
 denom = y;
}
```

позволява променлива величина от тип rat да се инициализира с указана от потребителя стойност.

Примери: След дефиницията

```
rat p=rat(1,3);
```

p се инициализира с 1/3, а дефиницията

```
rat q(2,5);
```

инициализира q с 2/5.
Последният запис е съкращение на

```
rat q=rat(2,5);
```

Ще отбележим, че и двата конструктора имат едно и също име, но се различават по броя на параметрите си. В този случай се казва, че функцията rat е **предефинирана.**

Декларацията на структура може да съдържа, но може и да не съдържа конструктори.

б) мутатори

Това са функции, които променят данните на структурата. Ще дефинираме мутатора read(), който въвежда от клавиатурата две цели числа (второто различно от нула) и ги свързва с абстрактните данни numer и denom.

```
void rat::read()
{cout << "numer: ";
 cin >> numer;
 do
 {cout << "denom: ";
  cin >> denom;
 }while(denom==0);
}
```

След обръщението

```
p.read();
```

стойността на p *се променя* като полетата ѝ numer и denom се свързват с въведените от потребителя стойности за числител и знаменател съответно.

в) функции за достъп

Тези функции **не променят** член-данните на структурата, а само извличат информация за тях. Последното е указано чрез използването на запазената дума const, записана след затварящата скоба на формалните параметри и пред знака ;. Ще дефинираме следните функции за достъп:

```
int get_numer() const;
int get_denom() const;
void print() const;
```

Първата от тях извлича числителя, втората – знаменателя, а третата извежда върху екрана рационалното число numer/denom. Реализациите им имат вида:

```
int rat::get_numer() const
{return numer;
}
int rat::get_denom() const
{return denom;
}
void rat::print() const
{cout << numer << "/" << denom << endl;
}
```

След включване на прототипите на тези конструктори, мутатори и функции за достъп във фигурните скоби на дефиницията на структурата rat, получаваме:

```
struct rat
{int numer;
 int denom;
// конструктори
 rat();
 rat(int, int);
// мутатор
 void read();
// функции за достъп
 int get_numer() const;
 int get_denom() const;
 void print() const;
```

```
};
```

След направените дефиниции са възможни следните действия над рационални числа:

```
// p се инициализира с 0/1, q – с 1/6, а r – с 5/9
      rat p, q(1,6), r=rat(5,9);
 // p се извежда чрез данновите полета на структурата rat
 cout << p.numer << "/"
           << p.denom << endl;
 // q се извежда като се използват
 // функциите за достъп до компонентите му
      cout << q.get_numer() << "/"
               << q.get_denom() << endl;
// q се извежда чрез функцията за достъп print()
       q.print();
 // p се модифицира чрез мутатора read()
       p.read();
```

С това завършихме реализирането на двете най-долни нива на абстракция.

реализиране на правилата за рационално-цифрова аритметика

Като използваме дефинираните конструктори, мутатори и функции за достъп, ще реализираме функциите:

```
rat sum(rat const &, rat const &);
rat sub(rat const &, rat const &);
rat prod(rat const &, rat const &);
rat quot(rat const &, rat const &);
```

извършващи рационално-числовата аритметика. Функцията sum може да се дефинира по следния начин:

```
rat sum(rat const& r1, rat const& r2)
{rat r(r1.get_numer()*r2.get_denom() +
    r2.get_numer()*r1.get_denom(),
    r1.get_denom()*r2.get_denom());
return r;
}
```

Другите функции се реализират по аналогичен начин.

Ще отбележим, че по подразбиране, членовете на структурата (член-данни и член-функции) са видими навсякъде в областта на структурата. Това позволява член-данните да бъдат използвани както от примитивните конструктори, мутатори и функции за достъп така и от функциите, реализиращи рационално-числова аритметика.

Например, функцията sum, дефинирана по-горе може да се реализира и така:

```
rat sum(rat const& r1, rat const& r2)
{rat r(r1.numer*r2.denom + r2.numer*r1.denom,
   r1.denom*r2.denom);
return r;
}
```

Нещо повече, освен чрез мутаторите, член-данните могат да бъдат модифицирани и от външни функции.

Последното противоречи на идеите на подхода абстракция със структури от данни, в основата на който лежи независимостта на използването от представянето на структурата от данни. Това води до идеята да се забрани на модулите от трето и четвърто ниво пряко да използват средствата от първо ниво на абстракция.

Езикът C++ позволява да се ограничи свободата на достъп до членовете на структурата като се поставят подходящи спецификатори на достъп в декларацията й. Такива спецификатори са private и public. Записват се като етикети. Всички членове, следващи спецификатора на достъп private, са достъпни само за член-функциите в декларацията на структурата. Всички членове, следващи спецификатора на достъп public, са достъпни за всяка функция, която е в областта на структурата. Ако са пропуснати спецификаторите на достъп, всички членове са public (както е в случая). Има още един спецификатор на достъп – protected, който е еднакъв със спецификатора private, освен ако структурата не е част от йерархия на класовете, което ще разгледаме по-късно.

С цел реализиране на идеите на подхода абстракция със структури от данни, ще променим дефинираната по-горе структура по следния начин:

```
struct rat
{private:
 int numer;
 int denom;
 public:
 rat();
```

```
 rat(int, int);
 void read();
 int get_numer() const;
 int get_denom() const;
 void print() const;
};
```

По такъв начин позволяваме член-данните numer и denom да се използват единствено от член-функциите. Операторът

```
cout << p.numer << "/"
     << p.denom << endl;
```

вече е недопустим.

Следва програмата, която решава задачата.

```
#include <iostream.h>
struct rat
{private:
 int numer;
 int denom;
 public:
 rat();
 rat(int, int);
 void read();
 int get_numer() const;
 int get_denom() const;
 void print() const;
};
rat::rat()
{numer = 0;
 denom = 1;
}
rat::rat(int x, int y)
{numer = x;
 denom = y;
}
void rat::read()
```

```
{cout << "numer: ";
 cin >> numer;
 do
 {cout << "denom: ";
  cin >> denom;
 }while(denom==0);
}
int rat::get_numer() const
{return numer;
}
int rat::get_denom() const
{return denom;
}
void rat::print() const
{cout << numer << "/" << denom << endl;
}
rat sum(rat const &, rat const &);
rat sub(rat const &, rat const &);
rat prod(rat const &, rat const &);
rat quot(rat const &, rat const &);
int main()
{rat p(1,4), q(1,2);
 p.print();
 q.print();
 cout << "sum:\n";
 sum(p,q).print();
 cout << "subtraction:\n";
 sub(p,q).print();
 cout << "product:\n";
 prod(p,q).print();
 cout << "quotient:\n";
 quot(p,q).print();
 return 0;
}
```

```
rat sum(rat const& r1, rat const& r2)
{rat r(r1.get_numer()*r2.get_denom()+
    r2.get_numer()*r1.get_denom(),
    r1.get_denom()*r2.get_denom());
return r;
}
rat sub(rat const & r1, rat const & r2)
{rat r(r1.get_numer()*r2.get_denom()-
    r2.get_numer()*r1.get_denom(),
    r1.get_denom()*r2.get_denom());
 return r;
}
rat prod(rat const & r1, rat const & r2)
{rat r(r1.get_numer()*r2.get_numer(),
    r1.get_denom()*r2.get_denom());
 return r;
}
rat quot(rat const & r1, rat const & r2)
{rat r(r1.get_numer()*r2.get_denom(),
    r1.get_denom()*r2.get_numer());
 return r;
 }
```

Като се използват функциите за рационално числова аритметика, могат да се реализират различни приложения. Забелязваме обаче, че тази реализация не съкращава рационални числа. За преодоляването на този недостатък е достатъчно да променим конструктора с параметри. За целта реализираме разделяне на числителя x и знаменателя y на най-големия общ делител на abs(x) и abs(y). Новият конструктор има вида:

```
rat::rat(int x, int y)
{if (x == 0 || y==0) {numer = 0; denom = 1;}
 else
 {int g = gcd(abs(x), abs(y));
  if (x>0 && y>0 || x<0 && y<0)
  {numer = abs(x)/g; denom = abs(y)/g;}
  else {numer = - abs(x)/g; denom = abs(y)/g;}
```

```
}
}
```

където int gcd(int x, int y) е известната вече функция за намиране на най-големия общ делител на две естествени числа.

Ще отбележим, че ако в горната програма заменим запазената дума struct с class, програмата няма да промени смисъла си. Така дефинирахме класа rat:

```
class rat
{private:
        int numer;
        int denom;
 public:
        rat();
        rat(int, int);
        void read();
        int get_numer() const;
        int get_denom() const;
        void print() const;
};
```

а дефиницията

```
rat p, q=rat(1,7), r(-2,9);
```

определя три негови **обекта**: p, инициализиран с рационалното число 0/1; q, инициализиран с 1/7 и r, инициализиран с -2/9.

Спецификаторът private, забранява използването на член-данните numer и denom извън класа. Получава се *скриване* на информация, което се нарича още **капсолиране на информация**. Член-функциите на класа rat са обявени като public. Те са видими извън класа и могат да се използват от външни функции. Затова public-частта се нарича още **интерфейсна част на класа** или само **интерфейс**. Чрез нея класът комуникира с външната среда. Освен функции, интерфейсът може да съдържа и член-данни, но засега ще се стараем това да не се случва.

Ще отбележим, че конструкторите се използват само когато се създават обекти. Опитите за **промяна** на обект чрез обръщение към конструктор предизвикват грешки.

Пример:

```
rat q(1,7);    // коректно
q.rat();       // предизвиква грешка
```

q(2,9);   // предизвиква грешка

q.rat(3,4);   // предизвиква грешка.

Забележете, езикът C++ дефинира структурите и класовете почти идентично. Съществената разлика е свързана със спецификаторите на достъп. По подразбиране членовете на структура имат public (публичен) достъп, а членовете на клас – private (частен) достъп. Все пак възниква въпросът: *Защо да има две различни конструкции struct и class, когато разликите са толкова малки?* Причината е свързана с мобилността на програмите, с цел да се запази съвместимостта между езиците C и C++. Бярн Страуструп обяснява, че причината е по-скоро културна, отколкото техническа. Той препоръчва структурите да се използват само когато се реализират свойства, които са прости и включват малки "натоварвания". По-точно, когато структурата от данни е идентична с интерфейса си. В останалите случаи да се използват класове.

2. Дефиниране на класове

Класовете осигуряват механизми за създаване на напълно нови типове от данни, които могат да бъдат интегрирани в езика, а също за обогатяване възможностите на вече съществуващи типове. Дефинирането на един клас се състои от две части:

- декларация на класа и
- дефиниция на неговите член-функции (методи).

2.1. Декларация на клас

Декларацията на клас се състои от заглавие и тяло. Заглавието започва със запазената дума class, следвано от името на класа.

Тялото е заградено във фигурни скобки. След скобите стои знака ";" или списък от имена на обекти. В тялото на класа са декларирани членовете на класа (член-данни и член-функции) със съответните им нива на достъп. Фиг. 1 илюстрира непълно синтаксиса на декларацията на клас.

<декларация_на_клас> ::= <заглавие> <тяло>

<заглавие> ::= class [<име_на_клас>]$_{opc}$

<тяло> ::= {<декларация_на_член>;

{<декларация_на_член>;}$_{opc}$

}[<списък_от_обекти>];

<декларация_на_член> ::=

<декларация_на_конструктор>|<декларация_на_мутатор>|

<декларация_на_функция_за_достъп>|<декларация_на_данна>

<декларация_на_конструктор> ::=

[<спецификатор_на_достъп>:]$_{opc}$<име_на_клас>(<параметри>)

<декларация_на_мутатор> ::=

[<спецификатор_на_достъп>:]$_{opc}$ <тип>

<име_на_мутатор>(<параметри>)

<декларация_на_функция_за_достъп> ::=

[<спецификатор_на_достъп>:]$_{opc}$ <тип>

<име_на_функция_за_достъп>(<параметри>) const;

<спецификатор_на_достъп> ::= private | public | protected

<параметри> :: <празно> | void |

<параметър> {, <параметър>}$_{opc}$

<параметър> ::= <тип> [&|$_{opc}$ * [const]$_{opc}$] $_{opc}$

<декларация_на_данна> ::= <тип> <име_на_данна>{, <име_на_данна>}$_{opc}$

<тип> ::= <име_на_тип>|<дефиниция_на_тип>

<списък_от_обекти> ::=

<обект> [= <име_на_клас>(<фактически_параметри>)]$_{opc}$

{,<обект>[=<име_на_клас>(<фактически_параметри>)]$_{opc}$ }$_{opc}$

{, <обект>(<фактически_параметри>)}$_{oopc}$

{, <обект> = <вече_дефиниран_обект>}$_{opc}$

където <име_на_клас>, <име_на_мутатор>, <име_на_данна>, <обект> и <име_на_функция_за_достъп> са идентификатори, а <фактически_параметри> е определено в Глава 8.

Фиг. 1.

За имената на класовете важат същите правила, които се прилагат за имената на всички останали типове и променливи. Също като при структурите името на класа може да бъде пропуснато.

Имената на членовете на класа са локални за класа, т.е. в различни класове в рамките на една програма могат да се дефинират членове с еднакви имена. Член-данни от един и същ тип могат да се изредят, разделени със запетая и предшествани от типа им.

Пример:

```
class point
{private:
  double x, y;         // x и y са член-данни на класа prat
 public:
  point(double, double);
  void read();               // мутатор със същото име като на класа rat
  int get_x() const;
  int get_y() const;
  void print() const;      // със същото име като на класа rat
}p=point(2,7), q(-2,3), r=q;
```

Препоръчва се член-данните да се декларират в нарастващ ред по броя на байтовете, необходим за представянето им в паметта. Така за повечето реализации се получава оптимално изравняване до дума.

Забележка: Типът на член-данна на клас **не може** да съвпада с името на класа, но типът на член-функция на клас **може** да съвпада с името на класа.

В тялото някои декларации на членове могат да бъдат предшествани от **спецификаторите на достъп** private, public или protected. Областта на един спецификатор на достъп започва от спецификатора и продължава до следващия спецификатор. Подразбиращ се специфитатор за достъп е private. Един и същ спецификатор на достъп може да се използва повече от веднъж в декларация на клас.

Препоръчва се, ако секция public съществува, да бъде първа в декларацията, а секцията private да бъде последна в тялото на класа.

Достъпът до членовете на класовете може да се разгледа на следните две нива:

- По отношение на *член-функциите в класа* е в сила, че те имат достъп до всички членове на класа. При това не е необходимо тези компоненти да се предават като параметри. Този

режим на достъп се нарича **режим на пряк достъп**. Поради тази причина функциите rat(), read(), print(), get_numer() и get_denom() са без параметри. Освен това член-функцията print() може да бъде дефинирана и по следния начин:

```
void rat::print() const
{cout << get_numer() << "\" <<
        << get_denom() << endl;
}
```

или

```
void rat::print() const
{cout << this->get_numer() << "\" <<
        << this->get_denom() << endl;
}
```

- По отношение на *функциите, които са външни за класа*, режимът на достъп са определя от начина на деклариране на членовете.

Членовете на даден клас, декларирани като private (декларирани след запазената дума private:) са видими само в рамките на класа. Външните функции нямат достъп до тях. По подразбиране членовете на класовете са private. Това позволява декларацията на класа rat да запишем и по следния начин:

```
class rat
{int numer;
 int denom;
 public:
 rat();
 rat(int, int);
 void read();
 int get_numer() const;
 int get_denom() const;
 void print() const;
};
```

Чрез използването на членове, обявени като private, се постига скриване на членове за външната за класа среда. Процесът на скриване се нарича още **капсолиране на информацията**.

Членовете на клас, които трябва да бъдат видими извън класа (да бъдат достъпни за функции, които не са методи на дадения клас) трябва да бъдат декларирани като public

(декларирани след запазената дума public:). Всички методи на класа rat са декларирани като public и следователно могат да се използват навсякъде в програмата за работа с рационални числа.

На теория конструкторите могат да имат право на достъп private. На практика обаче, декларирането им като такива ги прави неизползваеми.

Освен като private и public, членовете на класовете могат да бъдат декларирани и като protected. Тъй като този спецификатор на достъп има отношение към производните класове и процеса на наследяване, разглеждането му засега ще бъде отложено.

2.2. Дефиниране на методите на клас

След декларирането на клас, трябва да се дефинират неговите методи. Дефинициите са аналогични на дефинициите на функции, но името на функцията се предшества от името на класа, на който принадлежи метода, следвано от оператора за принадлежност :: (Нарича се още оператор за област на действие). Такива имена се наричат **пълни**. (Операторът :: е ляво-асоциативен и с един и същ приоритет със (), [] и ->). На Фиг. 2. е даден синтаксисът на дефиницията на метод на клас.

<дефиниция_на_метод> ::=
[<тип>]$_{opc}$ <име_на_клас>::<име_на_функция>(<параметри>) [const]$_{opc}$
{<тяло>}
<тяло> ::= <редица_от_оператори_и_дефиниции>
където <име_на_клас> и <име_на_функция> са идентификатори.

Фиг. 2.

Ще отбележим, че дефиницията на конструктор **не започва** с <тип>, а запазената дума const присъства в дефиницията на функция за достъп (приемаме, че техните прототипи задължително завършват с const).

Ако се пренебрегне това правило, ще се създадат класове, които няма да могат да се използват от други програмисти.

Пример: Нека искаме да използваме класа rat, но програмистът му е забравил или нарочно не е декларирал член-функцията print() като const, т.е.

```
class rat
{private:
  ...
 public:
  ...
  void print();
};
```

и нека декларираме класа prat коректно, т.е. функциите за достъп обявяваме като const.

