Увод в програмирането

alpha

Generated by Doxygen 1.8.5

Tue Dec 3 2013 22:33:47

Съдържание

Глава 1

Main Page

Увод в програмирането

1.1 Здравей!

Тук можете да намерите материали по дисциплина Увод в програмирането (УП).

1.2 Грешки, неточности и въпроси

Материалите тук са изготвени от Благовест Тренчев, студент във ФМИ на СУ, през есента на 2013г.

Ако откриете грешки и/или неточности или просто имате въпрос, моля, свържете се с мен на bl.trenchev@gmail.com

Глава 2

Алгоритми - определение, примери, свойства, начини на изразяване (словесно, блок-схеми, алгоритмични езици)

Компютър - техническо средство за изпълнение на алгоритми

Алгоритъм - разглежда най-общо представлява списък от краен брой правила за извършване на дадени действия в определен ред, чрез които се решава всяка една задача от даден тип, клас

Пример:

Списък с числа: a_1 , a_2 , a_3 , a_4 .... a_k ... a_{n-1} , a_n

Търсим: alpha - минималното число; beta - максималното число

Съставяме алгоритъм:

1. Въвеждаме n. Въвеждаме n числа a_1 , a_2 ... a_n . Изпълни правило 2.
2. Под alpha запомни a_1 . Изпълни правило 3
3. Под beta запомни a_2 . Изпълни правило 4
4. Под k запомни 2. Изпълни правило 5.
5. Ако a_k < alpha изпълни правило 6. иначе изпълни правило 7.
6. Под alpha запомни a_k . изпълни правило 9.
7. Ако a_k > beta изпълни правило 8 иначе изпълни правило правило 9
8. Под beta запомни a_k . Изпълни правило 9.
9. Увеличи k с едно. изпълни правило 10.
10. Ако k <= n изпълни правило 5. иначе правило 11.
11. Изведи alpha и beta. Изпълни правило 12.
12. Прекрати изпълнението на алгоритъма.

Пример: n = 5 a_1 = 17 a_2 = 58 a_3 = 3 a_4 = 12 a_5 = 27

Конкретна задача се определя от набор от допустими данни.

Правило - има си номер, действие, което се изпълнява, и наследник, който посочва как се продължава алгоритъма.

Действията могат да бъдат:

- Условни
  - имат 2 наследника (ако условнието е изпълнено и ако не).
- Безусловни
  - имат 1 наследник.

Изпълнение на 1 правило се нарича стъпка от работата на алгоритъма.

Свойства на алгоритмите:

- определеност
  - описанието на алгоритъма е ясно, разбираемо и определя еднозначно действията, които трябва да се извършат
- масовост
  - клас задачи, обработка на съвкупоност от набор от допустими данни
- резултатност
  - всяко изпълнение завършва за крайно време, крайното време е следствие от удовлетворението на две условия:
    - всяко изпълнение се състои от краен брой стъпки
    - всяка стъпка се изпълнява за крайно време
  - Забележка: от големината на крайното време зависи дали алгоритъмът е практически приложим
- цикличност
  - алгоритмите могат да се описват така, че групи от правила да се изпълняват многократно

Начини за изразяване:

- чрез блоксхеми
- описание на машинен език
  - последователност от команди
- алгоритмични (програмни) езици
  - C/C++ , C# , Java , VisualBasic
  - Python, Perl, Clojure
  - и други

```
#include <iostream.h>
void main()
{
	int n,k,i;
	float alpha, beta, a[101];
	cin>>n;
	for( i=1; i<=n; i++)
		cin>>a[i];
	alpha = a[1];
	beta = a[1];
	for(k=2; k<=n; k++)
		if ( a[k] < alpha )
			alpha = a[k];
		else if ( a[k] > beta )
			beta = a[k];
	cout<<alpha<<"  "<<beta<<endl;
}
```

Глава 3

Променливи и типове данни

Разглеждаме следната задача:

Да се състваи програма за намиране на лицето и периметъра на квадрат със страна a (цяло число).

Решение:

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

Служебни думи - всеки език използва определена съвкупност от думи. Тези думи се наричат служебни. В примера по горе служебни думи са: cout, int, cin, include, void.

"Азбука" на езика / символи, които могат да участват в кода на една програма /:

- букви - главни и малки латински
- символът за подчертаване: "_"
- цифри 0 до 9 включително
- празни символи
  - интервал, табулация, нов ред
- специални - #, $, >, < и д.р.

Идентификатор - последователност от латински букви и цифри, която започва с латинска буква. Служат за именуване на различни неща в програмата пример променливи подпрограмни типове, записи. Тесе измислят от програмиста в една и съща част не бива да има дублиране на идентификатори. В езика няма ограничение за дължина, но компилаторът има. Част от идентификаторите са запазени за синтактични единици в езика и се наричат ключови думи.

Note

Важни бележки:
- малките и главни букви се различават
- ключовите думи се състоят от малки букви (case, break, catch и д.р.)