```
class prat
{private:
 int a;
 rat p;
 ...
 public:
 ...
 void print() const;
};
```

където

```
void prat::print() const
{cout << a << endl;
 p.print(); // тази print() е член-функцията на rat
};
```

Компилаторът ще съобщи за грешка в обръщението p.print(), защото p е обект на класа rat, а член-функцията rat::print() не е декларирана като const. Компилаторът предполага, че p.print() може да модифицира p. Но p е член-данна на prat, а prat::print() е const, с което твърдо е обещава да не го модифицира.

Обикновено дефинициите на методите са разположени веднага след декларирането на класа, на който те са членове. Възможно е обаче, дефинициите на методите на един клас да бъдат част от декларациите на този клас, т.е. в декларациите на член-функциите на класа могат да се зададат не само прототипите им, но и техните тела.

Пример: Класът rat може да бъде дефиниран и по следния начин:

```
class rat
```

```
{private:
 int numer;
 int denom;
public:
 rat()
 {numer = 0;
  denom = 1;
 }
 rat(int a, int b)
 {if(a == 0 || b==0){numer = 0; denom = 1;}
  else
  {int g = gcd(abs(a), abs(b));
   if (a>0 && b>0 || a<0 && b < 0)
 {numer = abs(a)/g;
  denom = abs(b)/g;}
   else {numer = - abs(a)/g; denom = abs(b)/g;}
  }
 }
void read()
{cout << "numer: ";
 cin >> numer;
 do
 {cout << "denom: ";
  cin >> denom;
 }while(denom==0);
}
int get_numer() const
{return numer;
}
int get_denom() const
{return denom;
}
void print() const
{cout << numer << "/" << denom << endl;
```

```
 }
};
```

В този случай обаче член-функциите се третират като **вградени функции**.

Допълнение: С цел повишаване на бързодействието, езикът С++ поддържа т.нар. вградени функции. Кодът на тези функции не се съхранява на едно място, а се копира на всяко място в паметта, където има обръщение към тях. Използват като останалите функции, но при декларирането и дефинирането им заглавието им се предшества от модификатора inline.

Пример:

```
#include <iostream.h>
inline int f(int, int);
void main()
{cout << f(1,5) << endl;
}
inline int f(int a, int b)
{return (a+b)*(a-b);
}
```

Ще добавим, че дефиницията на вградена функция трябва да се намира в същия файл, където се използва, т.е. не е възможна разделна компилация, тъй като компилаторът няма да разполага с кода за вграждане. Използването на вградени функции води до икономия на време, за сметка на паметта. Затова се препоръчва използването йи само при "кратки" функции. Ще отбележим също, че модификаторът inline е само заявка към компилатора, която може да бъде, но може и да не бъде изпълнена. Възможно е компилаторът да откаже вграждане, ако реши, че функцията е прекалено голяма или има други причини, възпрепятстващи вграждането. Ограниченията за вграждане зависят от конкретния компилатор.

Често член-функциите се реализират като вградени функции. Това увеличава ефективността на програмата. Декларацията на вградени член-функции може да се осъществи и по следния начин:

```
class rat
{private:
  int numer;
  int denom;
```

```
 public:
  rat();
  rat(int, int);
  void read();
  int get_numer() const;
  int get_denom() const;
  void print() const;
};
inline rat::rat()
{numer = 0;
 denom = 1;
}
inline rat::rat(int x, int y)
...
inline void rat::read()
...
inline int rat::get_numer() const
...
inline int rat::get_denom() const
...
inline void rat::print() const
...
```

Телата на някои от член-функциите са пропуснати, тъй като вече са описани.

Ще отбележим също, че в тялото на дефиницията на член-функция явно не се използва обектът, върху който тя ще се приложи. Той участва неявно - чрез член-данните на класа. Заради това се нарича **неявен параметър,** а член-данните – **абстрактни данни**. Връзката между неявния параметър и обект ще бъде показана в т. 3. Параметри, които участват явно в дефиницията на член-функция се наричат **явни**. *Всяка член-функция има точно един неявен параметър и нула или повече явни.*

2.3. Област на класовете

За разлика от функциите, класовете могат да се декларират на различни нива в програмата: *глобално* (ниво функция) и *локално* (вътре във функция или в тялото на клас).

Областта на глобално деклариран клас започва от декларацията и продължава до края на програмата. Примерите досега бяха с такива класове.

Ако клас е деклариран във функция, всички негови член-функции трябва да са вградени (inline). В противен случай ще се получат функции, дефинирани във функция, което не е възможно.

Пример:

```
void f(int i, int* p)
{int k;
 class a
 {public:
      // всички методи са дефинирани в тялото на класа
  ...
  private:
  ...
 };
// тяло на функцията f
 a x;
 ...
}
```

Областта на клас, дефиниран във функция е функцията. Обектите на такъв клас са видими само в тялото на функцията. Възможно е използването на обекти (в широкия смисъл на думата) с еднакви имена. Важи правилото, че локалния обект скрива нелокалния.

Не е възможно в тялото на локално дефиниран клас да се използва функцията, в която класът е дефиниран.

Пример:

```
void f(...)
{...
 class cl
 {
      // не може да се използва функцията f
 };
 ...
}
```

3. Обекти

След като даден клас е дефиниран, могат да бъдат създавани негови екземпляри, които се наричат **обекти**. Връзката между клас и обект в езика C++ е подобна на връзката между тип данни и променлива, но за разлика от обикновените променливи, обектите се състоят от множество **компоненти** (член-данни и член-функции). На Фиг. 3 е даден синтаксисът на дефиниция на обект.

<дефиниция-на_обект_на_клас> ::=
<име_на_клас> <обект> [=<име_на_клас>(<фактически_параметри>)]$_{opc}$
{,<обект>[=<име_на_клас>(<фактически_параметри>)]$_{opc}$ }$_{opc}$
{, <обект>(<фактически_параметри>)}$_{opc}$
{, <обект> = <вече_дефиниран_обект>}$_{opc}$;
<обект> ::= <идентификатор>
където <фактически_параметри> е определено в Глава 8.

Фиг. 3.

Когато за даден клас явно са дефинирани конструктори, при всяко дефиниране на обект на класа те автоматично се извикват с цел да се инициализира обекта. Ако дефиницията е без явна инициализация (например rat p;), дефинираният обект се инициализира според дефиницията на конструктора по подразбиране, ако такъв е определен, и се съобщава за грешка в противен случай. Ако дефиницията е с явна инициализация, обръщението към конструкторите трябва да бъде коректно.

Пример: Дефиницията

```
rat p, q(2,3), r=rat(3,8);
```

определя три обекта: p, инициализиран с рационалното число 0/1, q, инициализиран с 2/3 и r, инициализиран с 3/8. Тя е добре оределена, тъй като класът rat има конструктор по подразбиране и двуаргументен конструктор. Ако елиминираме конструктора по подразбиране в класа rat, горната дефиниция ще съобщи за грешка заради p. Валидна е обаче дефиницията:

```
rat q(2,3), r=rat(3,8);
```

Ако се откажем и от другия конструктор, последната дефиниция също ще стане невалидна.

Когато за даден клас явно не е дефиниран конструктор, реализацията автоматично създава такъв. Последният се нарича **подразбиращ се конструктор**. Подразбиращият се конструктор изпълнява редица действия, като заделяне на памет за обектите, инициализиране на някой системни променливи и др. В този случай дефиницията на обект от този клас трябва да е без явна инициализция.

Пример:

```
#include <iostream.h>
class pom
{private:
  int a;
 public:
  int b;
  void read();
  void print() const;
};
void main()
{pom x;  // инициализация според подразбиращия се
                    //конструктор на C++
 x.read();
 x.print();
}
void pom::print()const
{cout << "a= "<< a << " b=" << b << endl;
}
void pom::read()
{cout << "a= ";
 cin >> a;
 cout << "b= ";
 cin >> b;
}
```

В случая обектът x се инициализира с неопределена стойност. Опитите за инициализацията му като структура предизвикват грешки.

Дефиницията на клас не заделя памет за него. Памет се заделя едва при дефинирането на обект от класа. Дефиницията от по-горе заделя за обектите p, q и r по 8 байта ОП (по 4B за всяка от данните им numer и denom).

Достъпът до компонентите на обектите (ако е възможен) се осъществява чрез задаване на името на обекта и името на данната или метода, разделени с точка (Фиг. 4). Изключение от това правило правят конструкторите (Фиг. 3).

```
<компонента_на_обект> ::=  <обект>.<данна>|
                <обект>.<име_на_член_функция>()|
                                <обект>.<име_на_член_функция>(<параметри>)
```

<име_на_член_функция> е <идентификатор>, означаващ име на мутатор или име на функция за достъп.

Фиг. 4.

Пример:

```
rat p(1,2), q;
p.get_numer()     // достъп до член-данната get_numer() за обекта p
q.get_numer()     // достъп до член-данната get_numer() за обекта q.
```

Ще отбележим също, че на практика обектите p и q нямат свои копия на метода get_numer(). И двете обръщения се отнасят за един и същ метод, но при първото обръщение ще се работи с данните за обекта p, а при второто – с данните на обекта q.

При създаването на обекти на един клас кодът на методите на този клас не се копира във всеки обект, а се намира само на едно място в паметта. Естествено възниква въпросът *по какъв начин методите на един клас "разбират" за кой обект на този клас са били извикани*. Отговорът на този въпрос дава указателят this. Всяка член-функция на клас поддържа допълнителен формален параметър - указател с име this и от тип <име_на_клас>*. За да разберем точно как става това, ще разгледаме работата на компилатора на C++. Тя се извършва на две стъпки:

а) Всяка член-функция на даден клас се транслира в обикновена функция с уникално име и един допълнителен параметър – указателят this.

Пример: Фукцията

```
void rat::print()
{cout << numer << "/" << denom << endl;
}
```

се транслира в

```
void print_rat(rat* this)
{cout << this->numer << "/" <<this->denom << endl;
}
```

б) Всяко обръщение към член-функция се транслира в съответствие с преобразуванието от а).

Пример: Обръщението

```
p.print();
```

се транслира в

```
print_rat(&p);
```

Указателят this може да се използва явно в кода на съответната член-функция, макар че е глупаво да се напише:

```
print_rat()
{cout << this->numer << "/" <<this->denom << endl;
}
```

Като приложение на указателя this функцията

```
rat sum(rat const &, rat const &);
```

ще направим член-функция на класа rat. За целта ще включим псевдонима й

```
void sum(rat const &, rat const &);
```

в public-секцията на тялото на rat и ще я дефинираме като член-функция по следния начин:

```
void rat::sum(rat const & r1, rat const & r2)
{numer = r1.numer*r2.denom+r2.numer*r1.denom;
 denom = r1.denom*r2.denom;
 return *this;
}
```

Нека

```
rat p=rat(1,4), r(1,2), q=rat(1,4);
```

Фрагментът

```
r.sum(p.sum(p, r), q);
r.print();
p.print();
```

съобщава за стойност на p 6/8, а за стойност на r – 32/32. Обръщението p.sum(p, r) намира сумата на рационалните числа p и r и я свързва с обекта p, а r.sum(p.sum(p, r), q) събира полученото рационално число с r и свързва резултата с обекта r.

Обекти от един и същ клас могат да се присвояват един на друг. Присвояването може да е и на ниво инициализация (фиг. 3).

Пример: Допустими са дефинициите

```
rat p, q(4,5), r=q;
p = q;
...
r = p;
```

При присвояването се копират всички член-данни на обекта. Така присвояването

```
r = p;
```

е еквивалентно на

```
r.numer = p.numer;
r.denom = p.denom;
```

Подробности относно процеса на присвояване са дадени в следващата точка.

4. Конструктори

Създаването на обекти е свързано със заделяне на памет, запомняне на текущо състояние, задаване на начални стойности и др. дейности, които се наричат **инициализация на обекта**. В езика C++ тези дейности се изпълняват от специален вид член-функции на класовете – конструкторите.

4.1. Дефиниране на конструктор

На Фиг. 5. е дадена най-често използваната форма за дефиниране на конструктор.

```
<дефиниция_на_конструктор> ::=
<име_на_клас>::<име_на_клас>(<параметри>)
{<тяло>}
<тяло> ::= <редица_от_оператори_и_дефиниции>
```

Фиг. 5.

Ще напомним, че конструкторът е член-функция, която притежава повечето характеристики на другите член-функции, но има и редица особености, като:

- Името на конструктора съвпада с името на класа.
- Типът на резултата е указателят this и явно не се указва.
- Изпълнява се автоматично при създаване на обекти.
- Не може да се извиква явно (обръщение от вида r.rat(1,4) е недопустимо).

Освен това в един клас може явно да не е дефиниран конструктор, но може да са дефинирани и няколко конструктора.

Типична дефиниция на конструктор е дадена в следващия пример.

Пример:

```
class CL
{public:
  CL(int, int, int);
  void print();
  ...
 private:
  int a, b, c;
};
CL::CL(int x, int y, int z) // конструктор с три параметъра
{a = x;
 b = y;
 c = z;
}
```

Класът CL в този пример има един триаргументен конструктор. При създаване на обект на класа, този конструктор ще се изпълнява автоматично, в резултат на което член-данните a, b и c на създадения обект ще се свържат със стойностите, които се подадат като фактически параметри на конструктора.

На Фиг. 6. е дадена по-обща дефиниция на конструктор.

<дефиниция_на_конструктор> ::=

<име_на_клас>::<име_на_клас>(<параметри>):

<член_данна>(<израз>){,<член_данна>(<израз>)}$_{\text{opc}}$

{<тяло>}

<тяло> ::= <редица_от_оператори_и_дефиниции>

Фиг. 6.

Забелязваме, че в заглавието на конструктора е възможно да се свърже член_данна с инициализираща стойност.

Пример: Конструкторът на класа CL може да се дефинира и по следния начин

```
CL::CL(int x, int y, int z): a(x), b(y), c(z)
{}
```

Ще отбележим, че не е задължително всички член-данни да са инициализирани само пред тялото или само вътре в тялото на конструктора.

Пример: Дефиницията

```
CL::CL(int x, int y, int z): a(x)
{b = y;
 c = z;
}
```

също е допустима.

Смисълът на обобщената синтактична конструкция е, че инициализацията на член-данните в заглавието предшества изпълнението на тялото на конструктора. Това я прави изключително полезна. Използването й увеличава ефективността на програмата поради следните съображения.

Когато член-данни на клас са обекти, в дефиницията на конструктора на класа се използват конструкторите на класовете, от които са обектите (член-данни). Преди да започне изпълнението на конструктора, автоматично се извикват конструкторите по подразбиране на всички член-данни, които са обекти. Веднага след това тези член-данни се инициализират с обектите, резултат от изпълнението на извикания конструктор. Това двойно извикване на конструктори намалява ефективността на програмата.

Пример: Нека дефинираме класа prat, като член-данна в него е обект на класа rat.

```
class prat
{public:
  prat(int, int, int);
  ...
 private:
  int a;
  rat r;
};
```

където

```
prat::prat(int x, int y, int z)
{a = x;
 r = rat(y, z);
}
```

и сме дефинирали обекта

```
prat q=prat(1,2,3);
```

Преди да започне изпълнението на конструктора prat, автоматично се извиква конструкторът по подразбиране на rat и член-данната r на prat се инициализира с 0/1. Веднага след това r се свързва с обекта rat(y,z) за текущите y и z. По-ефективно е данната r да се свърже с правилната стойност направо, без междинна инициализация. Това може да се реализира чрез дефиницията:

```
prat::prat(int x, int y, int z): r(rat(y, z))
{a = x;
}
```

или съкратено

```
prat::prat(int x, int y, int z): r(y, z)
{a = x;
}
```

4.2. Предефинирани конструктори

Обект (в общия смисъл) е предефиниран, ако за него има няколко различни дефиниции, задаващи различни негови интерпретации. За да бъдат използвани такива конструкции е необходим критерии, по който те да се различат.

В рамките на една програма може да се извършва предефиниране на функции. Възможно е:

а) да се използват функции с едно и също име с различни области на видимост

В този случай не възниква проблем с различаването.

б) да се използват функции с едно и също име в една и съща област на видимост

В този случай компилаторът търси функцията с възможно най-добро съвпадане. Като критерии за добро съвпадане са въведени следните нива на съответствие:

- точно съответствие (по брой и тип на формалните и фактическите параметри)
- съответствие чрез разширяване на типа.

  Извършва се разширяване по веригата

  char -> short -> int -> longint

  float -> double
- други съответствия (правила въведени от потребителя).

В един клас може да са дефинирани няколко конструктора. Всички те имат едно име (името на класа), но трябва да се различават по броя и/или типа на параметрите си. Наричат се **предефинирани конструктори**. При създаването на обект на класа се изпълнява само един от тях. ОПределя се съгласно критерия за най-добро съвпадане.

Пример: В класа rat дефинирахме два конструктора rat() и rat(int, int), които се различават по броя на параметрите си.

4.3. Подразбиращ се конструктор

В клас може явно да е дефиниран, но може и да не е дефиниран конструктор. Ако явно не е дефиниран конструктор, автоматично се създава един т. нар. **подразбиращ се конструктор**. Този конструктор реализира множество от действия като: заделяне на памет за данните на обект, инициализиране на някои системни променливи и др.

Подразбиращият се конструктор може да бъде предефиниран. За целта е необходимо в класа да бъде дефиниран конструктор без параметри.

Пример: В класа rat, подразбиращият се конструктор беше предефиниран от конструктора

```
rat::rat()
{numer = 0;
 denom = 1;
}
```

4.4. Конструктори с подразбиращи се параметри

Функциите в езика C++ могат да имат подразбиращи се параметри. За тези параметри се задават подразбиращи се стойности, които се използват само ако при извикването на функцията не бъде зададена стойност за съответния параметър.

Задаването на подразбираща се стойност се извършва чрез задаване на конкретна стойност в прототипа на функцията, а не в нейната дефиниция.

Пример: Да разгледаме програмата

```
#include <iostream.h>
void f(double, int=10, char* ="example1"); //интервал между * и =
int main()
{double x = 1.5;
 int y = 5;
 char z[] = "ezample 2";
 f(x, y, z);
 f(x, y);
 f(x);
 return 0;
}
void f(double x, int y, char* z)
{cout << "x= " << x << " y= " << y
   << " z= " << z << endl;
}
```

В резултат от изпълнението й се получава:

```
x = 1.5 y = 5 example 2
x = 1.5 y = 5 example 1
x = 1.5. y = 10 example 1
```

В тази програма е дефинирана функцията f с три формални параметъра. От прототипа й се вижда, че два от тях (вторият и третият) са подразбиращи се. Тъй като в обръщенията към f

f(x, y); и

f(x);

не са указани три фактически параметър, за стойности на липсващите параметри се вземат указаните стойности от прототипа на функцията.

При използване на подразбиращи се параметри, важна роля играе редът на параметрите. Прието е, че ако параметър на функция е подразбиращ се, всички параметри след него също са подразбиращи се.

Пример: Прототип на функция

void f(double = 1.5, int, char* “example 1”);

предизвиква грешка, тъй като първият формален параметър е обявен за подразбиращ се, а вторият не е такъв.

Ще отбележим също, че ако за даден подразбиращ се параметър е зададена стойност при обръщението към функцията, за всички параметри *пред него* също трябва да са указани такива.

Всичко казано за подразбиращите се параметри на функции се отнася и за конструкторите.

Ако променим дефиницията на rat по следния начин

```
class rat
{private:
  int numer;
  int denom;
 public:
  rat(int=0, int=1);
  void read();
  int get_numer() const;
  int get_denom() const;
  void print() const;
};
```

където конструкторът rat(int, int); се дефинира по същия начин, са допустими следните дефиниции на обекти:

```
rat p,                      // p се инициализира с 0/1
    q = rat(),              // q се инициализира с 0/1
    r = rat(5),             // r се инициализира с 5/1
    s = rat(13,21);         // s се инициализира с 13/21
    t(2)                    // t се инициализира с 2/1
```

4.5. Конструктори за присвояване и копиране

Инициализацията на новосъздаден обект на даден клас може да зависи от друг обект на същия клас. За да се укаже такава зависимост се използва знакът за присвояване или кръгли скоби.

Пример: Нека

rat p = rat(1,4);

Чрез еквивалентните конструкции

```
rat q = p;
rat q(p);
```

се създава обектът q от клас rat, като инициализацията на q зависи от p. Тази инициализация се създава от специален конструктор, наречен **конструктор за присвояване**.

Конструкторът за присвояване е конструктор, поддържащ формален параметър от тип <име_на_клас> const &.

- Ако в един клас явно не е дефиниран конструктор за присвояване, компилаторът автоматично създава такъв, в момента когато новосъздаден обект се инициализира с обекта, намиращ се от дясната страна на знака за присвояване или в кръглите скоби. Този конструктор за присвояване се нарича конструктор за копиране.

Пример: В класа rat не беше дефиниран конструктор за присвояване. Затова при създаване на обект p чрез дефиницията rat p = q;
автоматично се извиква конструкторът за копиране. Последният има вида:

```
rat::rat(rat const & r)
{numer = r.numer;
 denom = r.denom;
}
```

Тъй като няма конструктор за присвояване, при дефинирането на обекта p се създава нов обект *(без викане на конструктор)*, в който се копират съответните стойности на обекта q.

Освен в горния случай, служебният конструктор за копиране се използва и при предаване на обект като аргумент на функция, а също при връщане на обект като резултат от изпълнение на функция (Изключение правят параметрите псевдоними).

- Ако в класа е дефиниран конструктор за присвояване, компилаторът го използва.

Пример:

```
class rat
```

```
{private:
  int numer;
  int denom;
 public:
  rat(rat const &); // конструктор за присвояване
  rat();
  rat(int=0, int=1);
  void read();
  int get_numer() const;
  int get_denom() const;
  void print() const;
};
...
rat::rat(rat const & r)
{numer = r.numer + 1;
 denom = r.denom + 1;
}
rat p,          // p се инициализира с 0/1
    q = p,      // q се инициализира с 1/2
        r = q        // r се инициализира с 2/3
        s = r,       // s се инициализира с 3/4
         t(s);       // t се инициализира с 4/5.
```

5. Указатели към обекти

Дефинират се по същия начин както указател към основен тип данни.

Пример:

```
rat p;
rat * ptr = &p;
```

В резултат ще се отделят 4B ОП, които ще се именуват с ptr и ще се инициализират с адреса на обекта p.

Достъпът до компонентите на рационалното число, сочено от ptr, се осъществява по стандартния начин:

```
(*ptr).get_numer()
```

(*ptr).get_denom()

Синтактичната конструкция (*ptr). е еквивалентна на ptr ->. Така горните обръщения могат да се запишат и по следния начин:

ptr -> get_numer()

ptr -> get_denom()

Ще напомним, че this е указател от тип <име_на_клас>*.

6. Масиви и обекти

Елементите на масив могат да са обекти, но разбира се от един и същ клас. Дефинират се по общоприетия начин (Фиг. 7).

```
   <дефиниция_на_променлива_от_тип_масив> ::=
      T <променлива>[size]; |
      T <променлива>[size] = {<инициализиращ_списък>};
където
   T е име на клас;
   <променлива> ::= <идентификатор>
   size е константен израз от интегрален или изброен тип със
положителна стойност;
   <инициализиращ_списък> ::= <стойност>{, <стойност>}opc
         {, <име_на_конструктор>(<фактически_параметри>)}opc
         {, <обект_от_тип_T>(<фактически_параметри>)}
```

Фиг. 7.

Пример:

rat table[10];

определя масив от 10 обекта от клас rat.

Достъпът до елементите на масива е пряк и се осъществява по стандартния начин:

Пример: Чрез индексираните променливи

table[0], table[1], ..., table[9]

се осъществява достъп до първия, втория и т.н. до десетия елемент на table.

Тъй като table[i] (i = 0, 1, ..., 9) са обекти, възможни са следните обръщения:

```
table[i].read();        // въвежда стойност на table[i]
table[i].print();       // извежда стойността на table[i]
table[i].get_numer();   // намира числителя на table[i]
table[i].get_denom();   // намира знаменателя на table[i].
```

Връзката между масиви и указатели е в сила и в случая когато елементите на масива са обекти. Името на масива е указател към първия му елемент, т.е. ако

```
rat * p = table;     // p сочи към table[0]
                     // т.е. p ==&table[0]
*(p+i) == table[i], i = 0,1, ...,9
```

Тогава

```
(*(p+i)).print();    // е еквивалентно на table[i].print();
```

Масивът може да е член на клас.

Пример: Допустима е конструкцията

```
class exp
{int a;
 int table[10];
 public:
 int array[10];
}x[5];
```

Достъпът до компонентите на масива array ще се осъществи по следния начин:

```
x[i].array[j], i = 0, 1, ..., 4; j = 0, 1, ..., 9.
```

Конструкторът по подразбиране играе важна роля при дефинирането и инициализирането на масиви от обекти. аАсив от обекти, дефиниран в програмата, се инициализира по два начина:

- *явно* (чрез инициализиращ списък)
- *неявно* (чрез извикване на конструктора по подразбиране за всеки обект – елемент на масива).

Примери:

а) Класът

```
const NUM = 5;
class student
{private:
 int facnom;
 char name[26];
```

```
 double marks[NUM];
 public:
  void read_student();
  void print_student() const;
  bool is_better(student const &) const;
  double average() const;
};
```

няма явно дефиниран конструктор. Дефиницията

```
student table[30];
```

на масива table от 30 обекта от клас student е правилна. Инициализацията се осъществява чрез извикване на конструктора по подразбиране за всеки обект – елемент на масива.

б) Класът rat, дефиниран по-долу

```
class rat
{private:
   int numer;
   int denom;
  public:
   rat(rat const & r);
   rat(int=0, int=1);
   void read();
   int get_numer() const;
   int get_denom() const;
   void print() const;
};
```

притежава явно дефиниран конструктор с подразбиращи се параметри. В този случай са допустими дефиниции от вида:

```
rat x[10]; // x[i] се инициализира с 0/1, за всяко i=0,1,...9.
rat x[10] = {1,2,3,4,5,6,7,8,9,10}; //x[i] == i/1
rat x[10] = {rat(1,2),rat(2),rat(3, 5),4,5,6,7,8,9,10};
// x[0] == ½; x[1] == 2/1; x[2] == 3/5, x[3]== 3/1 ...
```

Задача 117. Да се напише програма, която въвежда следната информация за компютри: име на модела (name), цена (price) и точки (score) между 1 и 100. Да се изведе въведената информация, след което да се сортира в низходящ ред по съотношение точки/цена и изведе.

Program Zad117.cpp

```
#include <iostream.h>
#include <iomanip.h>
#include <string.h>
class product
{private:
  char name[21];
  double price;
  int score;
 public:
  product();
  void read();
  void print() const;
  bool is_better_from(product const &) const;
  double get_price() const;
  int get_score() const;
};
void sorttable(int n, product* []);
int main()
{cout << setprecision(4) << setiosflags(ios::fixed);
 product table[300];
 product* ptable[300];
 int n;
 do
 {cout << "number of products? ";
  cin >> n;
 }while (n<1 || n>300);
 int i;
 for (i=0; i<=n-1; i++)
 {table[i].read();
  ptable[i]=&table[i];
 }
 cout << "table: \n";
 for (i = 0; i <= n-1; i++)
```

```
 {table[i].print();
  cout << endl;
 }
 sorttable(n, ptable);
 cout << "\n New Table: \n";
 for (i = 0; i <= n-1; i++)
 {ptable[i]->print();
     cout << setw(7)
   << ptable[i]->get_score()/ptable[i]->get_price()
   << endl;
 }
 return 0;
}
void product::read()
{cout << "name: ";
 cin >> name;
 cout << "price: ";
 cin >> price;
 cout << "score: ";
 cin >> score;
}
 void product::print() const
{cout << setw(25) << name
    << setw(10) << price
    << setw(12) << score;
}
bool product::is_better_from(product const & x) const
{return score/price > x.score/x.price;
}
double product::get_price() const
{return price;
}
int product::get_score() const
{return score;
```

```
}
void sorttable(int n, product* a[])
{for (int i = 0; i <= n-2; i++)
 {int k = i;
  product* max = a[i];
  for (int j = i+1; j <= n-1; j++)
  if ((*a[j]).is_better_from((*max)))
  {max = a[j];
        k = j;
    }
   max = a[i]; a[i] = a[k]; a[k] = max;
 }
 }
```

Глава 15

Стек. Опашка. Свързан списък.
Приложения

1. Стек

1.1. Реализация на стек

Предложеното представяне на стека в глава 14 не е най-доброто заради използването на структура за представяне на двойката

inf	link

В следващото представяне е дефиниран класът stack, който използва помощния и малко странен клас item за реализиране на двойната кутия.

```
class item
{friend class stack;
 private:
  item(int x=0)   // конструктор
  {inf = x;
   link = 0;
  }
  int inf;
  item* link;
};
```

Тъй като item използва класа stack в декларацията си, прототипът на класа stack трябва да предшества декларацията на item. Странността на item произтича от това, че всичките му членове са капсолирани. Чрез декларацията

```
friend class stack;
```

item позволява само клас stack да създава и обработва негови обекти. Конструкторът на item има един подразбиращ се аргумент, което позволява да бъде използван като и конструктор по подразбиране.

Класът stack има вида:

```
class stack
{public:
  stack();
  stack(int x);
  ~stack();
  stack(stack const &);
  stack& operator=(stack const &);
  int push(int const&);
  int pop(int & x);
  int top() const;
  bool empty() const;
  void print();
 private:
  item *start;
  void delstack();
```

```
    void copy(stack const&);
  };
```

Предложени са два конструктора:

```
  stack::stack()
  {start = NULL;
  }
```

и

```
  stack::stack(int x)
  {start = new item(x);
  }
```

Необходими са за да могат да бъдат създавани празен стек и стек с един (указан като аргумент) елемент.

Тъй като обектите на класа stack са реализирани в динамичната памет, за него трябва да реализираме каноничното представяне – деструктор, конструктор за присвояване и операторна функция за присвояване. Тези член-функции са аналогични на съответните от реализацията на стека от предишното представяне.

деструктор

```
  stack::~stack()
  {delstack();
  }
```

конструктор за присвояване

```
  stack::stack(stack const & r)
  {copy(r);
  }
```

операторна функция за присвояване

```
  stack& stack::operator=(stack const& r)
  {if(this != &r)
  {delstack();
   copy(r);
  }
  return *this;
  }
```

където член-функциите delstack() и copy(stack const&) също са аналогични на тези от предното представяне.

Класът stack реализира стек от цели числа. В следващото приложение ще имаме нужда от два класа стек: клас стек от цели числа и клас стек от символи. Затова, като използваме проектирания

клас stack, ще дефинираме шаблон на клас stack за горното представяне.

```
template <class T>
class stack;
template <class T>
class item
{friend class stack<T>;
 private:
  item(T x = 0)
  {inf = x;
   link = 0;
  }
  T inf;
  item* link;
};
template <class T>
class stack
{public:
  stack(T x);
  stack();
  ~stack();
  stack(stack const &);
  stack& operator=(stack const &);
  void push(T const&);
  int pop(T & x);
  T top() const;
  bool empty() const;
  void print();
 private:
  item<T> *start;
  void delstack();
  void copy(stack const&);
};
template <class T>
stack<T>::stack(T x)
{start = new item<T>(x);
}
template <class T>
stack<T>::stack()
```

```
{start = NULL;
}
template <class T>
stack<T>::~stack()
{delstack();
}
template <class T>
stack<T>::stack(stack<T> const & r)
{copy(r);
}
template <class T>
stack<T>& stack<T>::operator=(stack<T> const& r)
{if(this != &r)
{delstack();
 copy(r);
}
return *this;
}
template <class T>
void stack<T>::delstack()
{item<T> *p;
 while(start)
 {p = start;
  start = start->link;
  delete p;
 }
}
template <class T>
void stack<T>::copy(stack<T> const & r)
{if(r.start)
{item<T> *p = r.start, *q = NULL, *s=NULL;
 start = new item<T>;
 start->inf = p->inf;
 start->link = q;
 q = start;
 p = p->link;
 while(p)
 {s = new item<T>;
  s->inf = p->inf;
```

```
  q->link = s;
  q = s;
  p = p->link;
 }
 q->link = NULL;
}
else start = NULL;
}
template <class T>
void stack<T>::push(T const& x)
{item<T> *p = new item<T>(x);
 p->link = start;
 start = p;
}
template <class T>
int stack<T>::pop(T & x)
{item<T> *temp, *p = start;
 if(p)
 {x = p->inf;
  temp = p;
  p = p->link;
  delete temp;
  start = p;
  return 1;
 }else return 0;
}
template <class T>
T stack<T>::top() const
{return (*start).inf;
}
template <class T>
void stack<T>::print()
{T x;
 while(pop(x))
   cout << x << " ";
 cout << endl;
}
template <class T>
bool stack<T>::empty() const
```

```
{return start==NULL;
}
```

Ще използваме тази дефиниция на шаблона stack за да покажем как става пресмятането на стойността на аритметичен израз чрез преобразуването му в обратен полски запис.

1.2. Пресмятане на стойност на аритметичен израз

Ще разглеждаме аритметични изрази от вида:

```
<израз> ::= <терм>|
            <израз>+<терм>|
            <израз>-<терм>
<терм> ::= <множител>|
            <терм>*<множител>|
            <терм>/<множител>
<множител> ::= <цифра>|<променлива>|(<израз>)|
              <множител> ^ <цифра>
<цифра> ::= 0|1| ... |9
<променлива> :: <буква>.
```

където ^ означава операцията степенуване.

Пресмятането на стойността на аритметичен израз се реализира по-просто, ако изразът е записан не в обичайния му инфиксен вид, а в т. нар. **обратен полски запис.** Характерно за този вид запис на израз е, че операцията се записва след аргументите си.

Примери:

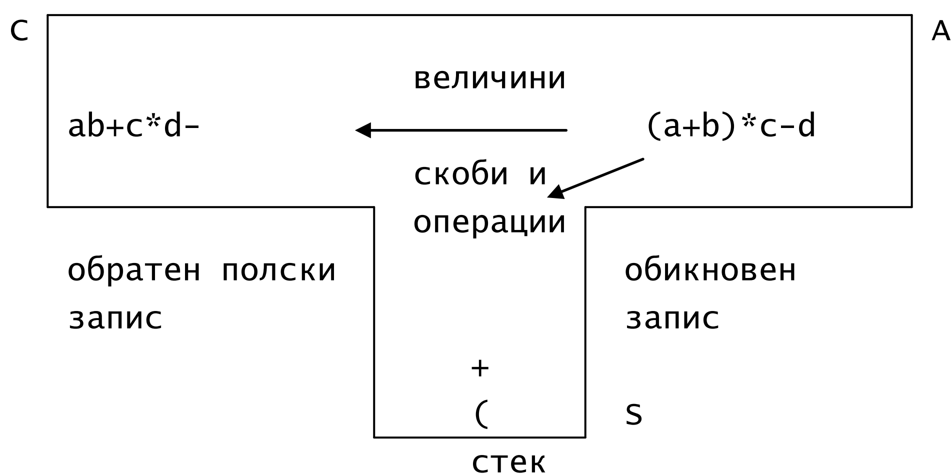
обикновен запис	*обратен полски запис*
(a-b/c)*m	abc/-m*
a^n*b^m	an^bm^*
(a-b)*(a+b)	ab-ab+*
(a+b)*c-d*f+m^p	ab+c*df*-mp^+

Ако аритметичен израз е представен чрез обратен полски запис, пресмятането на стойността му се осъществява по следния начин. Сканира се изразът, представен чрез обратен полски запис отляво надясно до намиране на знак за операция. Пресмята се стойността на терма с аргументи – първите два, намиращи се непосредствено пред операцията. Операцията и операндите се заменят с резултата от пресмятането, след което продължава търсенето на знак за операция.

Сканирането продължава до достигане края на израза. За целта е удобно да се използва стек.

Преобразуване на израз от обикновен в обратен полски запис

Ще използваме стек. Фиг. 1. илюстрира преобразуването.



Фиг. 1.

Стекът е означен със S. Символите, участващи в обикновения запис се прехвърлят от частта A в частта C, като някои от тях временно престояват в стека S. В частта C се получава изразът, записан в обратен полски запис.

Правилата за движение са следните:

- Величините (цифри и променливи) се преместват директно от частта A в частта C.
- Скобата '(' се включва в стека S.
- Операциите +, -, *, / и ^ се включват в стека S. Всяка операция има определен приоритет. Реализира се с цяло число, което се нарича тегло. Приемаме, че най-тежки са + и -, по-леки от тях са * и / и най-лека е операцията за степенуване. Ако при включване на една операция в стека S, под нея има по-лека или с равно тегло операция, по-леката или с равно тегло операция се премества от S в частта C. Това се повтаря докато се достигне до по-тежка операция, до ')' или до изпразването на стека.
- Скобата ')' изключва от стека S всички операции до достигане до '('. Операциите се записват в частта C в реда на изключването им, а скобата '(' се унищожава от ')'.

- Ако всички символи от частта A са обработени, елементите на стека S, до изпразването му или до достигане до ‘(‘, се прехвърлят в областта C.

Задача 122. Да се напишат функции, реализиращи преобразуване на аритметичен израз от вида, описан по-горе, в обратен полски запис, а също за намиране стойността на израз, представен в обратен полски запис.

Процедурата

```
void translate(char *s, char *ns);
```

превежда аритметичния израз, представен в обикновен запис чрез низа s, в обратен полски запис – низа ns. За да се избегне проверката за празен стек, в помощния стек още в началото включваме ‘(‘.

```
void translate(char *s, char *ns)
{stack<char> st;
 st.push('(');
 char x;
 int i = -1, j = -1, n = strlen(s);
 while(i < n)
 {i++;
  if (s[i] >= '0' && s[i] <= '9')
  {j++; ns[j] = s[i];}
  else if(s[i]=='(') st.push(s[i]);
  else
  if (s[i]==')')
  {st.pop(x);
   while(x!='(')
   {j++; ns[j] = x;
    st.pop(x);
    }
   }
   else
   if(s[i]=='+'||s[i]=='-'||
      s[i]=='*'||s[i]=='/'||s[i]=='^')
    {st.pop(x);
     while(x!='(' && t(x) <= t(s[i]))
     {j++;
```

```
         ns[j] = x;
         st.pop(x);
        }
        st.push(x);
        st.push(s[i]);
       }
      }
      st.pop(x);
      while(x!='(')
      {j++; ns[j] = x;
       st.pop(x);
      }
      j++;
      ns[j] = 0;
    }
```

Функцията t намира теглото на операциите, използвани в аритметичния израз, определен по-горе и има вида:

```
   int t(char c)
   {int p;
    switch(c)
    {case '+': p = 2; break;
     case '-': p = 2; break;
     case '*': p = 1; break;
     case '/': p = 1; break;
     case '^': p = 0; break;
     default:  p = -1;
    }
    return p;
   }
```

Функцията

```
   int value(char *s);
```

намира стойността на израза, като използва обратния полски запис, който му съответства.

```
   int value(char *s)
   {stack<int> st;   // генерира стек st от цели числа
    int x, y, z;
    unsigned int i = 0, n = strlen(s);
    while (i < n)
    {if(s[i] >= '0' && s[i] <= '9') st.push((int)s[i]-(int)'0');
```

```
else
 if(s[i]=='+'|| s[i]=='-' || s[i]=='*'
             || s[i]=='/' || s[i]=='^')
 {st.pop(y);
  st.pop(x);
  switch (s[i])
  {case '+': z = x+y; break;
   case '-': z = x-y; break;
   case '*': z = x*y; break;
   case '/': z = x/y; break;
   case '^': z = (int)pow(x,y);
  }
 st.push(z);
 }
 i++;
 }
 st.pop(z);
 return z;
}
```

Тогава намирането на стойността на аритметичен израз може да се реализира чрез изпълнението на следната главна функция:

```
void main()
{char s[200];
 cout << "s: ";
 cin >> s;
 char s1[200];
 translate(s, s1);
 cout << value(s1);
 cout << endl;
}
```

Задача 123. В масив е записан без грешка булев израз от вида:

```
<булев_израз> ::= t | f | (~<булев_израз>)|
                  (<булев_израз>*<булев_израз>)|
                  (<булев_израз>+<булев_израз>)
```

където t означава истина, f – лъжа, а ~, * и + означават съответно логическо отрицание, конюнкция и дизюнкция. Да се напише програма, която намира стойността на правилно записан в масив булев израз.

Функцията BulFormula решава задачата. Тя използва два помощни стека s1 и s2. Ако сканираният символ е знак за операция, се включва в стека s1, ako e t или f, се включва в стека s2. Затварящата скоба изключва операция от стека s1, анализира я и в зависимост от вида ѝ – унарна или бинарна изключва един или два елемента от s2. Изпълнява се операцията над съответните аргументи и резултатът се включва в стека s2. Останалите символи се пропускат.

Program Zad123.cpp

```
#include <iostream.h>
#include <string.h>
#include "stack-link"
bool BulFormula(char* s)
{stack<char> s1, s2;
 char c, x, y;
 int i = -1, n = strlen(s);
 while(i<n)
 {i++;
  if(s[i]=='~'||s[i]=='*'||s[i]=='+') s1.push(s[i]);
  else if(s[i]=='t'||s[i]=='f')s2.push(s[i]);
  else
  if(s[i]==')')
  {s1.pop(c);
   switch(c)
   {case '~': s2.pop(x);
              if(x=='t') c='f'; else c='t';
              s2.push(c); break;
   case '*': s2.pop(y); s2.pop(x);
             if(x=='t'&&y=='t') c='t'; else c='f';
             s2.push(c); break;
   case '+': s2.pop(y); s2.pop(x);
             if(x=='f'&&y=='f') c='f'; else c='t';
             s2.push(c); break;
   }
  }
 }
  s2.pop(c);
  return c=='t';
 }
void main()
```

```
{char s[200];
 cout << "s: ";
 cin >> s;
 cout << BulFormula(s) << endl;
 cout << endl;
}
```

2. Опашка

Логическо описание

Опашката е крайна редица от елементи от един и същ тип. Операцията включване е допустима за елементите от единия край на редицита, който се нарича **край на опашката**, а операцията изключване на елемент - само за елементите от другия край на редицата, който се нарича **начало на опашката**. Възможен е достъп само до елемента, намиращ се в началото на опашката, при това достъпът е пряк.

При тази организация на логическите операции, първият включен елемент се изключва първи. Затова опашката се определя още като структура от данни "*пръв влязъл – пръв излязъл*".

Физическо представяне

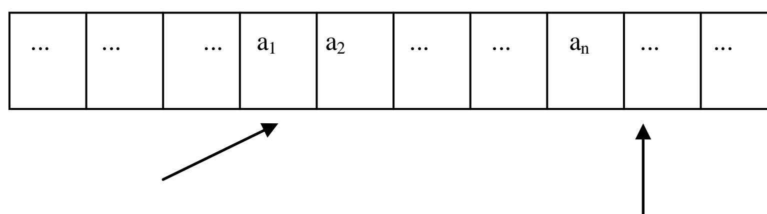
Широко се използват два основни начина за физическо представяне на опашка: *последователно* и *свързано*.

последователно представяне

При това представяне се запазва блок от паметта, вътре в който опашката да расте и да се съкращава. Ако редицата от елементи от един и същ тип

$a_1, a_2, ..., a_n$

е опашка с начало a_1, последователното представяне има вида:



указател към началото

указател към края

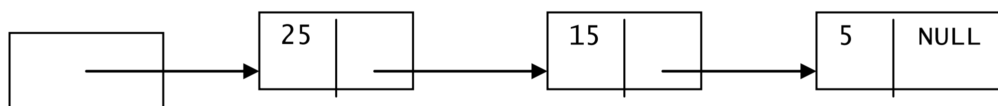
Фиг. 2.

Включването на елемент в опашката се осъществява чрез поместването му в последователни адреси в неизползваната част веднага след края на опашката. При изчерпване на масива, ако има освободена памет в началото му, включването се извършва там.

свързано представяне

Ще използваме представяне, аналогично на свързаното представяне на стек. За удобство, при реализирането на операцията включване, се въвежда указател към края.

указател към
началото



указател към края

Фиг. 3

Реализация на опашка

реализация на последователното представяне

Ще изполваме динамичен масив с размер size. С front означаваме указателя към началото, с rear – указателя към края му, а с n – текущия брой на елементите на опашката. Следва реализация на шаблон на клас, реализиращ последователно представяне на опашка.

```
const size = 100;
template <class T>
class queue
{public:
  queue();                        // конструктор
  ~queue();                       // деструктор
  queue(queue const &);           // конструктор за присвояване
  queue& operator=(queue const &);  // операторна функция за
```

```
                                     // присвояване
     void InsertElem(T const &);  //  включване  на  елемент  в
опашка
  int DeleteElem(T &);        // изключване на елемент от опашка
  void print();              // извеждане на опашка
 private:
  int front, rear, n;
  T *a;
  void delqueue();                // изтриване на опашка
  void copy(queue const &);      // копиране на опашка
};
template <class T>
queue<T>::queue()
{a=new T[size];
 n=0;
 front=0;
 rear=0;
}
template <class T>
queue<T>::~queue()
{delqueue();
}
template <class T>
queue<T>::queue(queue<T> const& r)
{copy(r);
}
template <class T>
queue<T>& queue<T>::operator=(queue<T> const& r)
{if(this!=&r)
{delqueue();
 copy(r);
}
return *this;
}
template <class T>
void queue<T>::InsertElem(T const & x)
{if (n == size) cout << "Impossible! \n";
 else
 {a[rear] = x; n++; rear++;
```

```
 rear = rear % size;
 }
}
template <class T>
int queue<T>::DeleteElem(T &x)
{if(n>0)
{x = a[front]; n--; front++;
 front = front % size;
 return 1;
}else return 0;
}
template <class T>
void queue<T>::delqueue()
{delete [] a;
}
template <class T>
void queue<T>::copy(queue<T> const &r)
{a = new T[size];
 for(int i=0; i<size; i++)
    a[i] = r.a[i];
 n = r.n;
 front = r.front;
 rear = r.rear;
}
template <class T>
void queue<T>::print()
{T x;
 while(DeleteElem(x))
   cout << x << "  ";
 cout << endl;
}
```

Включваме този шаблон във файл с име queue.cpp. Следва програма, която използва дефинирания шаблон. Забележете, файлът, съдържащ реализацията на шаблона на класа queue, е включен в програмата като заглавен файл, но е ограден в кавички, а не в “<” и “>” както е при системните заглавни файлове.

```
#include <iostream.h>
#include “queue.cpp”
```

```
void main()
{queue<int> q; // създава опашка от цели числа
 for(int i = 1; i<=10; i++)   // запълва опашката
   q.InsertElem(i);          // с целите числа от 1 до 10
 q.print();                  // извежда опашката q
 queue<char> p;              // създава опашка от символи
 for(char c='a'; i<='z'; i++) // запълва опашката
    p.InsertElem(c);         // с малките латински букви
 p.print();                  // извежда опашката p
}
```

реализация на свързаното представяне

Ще реализираме шаблон на клас queue, реализиращ свързаното представяне на опашка. Двойката

ще реализираме чрез шаблона на структурата

```
template <class T>
struct elem
{T inf;
 elem* link;
};
```

а опашката – чрез шаблона на класа queue. Указателят към началото на опашката ще означим с front, а този към края – с rear. Отново се налага реализирането на голямата тройка (деструктор, конструктор за присвояване и операторна функция за присвояване). Реализицията на това представяне има вида:

```
template <class T>
struct elem
{T inf;
 elem* link;
};
template <class T>
class queue
{public:
  queue();
```

```
  ~queue();
  queue(queue const &);
  queue& operator=(queue const &);
  void InsertElem(T const&);
  int DeleteElem(T &);
  void print();
  bool empty() const;
 private:
  elem<T> *front, *rear;
  void delqueue();
  void copy(queue const&);
};
template <class T>   // конструктор
queue<T>::queue()
{front = NULL;
 rear = NULL;
}
template <class T>   // деструктор
queue<T>::~queue()
{delqueue();
}
template <class T>   // конструктор за присвояване
queue<T>::queue(queue const & r)
{copy(r);
}
template <class T>   // операторна функция за присвояване
queue<T>& queue<T>::operator=(queue const& r)
{if(this != &r)
{delqueue();
 copy(r);
}
return *this;
}
template <class T>   // изтриване на опашка
void queue<T>::delqueue()
{T x;
 while(DeleteElem(x));
}
template <class T>      // копиране на опашка
```

```
void queue<T>::copy(queue const & r)
{rear = NULL;
 if(r.rear)
 {elem<T> *p = r.front;
  while(p)
  {InsertElem(p->inf);
   p = p->link;
  }
 }
}
template <class T>      // включване на елемент в опашка
void queue<T>::InsertElem(T const& x)
{elem<T> *p = new elem<T>;
 p->inf = x;
 p->link = NULL;
 if(rear)rear->link = p;
 else front = p;
 rear = p;
}
template <class T>   // изтриване на елемент от опашка
int queue<T>::DeleteElem(T & x)
{elem<T> *p;
 if(!rear) return 0;
 p = front;
 x = p->inf;
 if(p==rear) {rear = NULL; front = NULL;}
 else front = p->link;
 delete p;
 return 1;
}
template <class T>      // извеждане на опашка
void queue<T>::print()
{T x;
 while(DeleteElem(x))
   cout << x << " ";
 cout << endl;
}
template <class T>      // проверка за празна опашка
bool queue<T>::empty() const
```

```
{return rear == NULL;
}
```

Ще запишем този шаблон във файл с име queue-link.cpp, след което ще го включим в демонстрационната програма. В нея искаме да създадем опашка от опашки от цели числа. В нея се налага предефинирането на член-функцията print() на шаблона на класа, тъй като основната член-функция print() извежда чрез използване на <<, чрез което не може да се изведе опашка.

```
#include <iostream.h>
#include "queue-link.cpp"
// дефинираме клас IntQueue, реализиращ опашка от цели числа
typedef queue<int> IntQueue;
// дефинираме клас QueueQueue, реализиращ опашка
// от опашки от цели числа
typedef queue<IntQueue> QueueQueue;
// предефиниране на шаблонната член-функция print()
// за извеждане на опашка от опашки.
void queue<IntQueue>::print() // специализация на член-функцията
{IntQueue x;                  // print() за извеждане на
 while(DeleteElem(x))         // опашка от опашки
   x.print();
 cout << endl;
}
void main()
{QueueQueue qq; // qq е обект, означаващ опашка от опашки
 for(int i = 1; i <= 5; i++)
  {IntQueue q;    // конструиране на опашка от цели числа
   for(int j = i; j <= 2*i; j++)
    q.InsertElem(j);
   qq.InsertElem(q); // включване на опашката q в опашката qq
  }
 qq.print();
}
```

Задача 124. Да се напише програма, която само с едно преминаване през елементите на масив от числа (без използване на допълнителни масиви) извежда вурху екрана елементите на масива в следния ред: отначало всички числа, които са по-малки от a, след това всички

числа в интервала [a, b] и накрая всички останали числа, запазвайки техния първоначален ред (a и b са дадени числа, a < b).

Програма Zad124.cpp решава задачата. Тя използва две опашки q1 и q2, в които записва числата от интервала [a, b] и тези – по-големи от b, съответно в реда на тяхното срещане в масива arr.

```
Program Zad124.cpp
#include <iostream.h>
#include "queue-link.cpp"
typedef queue<int> IntQueue;
void read(int n, int *a)
{for(int i = 0; i<n; i++)
{cout << "a[" << i << "]= ";
 cin >> a[i];
}
}
void main()
{int arr[100];
 int n;
 do
 {cout << "n= ";
  cin >> n;
 }while (n<1 || n>100);
 read(n, arr);
 cout << "a<b= ";
 int a, b;
 cin >> a >> b;
 IntQueue q1, q2;
 for(int i = 0; i<n; i++)
    if(arr[i]<a) cout << arr[i] << " ";
    else if(arr[i]<=b) q1.InsertElem(arr[i]);
    else q2.InsertElem(arr[i]);
  q1.print();
  q2.print();
}
```

Задача 125. Да се напише програма, която създава две опашки, елементите на които са структури, съдържащи име и възраст на човек.

Сортира във възходящ ред по възраст елементите на опашките, след което ги слива и извежда получената опашка.

Програма Zad125.cpp решава задачата. За олесняване на сливането в края на всяка сортирана опашка добавяме фиктивен елемент, който се нарича **сентинел.** Този трик често се използва при работа с линейни динамични структури.

```
Program Zad125.cpp
#include <iostream.h>
#include "QUE-link.cpp"
struct people
{char name[31];
 int age;
};
typedef queue<people> pqueue;
void queue<people>::print() // предефиниция на шаблонната функция
{people x;
 while(DeleteElem(x))
 cout << x.name << " " << x.age << endl;
}
// за опашката q се намират лицето с най-малка възраст (min) и
// опашката без лицето с най-малка възраст (newq)
// newq е инициализирана с празната опашка
void minqueue(pqueue q, people & min, pqueue &newq)
{people x;
 q.DeleteElem(min);
 while(q.DeleteElem(x))
 if (x.age < min.age)
  {newq.InsertElem(min);
   min = x;
  }else newq.InsertElem(x);
 }
// сортиране на опашката (q) във възходящ ред
// по възраст на лицата (nq)
void sortqueue(pqueue q, pqueue& nq)
{while(!q.empty())
 {people min;
```

```
   pqueue q1;
   minqueue(q, min, q1); // q1 е инициализирана с празната опашка
   nq.InsertElem(min);
   q = q1;
  }
}
// сливане на сортираните опашки p и q
// връща резултата от сливането
pqueue merge(pqueue p, pqueue q)
{people x = {"xxx", -1}, y = {"yyy", -1}; // фиктивни елементи
 p.InsertElem(x); // включване на фиктивния елемент x в опашката
p
 q.InsertElem(y); // включване на фиктивния елемент y в опашката
q
 pqueue r;
 p.DeleteElem(x);
 q.DeleteElem(y);
 while(!p.empty() && !q.empty())
   if(x.age <= y.age)
   {r.InsertElem(x); p.DeleteElem(x);}
   else
   {r.InsertElem(y); q.DeleteElem(y);}
 if(!p.empty()) // q е празна
    do r.InsertElem(x);while (p.DeleteElem(x) && x.age!=-1);
 else // p е празна
    do r.InsertElem(y);while (q.DeleteElem(y) && y.age!=-1);
 return r;
}
void main()
{people s;
 pqueue q1;
 int i, n;
 cout << "First Queue:\n n= ";
 cin >> n;
 for(i=0; i < n; i++)
 {cout << "Name: ";
  cin>> s.name;
  cout << "Age: ";
  cin >> s.age;
```

```
  q1.InsertElem(s);
 }
 pqueue q2;
 cout << "Second Queue \n n= ";
 cin >> n;
 for(i=0; i < n;i++)
 {cout << "Name: ";
  cin>> s.name;
  cout << "Age: ";
  cin >> s.age;
  q2.InsertElem(s);
 }
 pqueue p, q, r;
 sortqueue(q1, p);
 sortqueue(q2, q);
 r = merge(p, q);
 r.print();
}
```

Задача 126. Да се напише програма, която създава опашка от опашки, елементите на които са структури, съдържащи име и възраст на човек. Да се сортират по възраст компонентите на опашката, след което да се слеят.

Програма Zad126.cpp решава задачата. Някои функции в нея са пропуснати, тъй като са същите като в задача 125.