Някои идентификатори са стандартни думи. (Това са идентификатори, които могат да се използват в програмите по принцип)

Променлива - има име и стойност. Името е идентификатор. Стойността се съхранява в 1 или няколко последователни байта от оперативната памет, която е разпределена за тази променлива. Всяка променлива в C/C++ преди да се използва трябва да се дефинира. С дефинирането на променливата за нея се разпределя памет в съответствие с типа ѝ. С това се определя диапазон на допустими стойности.

Общ вид за дефиниране на променлива:

```
тип_на_променливата списък_от_имена_на_променливи;

int counter;
```

Типът се задава с една или няколко думи:

тип	байтове
signed char	1
unsigned char	1
char	1
short int	2 ∗
int	2 или 4 ∗
unsigned int	2 или 4 ∗
long int	4 или 8
unsigned long int	4 или 8
float	4
double	8
long double	10

∗ в зависимост от компилатора.

Стойността на char е 1 символ. Кодът на символа се записва в разпределената памет. Използва се ASCII код. Ако кодът се използва като цяло число:

- unsigned от 0 до 255
- signed от -128 до 127

Изпозлването на различни начини за представяне е свързано с работата на централния процесор. Целта е алгоритмите за събиране и изваждане да са възможно най-прости.

char може да бъде signed или unsigned

Типът определя размера на зададената памет и начина за кодиране, т.е. допустимите стойности.

Тип	От	До
int	-32700	32700
unsigned int	0	65500
long int	-2 147 000 000	2 147 000 000
unsigned long int	0	4 294 000 000

Note

Стойностите са приближени!

Глава 4

Константи (литерали). Видове.

Нека съставим програма за пресмятане лицето на кръх и дължината на окръжност с даден радиус r - реално число:

```
#include<iostream>
using namespace std;
int main()
{
	float r, p, s;
	cout<<"Въведи r=";
	cin>>r;
	p = 2*3.14*r;
	s = 3.14*r*r;
	cout<<"Дължината на окръжността е "<<p<<endl;
	cout<<"Лицето на кръга е "<<s<<endl;
	return 0;
}
```

Костантите могат да бъдат няколко вида:

- числови. Те могат да бъдат:
  - цели:
    - десетични (записани са в десетична бройна система)
      - Пример: 356, -78, +95
    - осмични (записани са в осмична бройна система)
      - Пример: 0356
      - цели константи започващи с 0. Нямат знак т.е. не могат да бъдат отрицателни.
    - шестнайсетични (записани са шестнайсетична бройна система)
      - Пример: 0xA10
      - започват с 0x
- символи
  - типичен пример са буквите
- низови
  - типичен пример: "baba"

В разглежданата програма костанти са: 2 и 3.14.

Възможное да укажем, че някоя константа искаме да се отчита като точно определен тип променлива: int - 356 long int - 356L unsigned int - 356U

Глава 5

Коментари, структура на програмата и етапи на нейната обработка

Коментар

- пояснение към програмата
  - многоредов коментар
    - в C и C++ започва с "/∗" и свършва с "

Глава 6

Операции и изрази

Разглеждаме програма, която по дадена хипотенуза и разлика между катетите, намира катетите.

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

Израз наричаме всяка валидна за езика комбинация от операции, операнди и кръгли скоби. Операндите могат да бъдат константи, променливи, обръщения към функции и в общия случай изрази. Под валидна комбинация разбираме комбинация, която не противоречи на зададените в езика съответствия между операции и операнди и на синтаксиса на езика.

В зависимост от броя на операндите различаваме:

- едноместни (унарни) операции - 1 операнд
- двуместни (бинарни) операции - 2 операнда
- триместни (тернарни) операции - 3 операнда

двуместни +, - и ∗ Типът на резултата от една операция зависи от типа на операндите. Ако типът на двата операнда е цял, то резултатът е цял. Ако поне един от операндите е с плаваща точка, то резултатът е с плаваща точка. Ако и на двата операнда са цели, то при делене резултатът е целочислено.

% -> остатък от целочислено деление

едноместни + , -

В C/C++ истина (true) е всяка стойност различна от 0 и лъжа (false) е всяка стойност равна на 0. Възприето е стойността на резултата при операции за отношения и логически операции да е винаги или 1 (за true), или 0 (за false). Операндите за отношения се използват за сравняване на стойности: >, >= , <, <=, ==, !=

Логически опрации са:

1. && и and логическо умножение
2. || или or логическо събиране
3. не not логическо отрицание

При логическо събиране и умножение някой компилатори правят оптимизация операнд 1 && операнд 2 операнд 1 || операнд 2 понякога трябва да се избере опция да не се оптимизира

Присвояване! Присвояването е операция променлива = израз

- връща резултат Пресмята се изразът, типът на стойността на израза се преобразува към типа на променлива, стойността става стойност на променливата. Връща стойността на израза Присвояването на променливата е страничен ефект. x = a++ ; x = y = z приосвояването става израз x = y = 100 е израз x = y = 100; оператор