```
Program Zad126.cpp
#include <iostream.h>
#include "queue-link.cpp"
struct people
{char name[31];
 int age;
};
typedef queue<people> pqueue;
void queue<people>::print()
{people x;
 while(DeleteElem(x))
    cout << x.name << " " << x.age << endl;
}
```

```
void queue<pqueue>::print()
{pqueue x;
 while(DeleteElem(x))
   {x.print(); cout << endl;}
}
void minqueue(pqueue q, people & min, pqueue &newq)
...
void sortqueue(pqueue q, pqueue& nq)
...
pqueue merge(pqueue p, pqueue q)
...
void main()
{queue<pqueue> q2, q3;
 int m;
 cout << “m= “; cin >> m;
 for(int j=1; j<=m; j++)
 {int i, n;
  pqueue q1;
  cout << j << " Queue:\n n= ";
  cin >> n;
  for(i=0; i < n; i++)
  {people s;
   cout << "Name: ";
   cin>> s.name;
   cout << "Age: ";
   cin >> s.age;
   q1.InsertElem(s);
  }
 q2.InsertElem(q1);
 }
 pqueue r;
 while(q2.DeleteElem(r))
 {pqueue t;
  sortqueue(r, t);
  q3.InsertElem(t);
 }
 pqueue p1, p2;
 q3.DeleteElem(p1);
 while(q3.DeleteElem(p2))
```

```
 p1 = merge(p1, p2);
 p1.print();
}
```

Задача 127. Символен низ съдържа правилен текст от вида:

<текст> ::= <интервал> | <елемент><текст>

<елемент> ::= <буква> | (<текст>).

Като се изполват опашка и/или стек, да се напише функция, която за всяка двойка съответстващи си отваряща и затваряща скоби извежда позициите им в текста, подредени по нарастване на позицията на отварящите скоби.

(Например, за текста A+(TP-F(X)*(B-C)) резултатът е 3 17; 8 10; 12 16.)

3. Свързан списък

При стековете и опашките единствният начин за извличане на данни от структурата се осъществява чрез отсраняване на елементи. Често се налага да се използват всички данни от редица от елементи без да се отстраняват елементите от редицата. За целта се използва свързан списък

Логическо описание

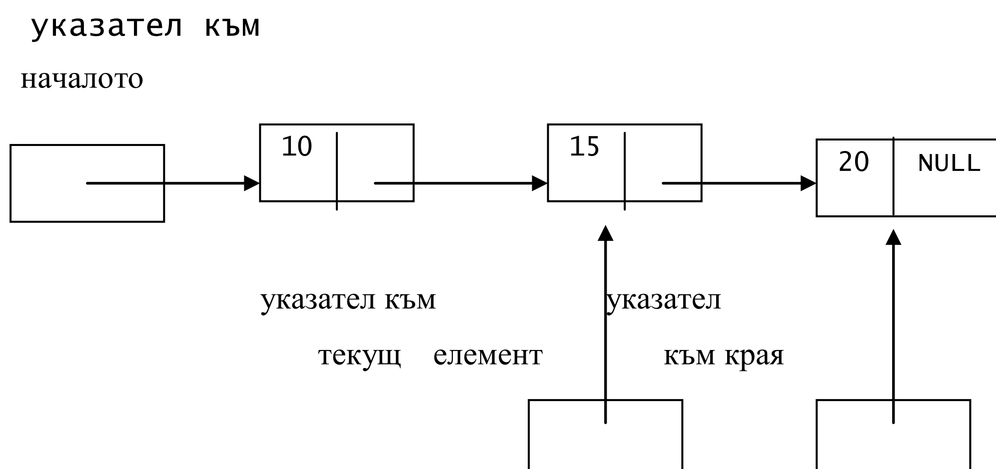
Свързаният списък е крайна редица от елементи от един и същ тип. Операциите включване и изключване са допустими в произволно място на редицата. Възможен е пряк достъп до елемента в единия край на редицата, наречен **начало на списъка**, и последователен до всеки от останалите елементи.

Физическо представяне

Има два основни начина за представяне на свързания списък в паметта на компютъра: *свързано представяне с една* и *свързано представяне с две връзки*. Има и *последователно представяне* (чрез масив от структури), което не се използва заради добре развитите средства за динамично разпределение на паметта.

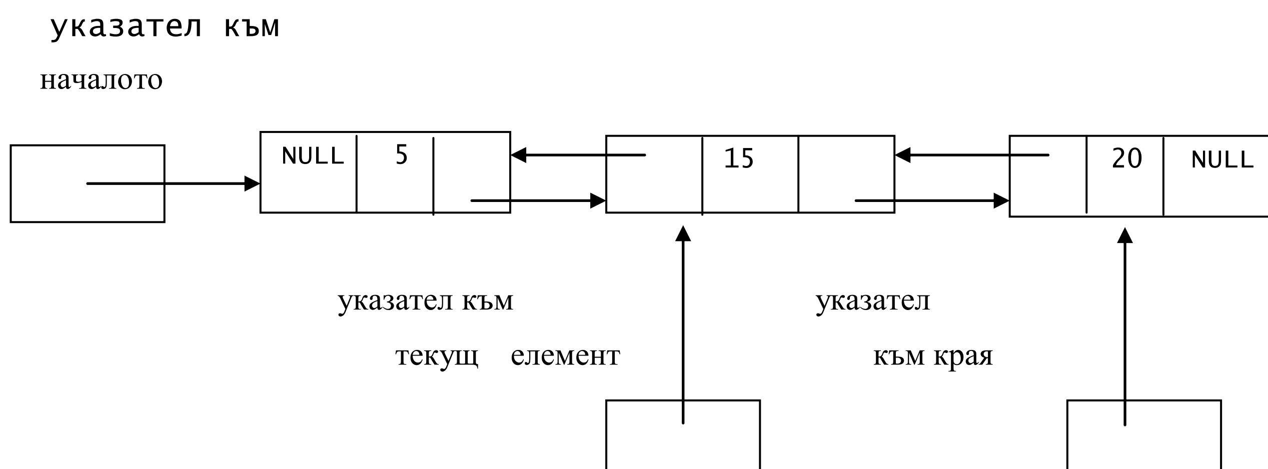
Свързано представяне с една връзка

Използва се представяне, аналогично на свързаното представяне на стек и опашка. За удобство, при реализирането на операциите включване, изключване и обхождане, се въвеждат указател към края и указател към текущ елемент на списъка.



Свързано представяне с две връзки

За удобство, при реализирането на операциите включване, изключване и обхождане, се въвеждат тройни кутии, с едно информационно и две свързващи полета, съдържащи елемента и адресите на предшестващия и следващия елементи на свързания списък.



Реализация на свързан списък с една връзка

Следният шаблон на клас реализира това представяне на свързан списък.

```
template <class T>
struct elem
{T inf;
 elem *link;
};
template <class T>
class LList
{public:
  LList();
  ~LList();
  LList(LList const&);
  LList& operator=(LList const &);
  void print();
  void IterStart(elem<T>* = NULL);
  elem<T>* Iter();
  void ToEnd(T const &);
  void InsertAfter(elem<T> *, T const &);
  void InsertBefore(elem<T> *, T const &);
  int DeleteAfter(elem<T> *, T &);
  int DeleteBefore(elem<T> *, T &);
  void DeleteElem(elem<T> *, T &);
  int len();
  void concat(LList const&);
  void reverse();
 private:
  elem<T> *Start,       // указател към началото
          *End,         // указател към края
          *Current;     // указател към текущ елемент
  void DeleteList();
  void CopyList(LList<T> const &);
};
```

Конструктора и член-функциите на голямата тройка няма да коментираме. Реализират аналогични идеи на тези при стека и опашката.

```
template <class T>   // конструктор
LList<T>::LList()
{Start = NULL;
 End = NULL;
}
template <class T>   // деструктор
LList<T>::~LList()
{DeleteList();
}
template <class T>   // конструктор за присвояване
LList<T>::LList(LList<T> const& r)
{CopyList(r);
}
template <class T>   //операторна функция за присвояване
LList<T>& LList<T>::operator=(LList<T> const & r)
{if(this!=&r)
 {DeleteList();
  CopyList(r);
 }
 return *this;
}
```

Следващите две функции са помощни и реализират изтриване на свързан списък и копиране на свързан списък на друго място в паметта. Изпозват се за реализиране на голямата тройка и затова са капсолирани в private–частта на класа. Макар, че могат да се реализират с помощта на член-функции на шаблона, по-добра е реализацията им без тях – чрез обхождане на елементите им.

изтриване на списък

```
template <class T>
void LList<T>::DeleteList()
{elem<T> *p;
 while(Start)
 {p = Start;
  Start = Start->link;
  delete p;
 }
 End = NULL;
}
```

Други реализации на тази член-функция на шаблона са:

```
template <class T>
void LList<T>::DeleteList()
{if(End)
 {T x;
  while(DeleteAfter(Start, x));
  DeleteElem(Start, x);
 }
}
```

и

```
template <class T>
void LList<T>::DeleteList()
{if(Start)
 {T x;
  while(DeleteBefore(End, x));
  DeleteElem(Start, x);
 }
}
```

копиране на списък

```
template <class T>
void LList<T>::CopyList(LList<T> const & r)
{elem<T> *p = r.Start, *q;
 if (r.Start)
 {Start = new elem<T>;
  End = Start;
  while(p)
  {End->inf = p->inf;
   End->link = NULL;
   p = p->link;
   if(p)
   {q = End;
    End = new elem<T>;
    q->link = End;
   }
  }
 }
}
```

Друга, по-лесна реализация на тази член-функция на шаблона е:

```
template <class T>
void LList<T>::CopyList(LList<T> const & r)
```

```
{Start = End = NULL;
 if(r.Start)
 {elem<T> *p = r.Start;
  while(p)
  {ToEnd(p->inf);
   p=p->link;
  }
 }
}
```

и я избираме за реализация на копирането.

извеждане на елементите на списък

Извеждането на елементите на свързан списък се реализира чрез последователното им обхождане, без разрушаването на списъка.

```
template <class T>
void LList<T>::print()
{elem<T> *p = Start;
 while(p)
 {cout << p->inf << " ";
  p = p->link;
 }
 cout << endl;
}
```

Следващите две член-функции реализират т. нар. **итератори.**

Итераторът е абстракция на означението указател към елемент на редица или по-точно може да се смята за указател към елемент на контейнер (стекът, опашката, свързаният списък са контейнери). Всеки конкретен итератор е обект от някакъв тип. РАзнообразието на типове води до разнообразие на итератори. В някои случаи итераторите са почти обикновени указатели към обекти, в други – са указател, снабден с индекс и т.н. В случая на свързан списък итераторът е указател към двойна или тройна кутия. Общото на всички итератори е тяхната семантика и имената на техните операции. Обикновено операциите са:

++ - приложена към итератор, дава итератор, който сочи към следващия елемент;

-- - приложена към итератор, дава итератор, който сочи към предшестващия елемент;

* - дава елемента, към който сочи итератора.

В шаблона на класа LList итераторът е Current. Операцията ++ е реализирана чрез член-функцията Iter. За да получим указател към началото на свързан списък, използваме процедурата IterStart. Тя е с един подразбиращ се параметър. Подразбиращата се стойност е NULL. Обръщението IterStart() установява указателя Current в началото на текущия списък. В случай, че е указан ненулев параметър Current се установява в него. Операцията * не е реализирана, тъй като Current сочи елемент, който е структура. Обръщението Iter()->inf реализира *.

итератор към началото

```
template <class T>
void LList<T>::IterStart(elem<T> *p)
{if (p) Current = p;
 else Current = Start;
}
```

Преместването на указателя Current в следващата позиция се осъществява чрез функцията Iter.

итератор за преместване в следваща позиция

```
template <class T>
elem<T>* LList<T>::Iter()
{elem<T> *p = Current;
 if(Current)Current = Current->link;
 return p;
}
```

Като използваме итератор, член-функцията на шаблона LList може да се реализира и по следния начин:

```
template <class T>
void LList<T>::print()
{IterStart();
 while(Iter())
 {cout << Iter()->inf << " ";
 }
 cout << endl;
}
```

Включването на елемент в свързан списък реализираме чрез следните три член-функции:

включване на елемент в края на списъка

```
template <class T>
void LList<T>::ToEnd(T const & x)
```

```
{Current = End;
 End = new elem<T>;
 End->inf = x;
 End->link = NULL;
 if(Current) Current->link = End;
 else Start = End;
}
```

включване на елемент след указан елемент

```
template <class T>
void LList<T>::InsertAfter(elem<T> *p, T const & x)
{elem<T> *q = new elem<T>;
 q->inf = x;
 q->link = p->link;
 if(p==End) End = q;
 p->link = q;
}
```

включване на елемент пред указан елемен

```
template <class T>
void LList<T>::InsertBefore(elem<T> * p, T const& x)
{elem<T> *q = new elem<T>;
 *q = *p;
 if(p==End) End = q;
 p->inf = x;
 p->link = q;
}
```

Изключването на елемент от свързан списък реализираме чрез следните член-функции:

изтриване на елемент след указан елемент

```
template <class T>
int LList<T>::DeleteAfter(elem<T> *p, T &x)
{if(p->link)
{elem<T> *q = p->link;
 x = q->inf;
 p->link = q->link;
 if(q==End) End = p;
 delete q;
 return 1;
 }
 else return 0;
```

```
}
```
изтриване на указан елемент
```
template <class T>
void LList<T>::DeleteElem(elem<T> *p, T &x)
{if(p==Start)
 {x = p->inf;
  if(Start == End){Start = End = NULL;}
  else Start = Start->link;
  delete p;
 }
 else
 {elem<T> *q = Start;
  while(q->link!=p)q = q->link;
  DeleteAfter(q, x);
 }
}
```
изтриване на елемент пред указан елемент
```
template <class T>
int LList<T>::DeleteBefore(elem<T> *p, T &x)
{if(p!=Start)
{elem<T> *q=Start;
while(q->link!=p)q = q->link;
 DeleteElem(q, x);
 return 1;
}else return 0;
}
```
дължина на списък

Член функцията len намира броя на елементите на свързан списък.
```
template <class T>
int LList<T>::len()
{int n = 0;
 IterStart();
 elem<T> *p= Iter();
 while(p)
 {n++;
  p = Iter();
 }
 return n;
}
```

или

```
template <class T>
int LList<T>::len()
{int n = 0;
 elem<T> *p = Start;
 while(p)
 {n++;
  p=p->link;
 }
 return n;
}
```

конкатенация на списъци

Следващата член-функция реализира конкатенация на неявния свързан списък с указания като формален параметър. В резултат в края на неявния списък са включени елементите на указания списък. Така неявния списък е разрушен (носи резултата).

```
template <class T>
void LList<T>::concat(LList<T> const &L)
{elem<T> *p = L.Start;
 while(p)
 {ToEnd(p->inf);
  p = p->link;
 }
}
```

Често пъти вместо тази функция се използва следната:

```
template <class T>
void LList<T>::concat(LList const& L)
{End->link = L.Start;
}
```

Тя е неправилна, тъй като една и съща редица от елементи е достъпна чрез два обекта на класа LList, т.е. имат поделена част.

обръщане елементите на списък

Следващата член-функция на класа LList обръща елементите на неявния параметър като ползва помощен списък, с който работи като със стек.

```
template <class T>
void LList<T>::reverse()
{LList<T> l;
 IterStart();
```

```
  elem<T> *p = Iter();
  if(p)
  {l.ToEnd(p->inf);
   p = p->link;
   while(p)
   {l.InsertBefore(l.Start, p->inf);
    p = p->link;
   }
  }
  *this = l;
 }
```

Следващата реализация на reverse обръща елементите на непразен списък, зададен чрез неявния параметър без да прави негово копие в паметта. Тя поправя връзките в него така, че първите елементи да станат последни и обратно.

```
template <class T>
void LList<T>::reverse()
{elem<T> *p, *q, *temp;
 p = Start;
 q = NULL;
 Start = End;
 while(p!=Start)
 {temp = p->link;
  p->link = q;
  q = p;
  p = temp;
 }
 p->link = q;
}
```

Забележка: Тази член-функция не работи добре при повторно извикване за обръщане на списъка. Намерете грешката.

Следва верният код на reverse:

```
template <class T>
void LList<T>::reverse()
{elem<T> *p, *q, *temp;
 p = Start;
 if(p)
 {q = NULL;
  temp = Start;
```

```
      Start = End;
      End = temp;
      while(p!=Start)
      {temp = p->link;
       p->link = q;
       q = p;
       p = temp;
      }
      p->link = q;
     }
   }
```

Този шаблон на класа LList записваме във файла L-List.cpp.

Ще демонстрираме създадения шаблон на клас.

```
#include <iostream.h>
#include "L-List.cpp"
void main()
{LList<int> l1, l2;
 // създава списъка l1 с елементи 1, 2, 3, 4
 l1.ToEnd(1); l1.ToEnd(2);
 l1.ToEnd(3); l1.ToEnd(4);
 // създава списъка l2 с елементи 5, 6
 l2.ToEnd(5); l2.ToEnd(6);
 // създава списъка l3 чрез конструктора за присвояване
 LList<int> l3 = l1;
 // извежда списъците
 cout << "l1= "; l1.print();
 cout << "l2= "; l2.print();
 cout << "l3= "; l3.print();
 // обръща списъка l1 два пъти
 l1.reverse();
 cout << "L1-rev: "; l1.print();
 l1.reverse();
 cout << "L1-rev: "; l1.print();
}
```

Чрез дефинирания шаблон на класа LList, като използваме InsertBefore и DeleteElem, можем да работим със списъка като със стек.

Пример:

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

```
#include "L-List.cpp"
void main()
{// създаване на стек
 LList<int> l1;
 l1.ToEnd(0);
 l1.IterStart();
 elem<int>* p = l1.Iter();
 for(int i = 1; i<=9; i++)
 l1.InsertBefore(p, i);
 // изключване на елементи от стека
 l1.IterStart();
 p = l1.Iter();
 int x;
 l1.DeleteElem(p, x);
 cout << x << " ";
 l1.IterStart();
 p = l1.Iter();
 l1.DeleteElem(p, x);
 cout << x << " ";
}
```

Аналогично, като използваме InsertAfter и DeleteElem, можем да работим със списъка като с опашка.

Пример:

```
#include <iostream.h>
#include "L-List.cpp"
void main()
{LList<int> l1;
 l1.ToEnd(1);
 l1.IterStart();
 elem<int>* p = l1.Iter();
 for(int i = 2; i<=10; i++)
   l1.ToEnd(i);
 l1.print();
 l1.IterStart();
 p = l1.Iter();
 int x;
 l1.DeleteElem(p, x);
 cout << "x= " << x << endl;
 l1.IterStart();
```

```
 p = l1.Iter();
 l1.DeleteElem(p, x);
 cout << "x= " << x << endl;
 l1.print();
}
```

Задача 128. Да се дефинира шаблон на функция, която да увеличава елементите на свързан списък от тип T с елемента a от тип T (T е числов тип).

```
template <class T>
LList<T> increase(LList<T> L, T a)
{elem<T> *p;
 L.IterStart();
 p = L.Iter();
 while(p)
 {p->inf = p->inf + a;
  p = L.Iter();
 }
 return L;
}
```

Задача 129. Даден е списък от цели числа. Да се напише функция, която:

а) проверява дали дадено цяло число се съдържа в списъка;

б) намира максималния елемент на списъка;

в) изтрива първото срещане на дадено цяло число;

г) изтрива всяко срещане на дадено цяло число;

д) извежда в обратен ред елементите на списъка.

Ще направим следната специализация на класа LList.

```
typedef LList<int> IntList;
```

а)

```
bool member(int x, IntList l)
{l.IterStart();
 elem<int> *p = l.Iter();
 if(!p) return false;
 int y;
 l.DeleteElem(p, y);
```

```
 return x==y || member(x, l);
}
```
б)
```
int maxelem(IntList l)
{int x;
 l.IterStart();
 elem<int> *p=l.Iter();
 if(!p->link) return p->inf;
 l.DeleteElem(p, x);
 int y = maxelem(l);
 if(x>=y) return x;
 else return y;
}
```
в)
```
void deletefirst(int a, IntList& l)
{int x;
 l.IterStart();
 elem<int> *p = l.Iter();
 while(p && p->inf != a)p=l.Iter();
 if (p->inf==a) l.DeleteElem(p, x);
}
```
или
```
void deletefirst(int a, IntList& l)
{l.IterStart();
 elem<int> *p=l.Iter();
 while(p)
 if(p->inf==a)
 {int x;
  l.DeleteElem(p, x); p = NULL;
 }
 else p = l.Iter();
}
```
г)
```
void deleteall(int a, IntList& l)
{while(member(a, l))
   deletefirst(a, l);
}
```
или
```
void deleteall(int a, IntList& l)
```

```
{int x;
 l.IterStart();
 elem<int> *p=l.Iter();
 while(p)
 {if(p->inf==a)l.DeleteElem(p, x);
  p=l.Iter();
 }
}
```
д)

```
void print_reverse(IntList l)
{int x;
 l.IterStart();
 elem<int> *p = l.Iter();
 if(p)
 {l.DeleteElem(p, x);
  print_reverse(l);
  cout << x << " ";
 }
}
```

Сортиране на списъци

пряка селекция

Шаблонът на функцията sortlist реализира метода на пряката селекция за сотриране на списъци с елементи от тип T, допускащи сравнение.

```
template <class T>
void sortlist(LList<T> &l)
{elem<T>* mp, *p;
 l.IterStart(); p = l.Iter();
 while(p->link)
 {T min = p->inf;
  mp = p;
  elem<T> *q = p->link;
   while(q)
   {if(q->inf<=min)
   {mp = q;
    min = q->inf;
```

```
      }
       q=q->link;
      }
      min = mp->inf;
      mp->inf = p->inf;
      p->inf = min;
      p = p->link;
     }
   }
```

сливане на сортирани свързани списъци

Шаблонът на функция mergelists реализира сливане на списъците l1 и l2 и връща резултата от сливането.

```
template <class T>
LList<T> mergelists(LList<T> l1, LList<T> l2)
{LList<T> l;
 l1.IterStart();
 l2.IterStart();
 elem<T> *p = l1.Iter(),
         *q = l2.Iter();
 while(p && q)
    if(p->inf <= q->inf)
    {l.ToEnd(p->inf);
     p=l1.Iter();
    }
    else
    {l.ToEnd(q->inf);
     q=l2.Iter();
    }
 if(q)
    while(q)
    {l.ToEnd(q->inf);
     q=l2.Iter();
    }
    else while(p)
    {l.ToEnd(p->inf);
     p=l1.Iter();
    }
 return l;
```

```
}
```

Задача 130. Да се напише програма, която създава списък от списъци от имена на хора, сортира компонентите на списъка, след което ги слива и извежда получения списък.

Програма Zad130.cpp решава задачата.

```
Program Zad130.cpp
#include <iostream.h>
#include <string.h>
#include "L-List.cpp"
typedef char str[31];
// дефиниране на клас strlist,
// реализиращ списък от имена на хора
typedef LList<str> strlist;
// дефиниране на клас list_of_lists,
// реализиращ списък от списъци от имена на хора
typedef LList<strlist> list_of_lists;
// специализация на член-функцията print() на шаблона
// за извеждане на списък от списъци от имена на хора
void LList<strlist>::print()
{elem<strlist> *p = Start;
 while(p)
 {p->inf.print();
  p=p->link;
 }
}
// специализация на член-функцията за включване на
// елемент в края на свързан списък
void LList<str>::ToEnd(str const & x)
{Current = End;
 End = new elem<str>;
 strcpy(End->inf, x);
 End->link = NULL;
 if(Current)Current->link=End;
 else Start = End;
}
// предефиниране на нова sortlist
// за сортиране на списък от хора
```

```
void sortlist(strlist &l)
{elem<str>* mp, *p;
 l.IterStart(); p = l.Iter();
 while(p->link)
 {str min;
  mp = p;
  strcpy(min, p->inf);
  elem<str> *q = p->link;
  while(q)
  {if(strcmp(q->inf,min)<=0)
  {mp = q;
   strcpy(min, q->inf);
  }
   q=q->link;
  }
   strcpy(min, mp->inf);
   strcpy(mp->inf, p->inf);
   strcpy(p->inf, min);
   p = p->link;
  }
}
// предефиниране на mergelists за сливане на
// сортирани списъци от имена на хора
strlist mergelists(strlist l1, strlist l2)
{strlist l;
 l1.IterStart();
 l2.IterStart();
 elem<str> *p = l1.Iter(),
           *q = l2.Iter();
 while(p && q)
    if(strcmp(p->inf,q->inf)<=0)
    {l.ToEnd(p->inf);
     p=l1.Iter();
    }
    else
    {l.ToEnd(q->inf);
     q=l2.Iter();
    }
 if(q)
```

```
    while(q)
    {l.ToEnd(q->inf);
     q=l2.Iter();
    }
    else while(p)
    {l.ToEnd(p->inf);
     p=l1.Iter();
    }
 return l;
}
void main()
{// създаване на списък от списъци от хора
 list_of_lists ll;
 cout << "брой на списъците в списъка от списъци: ";
 int n;
 cin >> n;
 for(int k=1; k<=n; k++)
 {strlist l;
  str s;
  cout << "брой на хората в списъка: ";
  int p;
  cin >> p;
  for(int i=1; i<=p; i++)
  {cout << "s= ";
   cin >> s;
   l.ToEnd(s);
  }
  ll.ToEnd(l);
 }
 ll.print();
 // обхождане на елементите на ll и
 // сортиране на всеки от съставящите го списъци
 ll.IterStart();
 elem<strlist> *p=ll.Iter();
 while(p)
 {sortlist(p->inf);
  p=ll.Iter();
 }
 ll.print();
```

```
 // сливане на списъците, изграждащи ll
 ll.IterStart();
 p=ll.Iter();
 strlist l=p->inf;
 p=ll.Iter();
 while(p)
 {l = mergelists(l, p->inf);
  p=ll.Iter();
 }
 l.print();
}
```

сортиране чрез сливане

Този вид сортиране се осъществява по следния начин: списъкът се разделя на две половини. За всяка половинка чрез същия начин за сортиране тези половинки се сортират. Накрая сортираните половинки се сливат.

Шаблонът mergesort реализира този начин за сортиране.

```
template <class T>
void mergesort(LList<T> &l)
{LList<T> l1, l2;
 l.IterStart();
 elem<T> *p = l.Iter();
 if(!p || p->link == NULL) return;
 while(p)
 {l1.ToEnd(p->inf);
  p=p->link;
  if(p)
  {l2.ToEnd(p->inf);
   p=p->link;
  }
 }
 mergesort(l1);
 mergesort(l2);
 l = mergelists(l1, l2);
}
```

Проверка на свойства на списъци

В следващите две задачи ще проверим дали елементите на списък са монотонно намаляващи и дали са различни.

Задача 131. Да се напише програма, която създава списък от имена на хора и проверява дали елементите му са подредени лексикографски.

Програма Zad131.cpp решава задачата.

```
Program Zad131.cpp
#include <iostream.h>
#include <string.h>
#include "L-List.cpp"
typedef char str[31];
typedef LList<str> strlist;
typedef LList<strlist> list_of_lists;

void LList<strlist>::print()
{elem<strlist> *p = Start;
 while(p)
 {p->inf.print();
  p=p->link;
 }
}
void LList<str>::ToEnd(str const & x)
{Current = End;
 End = new elem<str>;
 strcpy(End->inf, x);
 End->link = NULL;
 if(Current)Current->link=End;
 else Start = End;
}
bool monotone(strlist l)
{l.IterStart();
 elem<str> *p=l.Iter(), *q, *r;
 if(!p->link) return true;
 q=p->link;
 r=q->link;
```

```
 while(strcmp(p->inf, q->inf)<=0 && r)
 {p=q;
  q=r;
  r=r->link;
 }
 return strcmp(p->inf, q->inf)<=0;
}
void main()
{strlist l;
 str s;
 cout << "broj= ";
 int p;
 cin >> p;
 for(int i=1; i<=p; i++)
 {cout << "s= ";
  cin >> s;
  l.ToEnd(s);
 }
 l.print();
 cout << monotone(l) << endl;
}
```

Задача 132. Да се напише програма, която създава списък от цели числа и проверява дали елементите му са различни.

Програма Zad132.cpp решава задачата.

```
Program Zad132.cpp
#include <iostream.h>
#include "L-List.cpp"
typedef LList<int> IntList;
bool member(int x, IntList l)
{l.IterStart();
 elem<int> *p = l.Iter();
 if(!p) return false;
 int y;
 l.DeleteElem(p, y);
 return x==y || member(x, l);
}
bool diffr(IntList l)
```

```
{l.IterStart();
 elem<int> *p = l.Iter();
 if(!p->link) return true;
 int y;
 l.DeleteElem(p, y);
 return !member(y, l) && diffr(l);
}
void main()
{IntList l;
 int s;
 cout << "broj= ";
 int p;
 cin >> p;
 for(int i=1; i<=p; i++)
 {cout << "s= ";
  cin >> s;
  l.ToEnd(s);
  }
cout << diffr(l) << endl;
}
```

Глава 16

Йерархични структури от данни

двоично дърво и граф

3. Двоично дърво

Логическо описание

Двоично дърво от тип T е структура от данни, която е или празна, или е образувана от

- данна от тип Т, наречена **корен** (връх) на двоичното дърво от тип Т;

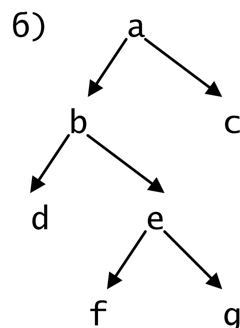
- двоично дърво от тип Т, наречено **ляво поддърво** на двоичното дърво от тип Т (ЛПД);

- двоично дърво от тип Т, наречено **дясно поддърво** на двоичното дърво от тип Т (ДПД).

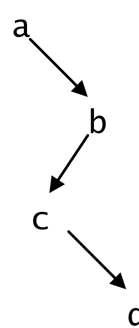
Примери:

Нека a, b, c, d, e, f и g са данни от тип Т. Тогава следните графични представяния определят двоични дървета от тип Т.

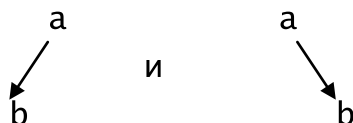
а) a б)



в)



Посоката на линиите, свързващи върховете с поддървета, позволява да се различи ляво от дясно поддърво. Двоичните дървета

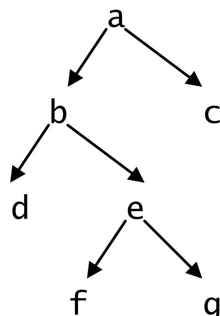


са различни. В единия случай дясното поддърво е празно, а в другия - дясното дърво не е празно.

Някои определения

Листо - това е връх с празни поддървета.

Пример: Върховете d, c, f и g



са листа.

Върховете, които не съвпадат с корена и листата, се наричат **вътрешни върхове**.

Пример: b и e, от примера по-горе, са вътрешни върхове на двоичното дърво от тип Т.

Всяко поддърво се нарича **наследник (син)** по отношение на своето дърво и **родител (баща)** по отношение на своите поддървета.

Над структурата от данни двоично дърво са възможни следните операции:

- достъп до връх

Възможен е пряк достъп до корена и непряк достъп до всеки от останалите върхове на двоичното дърво.

- включване и изключване на връх

Включването и изключването са възможни в произволно място на двоичното дърво. В резултат се получава двоично дърво от тип T.

Обхождане на двоично дърво

Обхождането на двоично дърво дава метод, позволяващ да се осъществи достъп до всеки връх на дървото един единствен път.

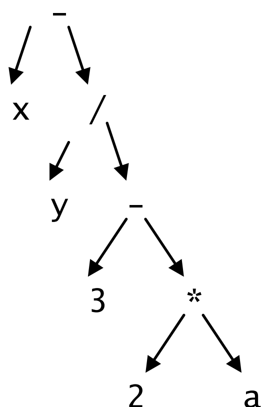
Осъществява се чрез изпълнение на следните три действия в някаква последователност:

- обработка на корена;
- обхождане на лявото поддърво;
- обхождане на дясното поддърво,

т.е. процесът на обхождане е рекурсивен. Съществуват шест различни начина за обхождане на двоично дърво: ЛКД (**смесено обхождане**), КЛД (**низходящо обхождане**), ЛДК (**възходящо обхождане**), ДКЛ, КДЛ и ДЛК, където К – означава корен, Л – ляво поддърво, Д – дясно поддърво. Обхождането ЛКД например означава, че първо се обхожда лявото поддърво, след него се обработва коренът и накрая се обхожда дясното поддърво.

Всеки аритметичен израз може да се представи чрез двоично дърво, в листата на което са операндите, а във вътрешните върхове и корена – операциите.

Пример: Изразът x-y/(3-2*a) може да се представи чрез двоично дърво по следния начин



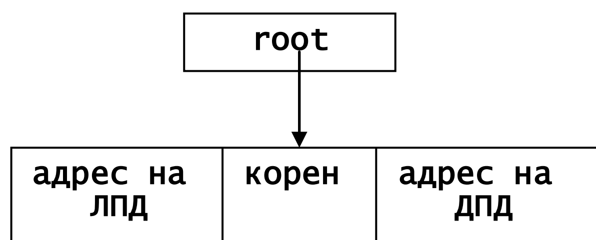
Възходящият обход на двоично дърво, представящо аритметичен израз, дава обратния полски запис на аритметичния израз. Смесеният обход на двоично дърво, представящо аритметичен израз, дава общоприетия (инфиксен) запис на аритметичния израз, но без скобите, а низходящият обход на двоично дърво, представящо аритметичен израз, дава представяне, което се нарича **прав полски запис.**

Физическо представяне

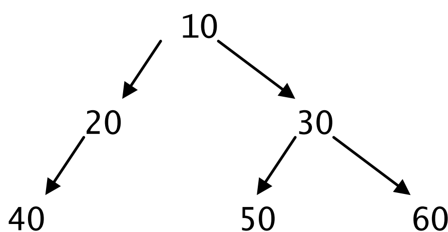
Използват се главно два начина за физическо представяне на двоично дърво от тип T – **свързано** и **верижно.**

Свързано представяне

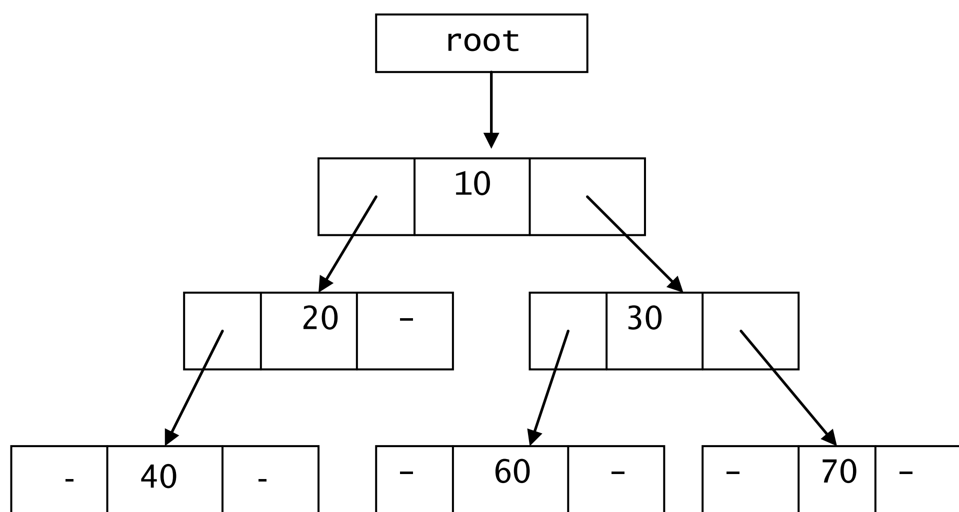
Реализира се чрез указател към кутия с три полета: информационно, съдържащо стойността на корена и две адресни, съдържащи представянията на ЛПД и ДПД съответно, т.е.



Пример: Двоичното дърво



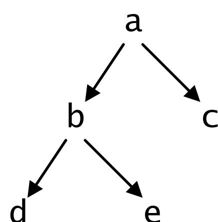
се представя по следния начин:



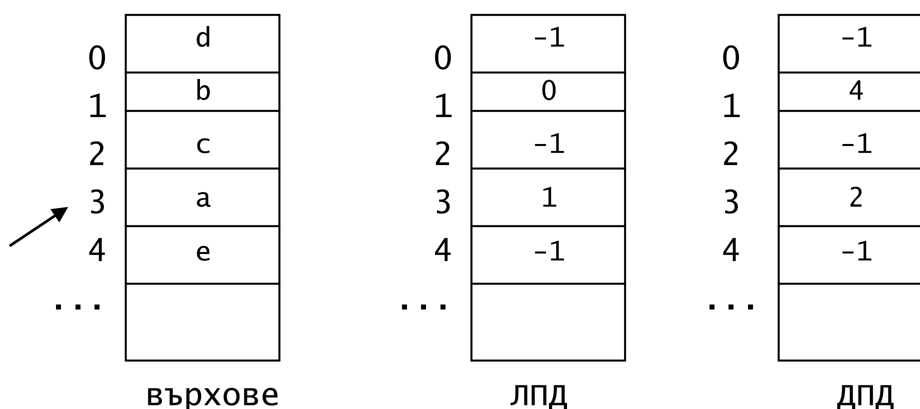
Верижно представяне

При това представяне се използват три масива - за елементите на дървото, за адресите на лявото и за адресите на дясното поддърво. Ролята на адреси се изпълнява от индекси. i-ят елемент на масива "върхове" съдържа стойността на връх на двоичното дърво, i-ят елемент на ЛПД съдържа адреса на лявото поддърво на поддървото с корен i-я елемент, а i-ят елемент на ДПД - адреса на дясното му поддърво. Има указател, който съдържа адреса на корена.

Пример: Верижното представяне на двоичното дърво от тип T



се представя по следния начин:



Указателят към корена е 3. Празното дърво представяме чрез –1.

В следващите разглеждания ще използваме свързаното представяне на двоично дърво.

Реализация на свързаното представяне

Тройната кутия, съдържаща корена и адресите на лявото и дясното поддървета, представяме чрез следния шаблон на структурата node.