Съкратен запис на комбинация от операции: преработи в табличка a = a+b a+=b a = a-b a-=b a = a∗b a∗=b a = a/b a/=b a = ab a%=b a = a+1 ++a a++ a = a-1 –a a– еквиваленти, ако се ползват автономно

При поставянето на двата + или - преди променливата, първо се изпълнява операцията и след това променливата се използва с новата си стойност.

int x, y; x = 15; еквивалентно на: x=x+1; y = ++x; y=x;

x = 15; y = x; y = x++; еквивалентно на x = x + 1;

++ или – след променлива - първо се използва променливата със старата си стойност и след това се изпълнява операцията.

операция за последователно изпълнение операнд 1, операнд 2 резултатът е стойността на операнд 2 (стойността на последния израз)

- явно преобразувание на типовете (тип) операнд използва се за явно преобразуване на типа на операнда към посочения в кръглите скоби тип

int suma, broj; float sr_uspeh; sr_uspeh = (float) suma/broj;

операция за размер на обект sizeof(операнд) резултатът е размерът на операнда в B. операндът е променлива или тип sizeof(sr_uspeh) cout<<sizeof(int)<<endl;

Редът на извикване на операндите в един израз се определя от техния приоритет и асоциативност. По-висок приоритет преди по-нисък. Операциите с един и същ приоритет се изпълняват отляво надясно или отдясно наляво в зависимост от техните асоциативност.

Глава 7

Преобразуване на типовете

Когато в един израз има операнди от смесени типове, то те се преобразуват автоматично към един тип. Общ принцип е, че това се извършва към по-старши тип:

long double 10 double 8 float 4 unsigned long int 4 long int 4 unsigned int 2 int 2 unsigned char 1 signed char 1

Отчита се колко байта се отделят. При еднакъв брой се гледа кой тип съдържа по-големи стойности. Update-ни нещата към табличка =)

операнд 1 операция 2 операнд 2

- преобразува се към старшия тип при всички операции без призсоваване променлива = израз
  - типът на изразът се преобразува към типа на променливата

Автоматично char се преобразува към Int. Възможно е, когато се преобразува типа да се променя и стойността на операнда. Примери: Int -> unsigned int или signed long int

При преобразуване операнда като последователност от битове не се променя, променя се интерпретацията, unsigned int -> знаковия бит става част от стойността unsigned long int -> добавят се два старши байта, в които се размножава знаковия бит int -> long int стойността не се променя знак -> без знак променя се int -> double, float, long double стойността не се променя

старши -> младши тип

Глава 8

Условни изрази и условни оператори

Условни изрази - операцията за условен израз, единствената триместна операция операнд1? операнд2 : операнд3 Пресмята се операнд 1, ако стойността му е true (!= 0), резултатът от изпълнение на операцията е стойността на операнд 2; ако операнд 1 има стойност 0, то се пресмята операнд 3 и неговата стойност е резултат от изпълнение на операцията. Ако операнд 2 и операнд 3 са от различни типове, резултатът се преобразува автоматично към по-старшия тип. x>y?10:100

Програма за пресмятане на заплащането за една седмица при почасово заплащане. Имаме R часа работа при заплащане norma лв на час. А извънредния труд се заплаща с 50% по-скъпо (над 40 часа)

стигаме до извода: suma = R∗norma, ако R<=40 = 40∗norma + 1,5 ∗ (R - 40) ∗ norma, ако R>40

```
#include<iostream.h>
#include<iomanip>
void main()
{
        float R, norma, suma;
        int cod;
        cout<<"\n Въведете cod, R и norma: ";
        cin>>cod>>h>>norma;
        if( R <= 40 ) suma = h * norma;
        else suma = 40*norma + 1,5*(R-40)*norma;
        cout<<setw(6)<<cod<<setw(10)<<setprecision(2)<<setiosflags(ios::fixed/ios::showpoint)<<suma<<endl;
}
```

NB! Бележки по програмата! A и R трябва да е h....

Можем да заменим конструкцията if else с условния: suma = (R <= 40)?R∗norma:40∗norma+1,5∗(R-40)∗norma; Условния израз се компилира по-ефективно.

Оператор if: if (израз) оператор 1 [else оператор 2]

- без else - кратка форма
- с else - пълна форма

Глава 9

Оператори за цикъл while и do while

Чрез операторите за цикъл се описва многократно изпълнение на 1 или няколко последователни оператори, които се наричат тяло на цикъла. Действия могат да се извършват преди или след изпълнението на цикъла.

Принципната разлика между операторите while и do while е в това къде се проверява условието за крайна цикъла. При while проверката се извършва в началото на цикъла, а при do while в края. От това следва, че тялото на цикъла в оператор do while винаги се изпълнява поне веднъж. While може да не се изпълни нито веднъж.