```
template <class T>
struct node
{T inf;
 node *Left,
      *Right;
};
```

Двоичното дърво ще представим чрез указател към тройна кутия от вида, описан по-горе. Ще го реализираме чрез шаблона на класа tree

```
template <class T>
class tree
{public:
 tree();
 ~tree();
 tree(tree const&);
 tree& operator=(tree const&);
 bool empty()const;
 T RootTree() const;
 tree LeftTree()const;
 tree RightTree()const;
 void Create3(T, tree<T>, tree<T>);
 void print() const
 {pr(root);
  cout << endl;
 }
 void Create()
 {CreateTree(root);
 }
 private:
  node<T> *root;
  void DeleteTree(node<T>* &)const;
  void Copy(node<T> * &, node<T>* const&)const;
  void CopyTree(tree<T> const&);
  void pr(const node<T> *) const;
  void CreateTree(node<T> * &)const;
};
```

Конструкторът и функциите на голямата тройка реализират познати идеи.

```
template <class T>
tree<T>::tree()
{root = NULL;
}
template <class T>
tree<T>::~tree()
{DeleteTree(root);
}
template <class T>
tree<T>::tree(tree<T> const& r)
{CopyTree(r);
}
template <class T>
tree<T>& tree<T>::operator=(tree<T> const& r)
{if(this!=&r)
{DeleteTree(root);
 CopyTree(r);
}
 return *this;
}
```

За реализирането им използваме член-функциите DeleteTree, CopyTree и Copy на шаблона на класа tree.

```
template <class T>
void tree<T>::DeleteTree(node<T>* &p)const
{if(p)
 {DeleteTree(p->Left);
  DeleteTree(p->Right);
  delete p;
  p=NULL;
 }
}
template <class T>
void tree<T>::CopyTree(tree<T> const& r)
{Copy(root, r.root);
}
template <class T>
void tree<T>::Copy(node<T> * & pos, node<T>* const & r)const
{pos=NULL;
 if(r)
```

```
 {pos = new node<T>;
  pos->inf = r->inf;
  Copy(pos->Left, r->Left);
  Copy(pos->Right, r->Right);
 }
}
```

Използването на допълнителния параметър в DeleteTree и Copy от тип node<T>* е заради рекурсията.

Проверката дали двоично дърво е празно се осъществява чрез булевата член-функция empty() на шаблона.

```
template <class T>
bool tree<T>::empty()const
{return root==NULL;
}
```

Достъпът до корена, до лявото и до дясното поддърво на дадено двоично дърво се осъществява чрез член-функциите: RootTree(), LeftTree() и RightTree().

```
template <class T>
T tree<T>::RootTree()const
{return root->inf;
}
template <class T>
tree<T> tree<T>::LeftTree()const
{tree<T> t;
 Copy(t.root, root->Left);
 return t;
}
template <class T>
tree<T> tree<T>::RightTree()const
{tree<T> t;
 Copy(t.root, root->Right);
 return t;
}
```

Извеждането на елементите на двоично дърво става чрез член-функцията print(). Тъй като за реализацията ще използваме механизма на рекурсията, print() използва помощната член-функция pr.

```
template <class T>
void tree<T>::pr(const node<T>*p)const
{if(p)
```

```
 {pr(p->Left);
  cout << p->inf << " " ;
  pr(p->Right);
 }
}
```

Следните две член-функции създават двоично дърво. Функцията Create3 създава двоично дърво по дадени корен, ляво и дясно поддървета.

```
template <class T>
void tree<T>::Create3(T x, tree<T> l, tree<T> r)
{root = new node<T>;
 root->inf = x;
 Copy(root->Left, l.root);
 Copy(root->Right, r.root);
}
```

Член-функцията Create създава произволно двоично дърво. Тя използва капсолираната член-функция CreateTree, дефинирана рекурсивно по следния начин:

```
template <class T>
void tree<T>::CreateTree(node<T> * & pos)const
{T x; char c;
 cout << "root: ";
 cin >> x;
 pos = new node<T>;
 pos->inf=x;
 pos->Left=NULL;
 pos->Right=NULL;
 cout << "Left Tree of: " << x << "  y/n? ";
 cin >> c;
 if(c=='y')CreateTree(pos->Left);
 cout << "Right Tree of: " << x << " y/n? ";
 cin >> c;
 if(c=='y')CreateTree(pos->Right);
}
```

Записваме този шаблон във файла Tree.cpp.

Задача 143. Да се напише програма, която създава двоично дърво от цели числа. Намира и извежда корена, лявото и дясното му

поддървета. Конструира и извежда двоично дърво с корен, съвпадащ с корена на първоначалното двоично дърво, с ляво поддърво – дясното поддърво на първоначалното и дясно поддърво – лявото поддърво на първоначалното двоично дърво.

Програма Zad143.cpp решава задачата.

```
Program Zad143.cpp
#include <iostream.h>
#include "Tree.cpp"
typedef tree<int> IntTree;
void main()
{IntTree t;
 t.Create();
 t.print();
 IntTree t1 = t.LeftTree(),
         t2 = t.RightTree();
 int x = t.RootTree();
 cout << "Root: " << x << endl;
 cout << "LeftTree: \n";
 t1.print();
 cout << "RightTree: \n";
 t2.print();
 IntTree t3;
 t3.Create3(x, t2, t1);
 t3.print();
}
```

Задача 144. Да се напише функция, която увеличава всеки от върховете на двоично дърво от цели числа с дадено цяло число.

Програма Zad144.cpp решава задачата.

```
Program Zad144.cpp
#include <iostream.h>
#include "Tree.cpp"
typedef tree<int> IntTree;
IntTree AddElem(int a, IntTree const& t)
{IntTree t1;
 if(!t.empty())
 t1.Create3(t.RootTree()+a,
```

```
            AddElem(a, t.LeftTree()),
            AddElem(a, t.RightTree()));
 return t1;
 }
void main()
{IntTree t;
 t.Create();
 t.print();
 cout << "Number: ";
 int a; cin >> a;
 AddElem(a, t).print();
}
```

Задача 145. Да се дефинира функция от по-висок ред map, прилагаща едноаргументната функция f към всеки от елементите на дадено двоично дърво.

В програма Zad145.cpp е дадена дефиницията на функцията map и използването ѝ за увеличаване с 1 на всеки от елементите на двоично дърво от тип int, а също за прилагане на функцията "факториел" към всеки връх на дадено двоично дърво.

```
Program Zad145.cpp
#include <iostream.h>
#include "Tree.cpp"
typedef tree<int> IntTree;
template <class T>
tree<T> map(T (*f)(T), tree<T> const &t)
{tree<T> t1;
 if(!t.empty())
 t1.Create3(f(t.RootTree()), map(f, t.LeftTree()),
                             map(f, t.RightTree()));
 return t1;
}
int f(int x)
{return x+1;
}
int g(int x)
{if(x==0) return 1;
 return x*g(x-1);
```

```
}
void main()
{IntTree t;
 t.Create();
 t.print();
 map(f, t).print();
 map(g, t).print();
}
```

Задача 146. Да се дефинира функция от по-висок ред accumulate, прилагаща бинарната лявоасоциативна операция op над елементите на дадено двоично дърво в реда ЛКД. Да се използва accumulate за намиране на сумата и произведението на елементите на дадено двоично дърво от тип int.

```
Program Zad146.cpp
#include <iostream.h>
#include "Tree.cpp"
typedef tree<int> IntTree;
template <class T>
T accumulate(T (*op)(T, T), T null_val, tree<T> const &t)
{if(!t.empty())
return               op(op(accumulate(op,null_val,t.LeftTree()),
t.RootTree()),
          accumulate(op, null_val, t.RightTree()));
return null_val;
}
int sum(int x, int y)
{return x+y;
}
int po(int x, int y)
{return x*y;
}
void main()
{IntTree t;
 t.Create();
 t.print();
 cout << accumulate(sum, 0, t) << endl;
 cout << accumulate(po, 1, t) << endl;
}
```

Задача 147. Да се дефинира булева функция equal, която установява дали двоичните дървета t1 и t2 от тип T, са равни.

```
template <class T>
bool equal(tree<T> const &t1, tree<T> const &t2)
{if(t1.empty()&&t2.empty()) return true;
 if(t1.empty()&&!t2.empty() ||
    !t1.empty()&&t2.empty()) return false;
 if(t1.RootTree()!=t2.RootTree())return false;
 return equal(t1.LeftTree(), t2.LeftTree())&&
        equal(t1.RightTree(), t2.RightTree());
}
```

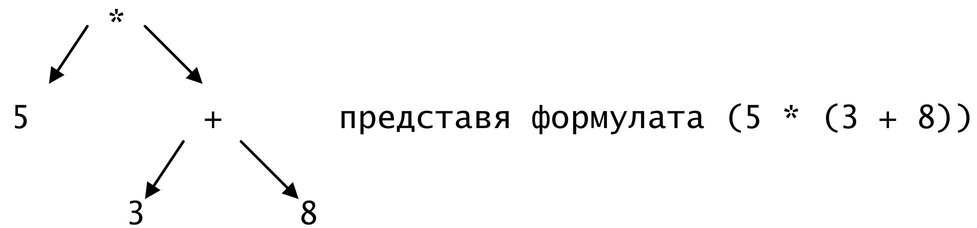
Задача 148. Да се напише функция depth, която намира дълбочината на двоично дърво от тип T, т.е. броя на върховете в най-дългия път от корена до листо.

```
template <class T>
int depth(tree<T> const&t)
{if(t.empty())return 0;
 int n, m;
 n=depth(t.LeftTree());
 m=depth(t.RightTree());
 if(n>m)return n+1;
 return m+1;
}
```

Задача 149. Формулата

<формула> ::= <терминал>|
              (<формула><знак><формула>)
<знак> ::= +|-|*|/
<терминал> ::= 0|1| ...|9

може да се представи във вид на двоично дърво от тип char съгласно следното правило: формула състояща се от един терминал се представя чрез двоично дърво с един връх – цифрата; формула от вида (f1 s f2) се представя чрез двоично дърво, коренът на което е знака s, ЛПД съответства на формулата f1, ДПД – на формулата f2. Например, двоичното дърво



Да се напише рекурсивна функция или процедура, която:

а) създава двоично дърво, представящо формула от горния вид;

б) проверява, явява ли се двоичното дърво d, дърво на формула.

в) намира стойността на формула, представена с двоичното дърво d;

г) извежда двоичното дърво d във вид, съответстващ на формулата;

Програма Zad149.cpp решава задачата. Дървото създаваме чрез член-функцията Create на шаблона на класа tree.

Program Zad149.cpp

```
#include <iostream.h>
#include "Tree.cpp"
typedef tree<char> CharTree;
// б)
bool IsForm(CharTree const&t)
{if(t.empty())return false;
 char c = t.RootTree();
 if(c>='0' && c<='9')
    return t.LeftTree().empty() && t.RightTree().empty();
 if(c!='+' && c!='-' && c!='*' && c!='/')return false;
 return IsForm(t.LeftTree()) && IsForm(t.RightTree());
}
// в)
int ArExpr(CharTree const&t)
{char c = t.RootTree();
 if(c>='0'&&c<='9')return (int)c-(int)'0';
 switch(c)
 {case '+':return ArExpr(t.LeftTree())+ArExpr(t.RightTree());
  case '-':return ArExpr(t.LeftTree())-ArExpr(t.RightTree());
  case '*':return ArExpr(t.LeftTree())*ArExpr(t.RightTree());
  case '/':return ArExpr(t.LeftTree())/ArExpr(t.RightTree());
 }
 return 99999;
}
```

```
// г)
void print_tree(CharTree const&t)
{char c = t.RootTree();
 if(c>='0' && c<='9')cout << c;
 else
 {cout << '(';
  print_tree(t.LeftTree());
  cout << c;
  print_tree(t.RightTree());
  cout << ')';
 }
}
void main()
{CharTree t;
 t.Create();
 if(IsForm(t))
 {print_tree(t);
  cout << endl << ArExpr(t) << endl;
 }
 else cout << "Is not a formula.\n";
}
```

Задача 150. Нека в двоичното дърво от тип char е записана формула според синтаксиса от Задача 149, но в качество на терминали се използват не само цифри, а и букви, играещи ролята на променливи. Да се напише функция, която:

а) опростява двоично дърво, представящо формула като заменя в него всички поддървета, съответстващи на формулите (f+0), (0+f), (f-0), (f*1), (1*f) и (f/1) с поддърво, съответстващо на формулата f, а поддърветата, съответстващи на формулите (f*0), (0*f) и (0/f) - с върха 0.

б) преобразува двоично дърво, представящо формула като заменя в него всички поддървета, съответстващи на формулите ((f1+f2)*f3), ((f1-f2)*f3), (f1*(f2+f3)), (f1*(f2-f3)), ((f1+f2)/f3) и ((f1-f2)/f3) - с поддървета, съответст-ващи на формулите ((f1*f3)+(f2*f3)), ((f1*f3)-(f2*f3)), ((f1*f2)+ (f1*f3)), ((f1*f2)-(f1*f3)), ((f1/f3)+ (f2/f3)) и ((f1/f3)-(f2/f3)) съответно.

Програма Zad150.cpp решава условие а) на задачата.

```
Program Zad150.cpp
#include <iostream.h>
#include "Tree.cpp"
typedef tree<char> CharTree;
void Reduce(CharTree &t)
{char c=t.RootTree();
 if(c=='+'||c=='-'||c=='*'||c=='/')
 {CharTree t1, t2;
  t1 = t.LeftTree();
  t2 = t.RightTree();
  Reduce(t1);
  Reduce(t2);
  t.Create3(c,t1,t2);  // very important
  if(c=='+' && t1.RootTree()=='0' ||
     c=='*' && t1.RootTree()=='1')
  t=t2;
  else
  if((c=='*'||c=='/') && t2.RootTree()=='1' ||
     (c=='+'||c=='-') && t2.RootTree()=='0')
  t=t1;
  else
  if(c=='*' && (t1.RootTree()=='0'||t2.RootTree()=='0') ||
     c=='/' && t1.RootTree()=='0')
  {CharTree t3;
   t.Create3('0', t3, t3);
  }
 }
}
void main()
{CharTree t;
 t.Create();
 Reduce(t);
 t.print();
}
```

Задача 151. Да се напише булева функция, която проверява дали елемент се съдържа в двоично дърво.

```
template <class T>
```

```
bool member(T a, tree<T> const&t)
{if(t.empty())return false;
 if(a==t.RootTree())return true;
 return member(a, t.LeftTree()) || member(a, t.RightTree());
}
```

Това решение е неефективно, тъй като ако елемент не съдържа в дървото се налага обхождане на дървото с пълно изчерпване. Операциите включване и изключване на връх не се реализират добре за обикновените дървета. По-удобно за редица цели е използването на т. нар. **двоично наредени дървета** или **двоични справочници.**

4. Двоично наредено дърво

Предполагаме, че за елементите от тип T е установена наредба.

Дефиниция: Празното двоично дърво е двоично наредено дърво. Непразно двоично дърво, върховете на лявото поддърво на което са по-малки от корена, върховете на дясното поддърво са по-големи от корена и лявото, и дясното поддърво са двоично наредени дървета, се нарича **двоично наредено дърво от тип T.**

Нека t е двоично наредено дърво от тип T. *Включването на елемента a от тип T в t* се осъществява по следния начин:

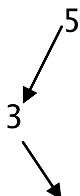
- ако t е празното двоично дърво, новото двоично наредено дърво е с корен елемента a и празни ляво и дясно поддървета.

- ако t не е празно и a е по-малко от корена му, елементът a се включва в лявото поддърво на t,

- ако t не е празно и a е не по-малко от корена му, елементът a се включва в дясното поддърво на t.

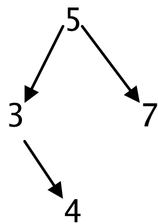
Пример: Ако в двоично нареденото дърво с корен 5 и празни поддървета се включи 3, ще се получи:

Ако към полученото двоично наредено дърво се включи 4, ще се получи:



4

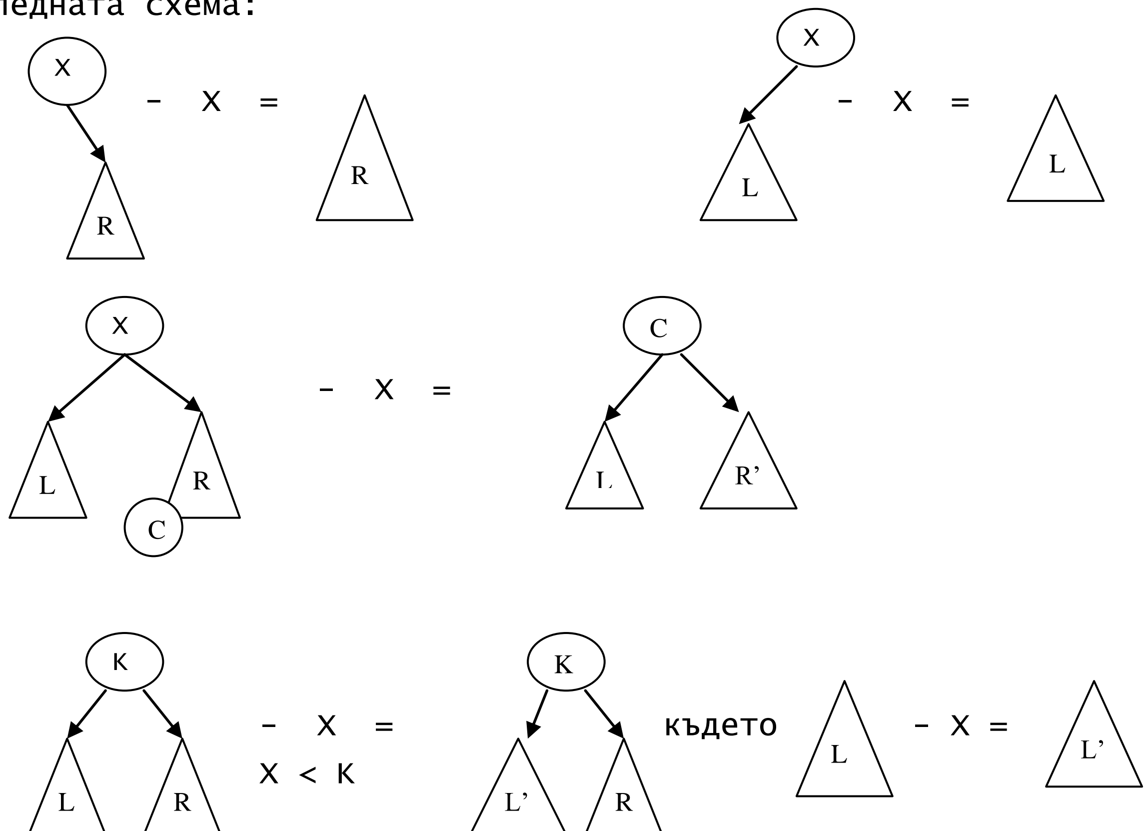
Ако към полученото двоично наредено дърво се включи 7, ще се получи:

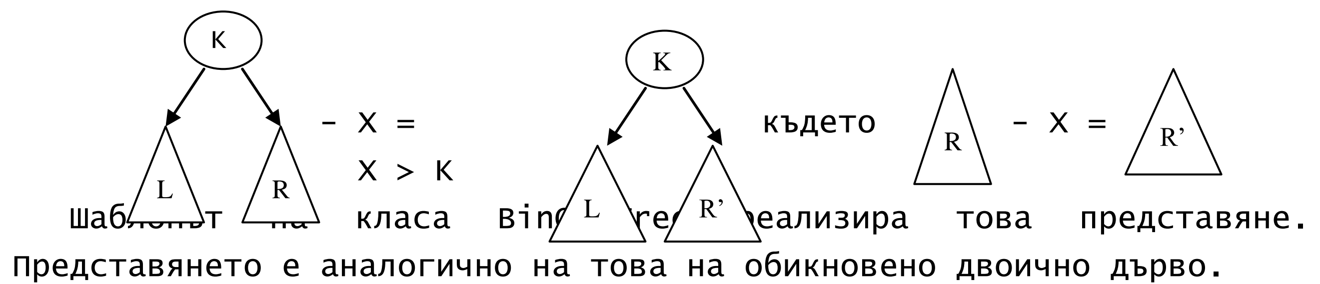


Включваният елемент "се спуска" по двоичното дърво, започвайки от корена и се насочва към лявото или дясното поддърво в зависимост от стойността си, докато намери празно място, което да заеме. Този начин на включване на елемент в двоично наредено дърво ще използваме за създаване на такива дървета.

Създаденото по този начин двоично наредено дърво притежава следното свойство: Обхождането на такова дърво по метода ЛКД сортира във възходящ ред елементите от върховете на дървото, а обхождането му по метода ДКЛ сортира в низходящ ред елементите от върховете на дървото.

Изтриването на елемент от двоично наредено дърво се осъществява по следната схема:





Шаблонът на класа BinOrdTree реализира това представяне. Представянето е аналогично на това на обикновено двоично дърво.

```
template <class T>
struct node
{T inf;
 node *Left;
 node *Right;
};
template <class T>
class BinOrdTree
{public:
 BinOrdTree();
 ~BinOrdTree();
 BinOrdTree(BinOrdTree const&);
 BinOrdTree& operator=(BinOrdTree const&);
 bool empty()const;
 T RootTree() const;
 BinOrdTree LeftTree()const;
 BinOrdTree RightTree()const;
 void print()const
 {pr(root);
  cout << endl;
 }
 void AddNode(T const & x)
 {Add(root, x);
 }
 BinOrdTree DeleteNode(T const&);
 void Create3(T, BinOrdTree, BinOrdTree);
 void Create();
 void MinTree(T &, BinOrdTree &);
 private:
  node<T> *root;
  void DeleteTree(node<T>* &)const;
```

```
  void Copy(node<T> * &, node<T>* const&)const;
  void CopyTree(BinOrdTree const&);
  void pr(const node<T> *) const;
  void Add(node<T> *&, T const &)const;
 };
```

Голямата четворка е реализирана по същия начин като при обикновените двоични дървета.

```
template <class T>
BinOrdTree<T>::BinOrdTree()
{root = NULL;
}
template <class T>
BinOrdTree<T>::~BinOrdTree()
{DeleteTree(root);
}
template <class T>
BinOrdTree<T>::BinOrdTree(BinOrdTree<T> const& r)
{CopyTree(r);
}
template <class T>
BinOrdTree<T>& BinOrdTree<T>::operator=(BinOrdTree<T> const& r)
{if(this!=&r)
 {DeleteTree(root);
  CopyTree(r);
 }
 return *this;
}
template <class T>
void BinOrdTree<T>::DeleteTree(node<T>* &p)const
{if(p)
 {DeleteTree(p->Left);
  DeleteTree(p->Right);
  delete p;
  p=NULL;
 }
}
template <class T>
void BinOrdTree<T>::CopyTree(BinOrdTree<T> const& r)
{Copy(root, r.root);
```

```
}
template <class T>
void BinOrdTree<T>::Copy(node<T> * &pos, node<T>* const &r)const
{pos=NULL;
 if(r)
 {pos = new node<T>;
  pos->inf = r->inf;
  Copy(pos->Left, r->Left);
  Copy(pos->Right, r->Right);
 }
}
```

Член-функциите empty, RootTree, LeftTree, RightTree, pr и print също са като тези на обикновените двоични дървета.

```
template <class T>
bool BinOrdTree<T>::empty()const
{return root==NULL;
}
template <class T>
T BinOrdTree<T>::RootTree()const
{return root->inf;
}
template <class T>
BinOrdTree<T> BinOrdTree<T>::LeftTree()const
{BinOrdTree<T> t;
 Copy(t.root, root->Left );
 return t;
}
template <class T>
BinOrdTree<T> BinOrdTree<T>::RightTree()const
{BinOrdTree<T> t;
 Copy(t.root, root->Right);
 return t;
}
template <class T>
void BinOrdTree<T>::pr(const node<T>*p) const
{if(p)
 {pr(p->Left);
  cout << p->inf << " ";
  pr(p->Right);
```

```
  }
 }
```

Член-функцията Add е помощна. Използва се за реализиране на член-функцията AddNode, чрез която се включва елемент в двоично наредено дърво по описания по-горе начин.

```
template <class T>
void BinOrdTree<T>::Add(node<T>* &p, T const & x)const
{if(!p)
 {p = new node<T>;
  p->inf = x;
  p->Left = NULL;
  p->Right = NULL;
 }
 else
 if(x < p->inf)Add(p->Left, x);
 else Add(p->Right, x);
}
```

Създаването на двоично наредено дърво се осъществява чрез член-функцията Create. Тя реализира въвеждане на елементи и включването им чрез AddNode в двоично наредено дърво.

```
template <class T>
void BinOrdTree<T>::Create()
{root = NULL;
 T x; char c;
 do
 {cout << "> ";
  cin >> x;
  AddNode(x);
  cout << "next elem  y/n? "; cin >> c;
  }while(c=='y');
}
```

Член-функцията Create3 е същата като тази при обикновените двоични дървета.

```
template <class T>
void BinOrdTree<T>::Create3(T x, BinOrdTree<T> l, BinOrdTree<T>
r)
{root = new node<T>;
 root->inf = x;
```

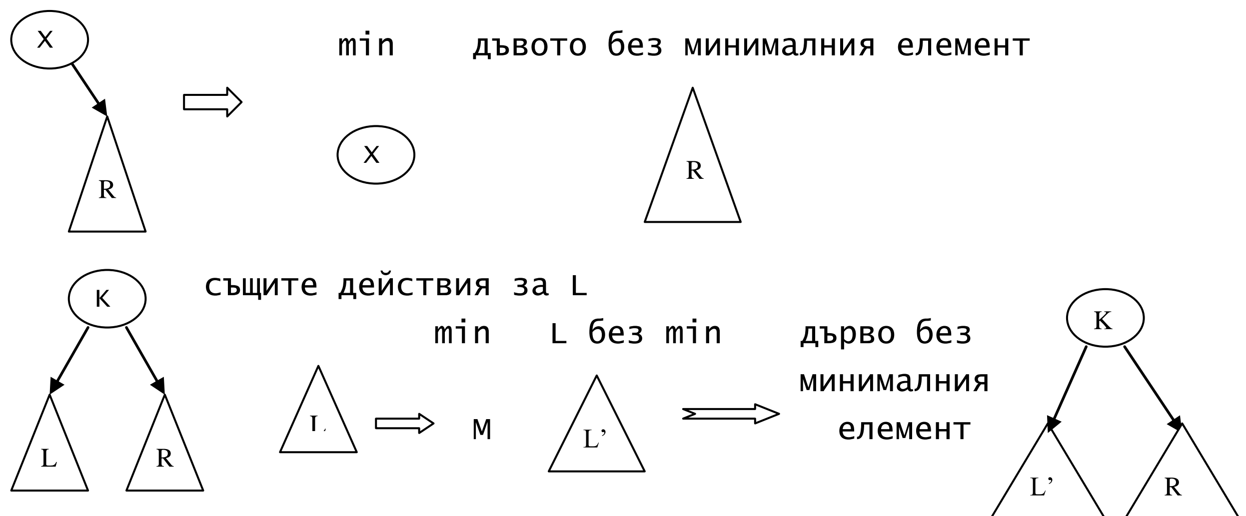
```
 Copy(root->Left, l.root);
 Copy(root->Right, r.root);
}
```

Член-функцията MinTree намира минималния елемент на подразбиращото се двоично наредено дърво и подразбиращото се двоично наредено дърво без минималния му елемент. Реализира следния алгоритъм.

- Ако лявото поддърво на подразбиращото се двоично нареденото дърво е празно, минималният му елемент е корена, а дървото без минималния елемент е дясното му поддърво.

- Ако лявото му поддърво е непразно, на това двоично наредено дърво се намира минималния елемент и дървото без минималния елемент. След това се конструира двоично нареденото от корена, лявото поддърво без минималния му елемент и дясното поддърво на подразбиращото се дърво.

Тези действия могат да се опишат графично така:




```
template <class T>
void BinOrdTree<T>::MinTree(T &x, BinOrdTree<T> &mint)
{T a=RootTree();
 if(LeftTree().empty())
 {x = a;
  mint = RightTree();
 }
 else
 {BinOrdTree<T> t, t1;
  t=*this;
```

```
  *this = LeftTree();
  MinTree(x, t1);
  mint.Create3(a, t1, t.RightTree());
 }
}
```

Член-функцията DeleteNode изтрива връх на подразбиращото се двоично наредено дърво. Реализира описания по-горе алгоритъм. Изключването трябва да се предшества от проверка дали изключваният елемент принадлежи на двоично нареденото дърво.

```
template <class T>
BinOrdTree<T> BinOrdTree<T>::DeleteNode(T const& x)
{T a=RootTree();
 if(a==x && LeftTree().empty())
 return RightTree();
 if(a==x && RightTree().empty())
 return LeftTree();
 if(a==x)
 {T c;
  BinOrdTree<T> tmin;
  RightTree().MinTree(c, tmin);
  Create3(c, LeftTree(),tmin);
  return *this;
 }
 if(x<a)
 {Create3(a, LeftTree().DeleteNode(x), RightTree());
  return *this;
 }
 Create3(a, LeftTree(), RightTree().DeleteNode(x));
 return *this;
}
```

Записваме този шаблон във файла BinOrdTree.cpp.

Задача 152. Да се напише програма, която въвежда редица от цели числа, сортира ги във възходящ ред, след което изключва въведен еленет като запазва наредбата на елементите.

```
#include <iostream.h>
#include "BinOrdTree.cpp"
```

```
typedef BinOrdTree<int> IntTree;
void main()
{IntTree t, t1;
 t.Create();
 t.print();
 int x;
 cout << "x: "; cin >> x;
 t=t.DeleteNode(x);
 t.print();
}
```

Задачи

Задача 1. Да се дефинира процедура, която извежда отначало всички положителни, а след това всички отрицателни върхове на двоично дърво от тип int.

Задача 2. Дадено е двоично дърво от тип int. Да се напише функция, която определя има ли сред върховете на двоичното дърво поне два върха с равни стойности.

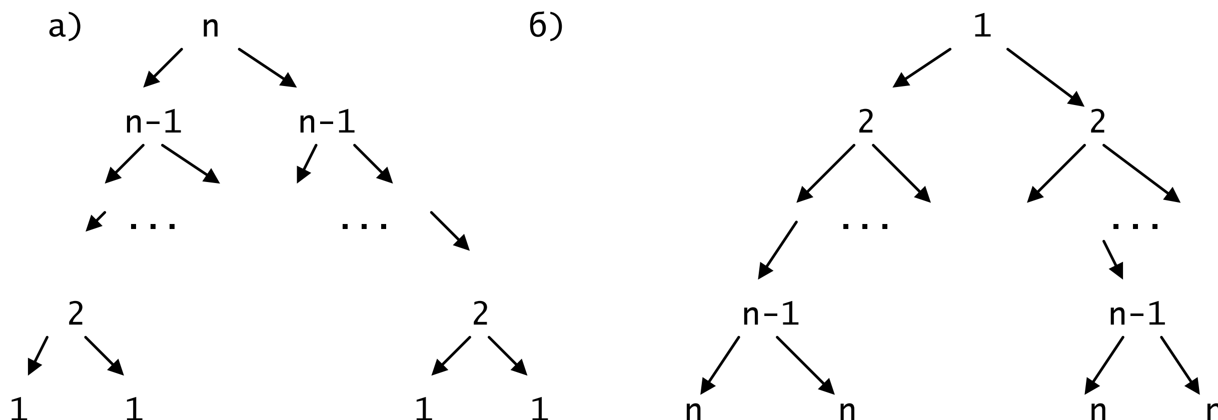
Задача 3. Дадено е двоично дърво от тип int. Да се напише функция, която намира сумата на четните елементите от върховете на двоичното дърво.

Задача 4. Да се напише функция или процедура, която:

а) определя броя на включванията на елемента a в двоичното дърво d;

б) намира броя на върховете на n-то ниво на непразното двоично дърво d (коренът се счита за връх от 0-во ниво).

Задача 5. Да се напише процедура CreateBinTree(d, n), където n е положително цяло число, която създава следното двоично дърво:



Задача 6. Да се напише програма, която за даден аритметичен израз от вида:

<АИ> ::= <терминал>|
(<АИ> <знак> <АИ>);
<знак> ::= +|-|*|/;
<терминал> ::= 0|1|...|9,

намира и извежда правия полски запис, който съответства на израза.

Задача 7. Да се напише програма, която за даден аритметичен израз от вида:

<АИ> ::= <терминал>|
(<АИ> <знак> <АИ>);
<знак> ::= +|-|*|/;
<терминал> ::= 0|1|...|9,

намира и извежда обратния полски запис, който съответства на израза.

Задача 8. Даден е свързан списък, съдържащ реални числа. Да се напише програма, която от елементите на списъка генерира двоично наредено дърво.

Задача 9. Даден е стек от реални числа. Да се напише програма, която сортира елементите на стека като за целта използва двоично наредено дърво.

Задача 10. Да се напише програма, която установява дали съществува път между два върха на дадено двоично дърво. Ако съществува път, да се намери пътя, а също и дължината му.

Допълнителна литература

5. B. Stroustrup, C++ Programming Language. Third Edition, Addison – Wesley, 1997.
6. А. Берстисс, Структуры данных, М. Статистика, 1974.
7. Ст. Липман, Езикът С++ в примери, “КОЛХИДА ТРЕЙД” КООП, С. 1993.
8. Л. Амерал, Алгоритми и структури от данни в С++, С., ИК СОФТЕХ, 2001.
9. М. Тодорова, Програмиране на Паскал, С., Полипринт, 1993.