Разлика в синтатксиса:

while - след while (израз) няма ; (точка и запетая)

do while - има ;

NB! Оператор може да е цикъл (влагане на цикъл в цикъла)

while (израз) /whitespace/ -> изразът е празен

while (израз) {} -> the same

Компилаторът разглежда while без ; като начало на while, а със ; - край на do while;

Всички променливи в израза трябва да имат стойност:

дали при следващо изпълнение някоя от променливите се променя ако не се промененя - безкраен цикъл (дали поне една от променливите се изменя) -> зациклене умишлено безкраен чрез условие - в оператор прекъсване break

изразът никога не получава стойност false

Пример:

От клавиатурата се въвежда текст. Да се определи колко пъти даден символ се среща в текста.

```
#include <iostream>
#include <stdio.h> // <cstdio>

using namespace std;

int main()
{
	int R, i;
	char symbol;
	cout<<"Въведете символ: ";
	symbol = getchar();
	getchar(); // за да работи правилно трябва да се махне символа от входящия буфер.
	cout<<"Въведете текста: \n";
	i = 0;
	while( ( R = getchar() ) != '\n' )
		if ( R == symbol )
			i++;
	cout<<"Честотата на символа в текста = "<<i<<endl;
```

```
        return 0;
}
```

getchar() - връща int

Да се намери максимума и квадрата на максимума на двойка числа, въвеждана с потвърждаване.

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

using namespace std;

int main()
{
        short n1, n2, max, R;
        long max2;

        do {
                cout<<"Въведете първото число: ";
                cin>>n1;
                cout<<"Въведете второ число: ";

                if ( n1 >= n2 ) max = n1;
                else
                        max = n2;

                cout<<"max= "<< max << endl;

                max2 = (long) max*max;
                cout<<"max2= "<<max2<<endl;

                cout<<"Ще въведете ли още числа? ( Y/N )";

                R = getchar();
                getchar();
        } while ( R == 'Y' || R=='y' );


        return 0;
}
```

Глава 10

Масиви низ от символи

10.1 Масив

Масивът е n-мерна наредена съвкупност от елементи от един и същ тип. Преди да се използва един масив той трябва да се дефинира. NB! Има ограничения в размерността!

Примери за дефиниране на масив:

```
int arr[5]; // дефинираме масив с 5 елемента тип int
```

На мястото на "5" може да стои произволна целочислена константа.

Примери: 4,10,11,12,15,100,100000 ;

Работа с елементи на масив:

```
int vect[7] = { 1, 2, 3, 4, 5, 6, 7 };
// Дефинира масив с 7 елемента като същевременно инициализираме стойностите на масива
cout<<vect[0]<<endl;
// извежда първият елемент от масива ( 1 ) на екрана последван от нов ред;

cout<<vect[1]<<endl;
// извежда първият елемент от масива ( 2 ) на екрана последван от нов ред;

cout<<vect[2]<<endl;
// извежда първият елемент от масива ( 3 ) на екрана последван от нов ред;

cout<<vect[3]<<endl;
// извежда първият елемент от масива ( 4 ) на екрана последван от нов ред;

cout<<vect[4]<<endl;
// извежда първият елемент от масива ( 5 ) на екрана последван от нов ред;

cout<<vect[5]<<endl;
// извежда първият елемент от масива ( 6 ) на екрана последван от нов ред;

cout<<vect[6]<<endl;
// извежда първият елемент от масива ( 7 ) на екрана последван от нов ред;
```

Note

При дефиниране на масив arr[N] индексите започват от 0 и свършват с N-1.

Пример:

```
int arr[5];
// Валидни индекси на елементи от масива са 0, 1, 2, 3, 4 .
```

Едномерните масиви се използват за представяне на вектори: -> Последователните елементи се представят като последователни полета в оперативната памет.

Пример: Искаме да ползваме следния вектор: a (1, 2, 3) Като код представянето изглежда така:

```
int vector_a [3] = { 1, 2, 3};
```

Представяне на матрици - двумерни масиви:

```
double mat[3][2];
```

Инициализиране на елементите на масива:

```
ляляляля
```

Обръщения към елементите на масива:

```
mat[0][0];
mat[0][1];
mat[1][0];
mat[1][1];
mat[2][0];
mat[2][1];
```

Елементите на масива се записват в посочения ред в оперативната памет.
Общ модел на достъпване на елемент:

```
mat[i][j];
```

Пример за работа с 3мерен масив:

```
float table[4][3][2];
// дефиниране на 3мерен масив;

// Достъпване на елементи:
table[0][0][0];
table[0][0][1];
table[0][1][0];
table[0][1][1];
table[0][2][0];
table[0][2][1];

table[1][0][0];
table[1][0][1];
table[1][1][0];
table[1][1][1];
table[1][2][0];
table[1][2][1];

table[2][0][0];
table[2][0][1];
table[2][1][0];
table[2][1][1];
table[2][2][0];
table[2][2][1];

table[3][0][0];
table[3][0][1];
table[3][1][0];
table[3][1][1];
table[3][2][0];
table[3][2][1];
```

Note

Компилаторите не правят проверки дали индексите са част от масива!
За компилаторът валидно обръщение е table[10][10][10] , но то води до грешка по време на изпълнение на програмата.
При работа с масиви бъдете изключително внимателни с индексите им.

Едномерен символен масив / масив от char-ове /

```
char str[12];
```

В C/C++ няма вграден тип низ от символи (string).

За това разглеждаме едномерни символни масиви като променливи от тип низ от символи.

Работа с низове:

```
char str[12];
// Грешно:
str = "abc"; // Невалидна команда!

// Правилно:
str[0] = 'a';
str[1] = 'b';
str[2] = 'c';
str[3] = '\0'; // специален символ; нулев байт; сигнализира край на низ
```

С дефинирането на един масив се определя диапазоните на индексите.

Не се прави вградена проверка дали индексът, който се използва, е в диапазоните.

Ако индексът не е в зададените диапазони се генерира адрес на поле, което е извън разпределената памет за масива.

Стойността на елемента е неопределена.

Трябва да се оцени правилно колко памет изисква масивът.

Задача:

От клавиатурата се въвеждат n цели числа и се извеждат в обратен ред:

```
#include <iostream >

using namespace std;

int main()
{
        int n,i;
        int number[20];

        cout<<"Въведете броя на числата: \n"
        cin>>n;
        i = 0;
        while( i < n)
        {
                cout<<"Въведете "<< i <<"-то число: ";
                cin>>number[i];
                i++;
        }
        cout<<"\n Числата в обратен ред: \n";
        i = n-1;
        while( i >= 0 )
        {
                i--;
                cout<<number[i]<<"  ";
        }
        return 0;

}
```

Note

String не е вграден тип в C++!

Глава 11

Оператор за цикъл for

Общ вид:

```
for ( израз1 ; израз2; израз3 )
{

}
```

For е съставен оператор:

<!- // блок схема! –>

For

Обикновено с израз1 се задават начални стойности на променливи, които участват в израз2.

Израз2 е условие за продължаване на цикъла. Това условие управлява продължителността на цикъла.

С израз3 се определя правило, по което променливи от израз2 се изменят на всяка итерация на цикъла.

Най-често израз1 и израз3 са изрази за присвояване, а израз 2 е израз за сравнение.

Всички променливи, които се използват в израз1, израз 2 и израз3, трябва да са предварително дефинирани така както изисква общото правило за дефиниране на променливи.

Най-често срещаното приложение на for са цикли с предварително известен брой итерации и използващи управляваща променлива.

Задача:

Да се сумират целите числа от n1 до n2 включително.

Решение:

```
int i, n1, n2, result;
result = 0;
n1=3;
n2=5;
for(i=n1; i<=n2; i++)
        result = result+i;
```

Note

В примера по-горе i е управляваща променлива.

Управляващата променлива на цикъла запазва своята последна стойност след завършването на цикъла и може да бъде използвана за следващи пресмятания.

В синтаксиса на for израз1, израз2 и израз3 не са задължителни. Всеки от тях, поотделно и в комбинация с другите, може да бъде пропуснат. В тези случаи разделителите ";" отнсящи се към пропуснатите изрази трябва задължително да присъстват.

Ако и трите израза са пропуснати:

```
for(;;)
    оператора;
```

Ако израз2 е пропуснат се смята, че условието е изпълнено винаги. Така се създава безкраен цикъл. За да се прекъсне подобен цикъл трябва да използваме команда break.

```
for( ; израз2 ; )
    оператор
```

Изразът по-горе е равносилен на:

```
while(израз2)
    оператор
```

Задача:

Да се намери първият елемент равен на 0 в даден масив.

```
int count, a[100];
for( count = 0; a[count]!=0 ; count++ );
// for ( cout = 0; a[count++] != 0; ); равносилно на предния ред;
```

Задача:

Намерете сумата на елементите на масива.

Решение:

```
int suma, count, a[100];

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

Глава 12

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

Оператор switch

switch - избира се една от няколко взаимноизключващи се възможности, алтернативи:

```
switch ( израз_селектор )
{
        case израз1:
                оператори
                [break;]

        case израз2:
                оператори
                [break;]
        .............
        [ default:
                оператори
                [break;]]
}
```

израз_селектор е условие за избот. Стойността му е от цял тип (частност тип char)

израз1, израз2, ... са константни изрази от цял тип (в частност тип char) като етикет към първия/следващи оператори.

switch, case, default са ключови думи.

Начин на работа на switch:

switch се изпълнява по следния начин:

1. Първо се пресмята стойността на израза селектор.
2. Стойността на израза селектор се сравнява със стойността на констрантните изрази1, изрази2, ...
3. Ако стойността на израза селектор съвпада със стойността на някой от константните изрази, то се изпълнява групата от оператори, преписана към съответния израз
4. С изпълнение на break управлението се предава на оператори, записани след затварящата фигурна скоба на switch
5. Ако няма съвпадение с нито един от константните изрази, изпълнява се групата от оператори след default

При състяване на switch трябва да се спазват следните изисквания:

1. не може да има константни изрази с еднаква стойност

2. последователността на подредба на константните изрази може да е напълно произволна
3. в какъв ред се пресмятат от компилатора стойностите на изразите зависи от компилатора
4. няколко етикета (изрази) могат да бъдат свързани с една и съща група оператори:

```
case израз1:
case израз2:
.....
case изразK:
        оператори
```

5. операторите break не са задължителни

Ако не се използва break, след изпълнението на избрания вариант веднага се преминава към изпълнението на непосредсвено следващия вариант и т.н. Ако няма нито един break , изпълнението на switch приключва със '}' .

Default не е задължителен. Ако такъв вариант не е предвиден и няма съвпадение на стойност на израза селектор с някой израз, то switch-а е равносилен на празен оператор. Т.е. switch не прави нищо.

Операторите между {} образуват по същество блок. Ето защо в блока може да има дефиниране на променливи.

Изход от switch може да се осъществи освен с оператор switch и с операторите goto, return и exit.

switch може да бъде влаган в себе си.

Операторът switch може да се счита за еквивалентен на следната вложена конструкция:

```
if ( израз_селектор == израз1 )
{
        оператори;
}
else if ( изрази_селектор == израз2 )
{
        оператори;
}
// Последния блок може и да не се ползва:
else
{
        оператори;
}
```

switch е за предпочитане, тъй като води до по-лесни програми.

Предимството на вложената конструкция ifelse е, че определя точно реда, по който се оценяват израз1, израз2 и т.н. Това позволява най-често срещания случай да се постави на първо място и с това да се избегне проверките на осталинте изрази като по този начин се повиши ефективността на програмния код. Очевидно if lese е по-обща от switch.

Задача: Да се състави програма, която пресмята периметъра на окръжност, квадрат или равностранен триъгълник:

Решение:

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

void main()
{
        char sys;
        float r, x;
        cout<<"Въведете кода на фигурата: ";
        sys = getchar();
        cout<<endl<<"Въведете r= ";
        cin>>r;
        cout<<endl;
```

```
        switch(sys)
        {
                case 'c':
                        x = 2 * 3.14 * r;
                        break;

                case 's':
                        x = 4 * r;
                        break;

                case 't':
                        x = 3 * r;
                        break;
                default:
                        cout<<"Въведен е неправилен код на фигура!"<<endl;
                        exit(1);
        }
        cout<<"Периметърът е "<< x << endl;
}
```

Note

exit() -> завършва изпълнението на програмата:
exit(1) -> програмата не е изпълнена успешно; сигналът се предава на Операционната Система
exit(0) -> програмата е изпълнена успешно; сигналът се предава на Операционната система.

break

Командата break се използва в тялото на цикли.

Общ вид:

```
break;
```

break може да бъде използван на произволно място в тялото на цикъла, което е допустимо, за който и да е друг оператор.

Изпълнението на break предизвиква излизането от цикъла и предаването на управлението на операторите, записани непосредствено след цикъла. След излизането от цъкъла с оператор break стойностите на променливите от цикъла не се изменят като се запазват такива, каквито са били преди изпълнението на break.

Задача: Да се състави програма за търсене на първо срещане на даден символ в текст

Решение:

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

using namespace std;

int main ()
{
        int i, j, R;
        char symbol;
        cout<<"Въведете символа: "
        symbol = getchar() ; // чете символа
        getchar(); // "доизчиства" буфера
        cout<<"Въведете текста: \n";
        i=1;
        j=0;
        while ( ( R = getchar() ) != '\n' )
        {
                if ( R == symbol )
                {
                        j = 1;
                        break;
                }
                else
```

```
                i++;
        }

        if ( j == 1 )
                cout<<"Символът "<<symbol<<" е на " << i <<"-та позиция в текста."<<endl;
        else
                cout<<"Символът "<<symbol<<" не е открит в текста."<<endl;

        return 0;
}
```

Continue

Оператор continue се използва за завършване на итерация на цикъл. Той може да бъде записан на произволно място в тялото на цикъла, разрешено за оператор.

Оператор continue завършва изпълнението на текущата итерация в цикъла. Изпълнението на цикъла продължава по стандартния начин, определен от дефиницията на цикъла.

Задача:

Да се напише програма, която намира индексите на елементите на масив, който съдържат четни числа.

Решение:

```
#include<iostream.h>
void main()
{
        int i, a[10];

        for( i = 0; i < 10 ; i++ )
        {
                cout<<"Въведете a["<<i<<"] = ";
                cin>>a[i];
        }

        cout<<"\n Индекси на четните числа в масива са: \n"

        for( i = 0; i<10; i++)
        {
                if( a[i] % 2 != 0 )
                        continue;
                cout<<i<<"  ";
        }
}
```

Оператор Goto

Операторът осъществява безусловен преход.

общ вид:

```
етикет: оператор;
.................
goto етикет;
```

Препоръчително е goto да не се използва. В повечето съвременни езици е отхвърлен и не се ползва.

goto предава управлението на програмата в тялото на една (и съща) функция.

Ако goto извън цикъл предава управлението на програмата в цикъл, не е ясно какво става.

Обичайно приложение:

Goto се ползва във множество вложени цикли като goto от най-вътрешния предава управлението извън най-външния цикъл.

Глава 13

Функции - общ вид. Оператор return

Дефиниция на функция

```
[тип_на_резултатата] име_на_функцията ( списък от тип_и_име_на_формалните_параметри )
{
    последователност_от_дефиниции_на_вътрешни_променливи_и_оператори
}
```

Note

име_на_функцията и име_на_формалните_параметри се заместват от идентификатори.

Полето тип_на_резултата задава типа на връщаната стойност от функцията. Ако това поле отсъства, се предполага, че функцията връша стойност от тип int. В последните версии на C++ е задължително задаването на типа на връщаната стойност. Ако полето съдържа думата void, то функцията не връща резултат (стойност). Ключовата дума void задава празен тип. Името на функцията е идентификатор. Скобите след името на функцията са задължителни, дори ако в тях няма формални параметри. Формалните параметри са идентификатори. Типът на параметрите на функцията трябва задължително да се запишат.

Пример:

```
// Правилно:
void f( int x, int y );

// Грешно:
void f( int x, y);
```

За да се изпълни една функция на нея трябва да й се предаде управлението. Това става чрез обръщение към тази функция (чрез нейно извикване). Функцията, която предава управлението, е извикваща, а тази, която приема управлението, е извикана. Една извикана функция може да извика друга функция и т.н. Точно една функция в програмата е с име main. Тя се нарича главна функция. При стартиране на изпълнение на програмата на C/C++ Операционната Система предава управлението на главната функция, която от своя страна може да се обръща към други функции. Изпълнението на програмата завършва с изпълнението на главната функция, която връща управлението на Операционната Система. В програма на C/C++ е предвидена възможност за изход от всяка функция направо към операционната система чрез обръщение към стандартната функция exit. Резултатите от изпълнението на дадена функция могат да се върнат като се използва специалния оператор return, външни променливи или параметрите на функцията. Обръщение към функция, което не завършва с ";" , се разпознава от компилатора като операнд в израз и при изпълнение се замества със стойност, предоставляваща връщания резултат от извиканата функция. За да върне една функция искания резултат, тялото й трябва да съдържа оператора return.

return

Общ вид;

```
return (израз);
```

Note

"()" не са задължителни.
За по-голяма яснота се препоръчва употребата на "( )".

Операторът return може да се запише на всяко място в тялото на функцията, което е разрешено за оператор. В една функция могат да се използват няколко оператора return, когато това се изисква от нейната логика.

Действие на оператора return:

1. Пресмята се стойността на израза след оператора
2. Ако е необходимо типът на стойността се преобразува към типа на функцията
3. Изпълнението на извиканата функция се прекратява
4. Управлението се връща на извикващата функция и ефектът е такъв, като че ли обръщението към функцията се замества със стойността на върнатия от нея резултат.

Изразът след оператор return не е задължителен. Ако той (изразът) липсва, функцията не връща резултат и тогава типът на функцията трябва да бъде void. Можем да съставим и функция без да използваме return. В този случай ролята на return изпълнява затварящата фигурна скоба на тялото на функцията.

Задача: Да се напише функция, която преобразува малки в главни букви, а останалите символи не променя.

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

int toupper( char c )
{
        return ( ( c >= 'a' && c<='z' )? c-'a'+'A':c );
}
```

След като дадена функция е състаевена, можем да се обръщаме към нея.

Глава 14

Обръщение към функция

общ вид:

```
име_на_функцията ( списък_от_фактически_параметри );
```

В общия случай фактическия параметър е израз.

Обръщението към функция може да се използва като операнд в израз. В този случай, то не завършва с ";". Ако обръщението към функцията не е в израз, тогава трябва да завършва с ";". То представлява оператор в C/C++, но тогава върнатия резултат ще се загуби и няма да може да се използва в извикващата функция.

Ако функцията не връща резултат, тя не трябва да се използва като операнд.

Между формални и фактически параметри задължително трябва да има съответствие по тип и брой на параметрите. Като при необходимост типът на фактическите параметри се преобразува към типа на съответния формален параметър.

Пример:

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

using namespace std;

int toupper( char c )
{
        return ( ( c >= 'a' && c<='z' )? c-'a'+'A':c );
}

int main()
{
        char ch;

        cout<<"Въведете символ: ";
        ch = getchar();
        getchar(); // Clear the buffer;
        putchar( toupper ( ch ) ); // putchar () -> извежда един символ
}
```

Какво сле случва при извикване функцията:

1. пресмята се стойността на фактическия параметър и се записва в програмния стек
2. предава се управлението на функцията
3. разпределя се памет за формалния параметът

Note

тази памет е различна от паметта за фактическия параметър

4. започва изпълнение на функцията
5. тя връща резултат, който се замества обратно към функцията

Задача:

Да се състави програма за намиране на max в редица от числа (чрез функция).

Решение:

```
#include<iostream>

using namespace std;

float maxi ( float c[], int n )
{
	int i;
	float m;

	m = c[0];

	for( int i = 1; i<n; i++ )
		if( m < c[i])
			m=c[i];

	return m;
}

int main()
{
	int br, i;
	float mas[100];

	cout<<"Въведете броя на числата: ";
	cin>>br;

	for( i = 0; i<br; i++)
	{
		cout<<"Въведете mas["<<i<<"]= ";
		cin>>mas[i];
	}
	cout<<"Максималното число = "<<maxi(mas, br)<<endl;

	return 0;
}
```

Параметрите се пресмятат от дясно на ляво и се записват в програмния стек.

Стойностите на променливите се записват от горе на долу.

Записва се стойността на името на масива (т.е. адреса към първия му елемент).

Управлението се предава на извиканата функция.

Следва да се разпредели място за локални параметри. Паметта се разпределя от дясно на ляво. Когато се използва масив, се извършва заместване по име. За това не се декларира броя на елементите при едномерни масиви.

Свързваме по стойност или предаваме по стойност.

Глава 15

Прототипи на функции

Ако извиканата функция е разположена след извикваната, то трябва да бъде подходящо декларирана в извикващата функция. Това предварително обявяване се нарича прототип на функцията и информира компилатора за типа на на връщаната стойност и за броя и типа на формалните параметри. Използвайки прототипи компилаторът извършва контрол за съответствие по тип и брой между формални и фактически параметри,

Пример:

```
#include<conio.h> // контролиране на вход и изход
#include<iostream>

// Прототипи на функции:
void div(int a, int b);
int mnum(int, int);
// Имената на формалните параметри не го интересуват,
// но ако са написани не ги разглежда;

int main()
{
	int x, n1, n2;

	clrscr();
	// Изчиства екрана
	// функция част от conio.h

	cout<<"Въведете първото число: ";
	cin>>n1;
	cout<<"Въведете второто число: ";
	cin>>n2;
	div( n1,n2) ; // това е операция

	x = mnum(n1,n2);
	cout<<"max= "<<x<<endl;
	// Ако се обърнем към mnum() като към операнд,
	// няма да има грешка,
	// ама нищо не се извежда

	return 0;
}

void div(int a, int b)
{
	int t;
	while ( b!=0)
	{
		t = a%b;
		a = b;
		b = t;
	}
	cout<<"НОД: "<<a<<endl;
}
```

```
int mnum( int c, int d )
{
        return ( c  > d ) ? c : d;
}
```

Задача:

Да се напише програма, която транспонира матрица.

Решение:

```
#define DIM 3; // Дефинира константа

#include <iostream.h>

void trans ( float[][DIM], float[][DIM] );

int main()
{
        float u[DIM][DIM], v[DIM][DIM];

        int i, j;

        for( i = 0; i<DIM; i++)
        {
                for(j=0; j<DIM; j++)
                {
                        cout<<"Въведете u["<<i<<"]["<<j<<"]= ";
                        cin>>u[i][j];
                }
        }

        return 0;
}

void trans(float a[][DIM], float b[][DIM])
{
        int i,j;

        for(i=0; i<DIM; i++)
        {
                for(j=0; j<DIM; j++)
                        b[j][i] = a[i][j];
        }
}
```

Глава 16

Области на действие на променливите

В тялото на функция могат да се дефинират вътрешни променливи. Те са определени от мястото на дефиниране до края на съответната функция или това е тяхната област на действие. Външните променливи се дефинират в началото на програмата, преди всички функции или се дефинират между функциите в първичния файл. Те са определени от мястото на дефиниране до края на първичния файл. По такъв начин една външна променлива може да се използва от различни функции и чрез нея да се предават стойности между функциите. Имата на външните променливи трябва да бъдат уникални в първичния файл

Note

Първичен файл - този, който се компилира.

Ако в дадена функция се среща вътрешна променлива с име, съвпадащо с това на външните променливи, то вътрешната променлива скрива външната променлива в своята област на действие.

Пример prog.cpp:

```
float x,y;

void main()
{
	int k;
	// Тук може да се използват x и y;
}

float z;

void f1()
{
	int x;
	// Тук се използва вътрешната променлива x и
	// Външните променлив y и z;
}

void f2()
{
	int y;
	// Тук се използва вътрешната променлива y и
	// Външните променлив x и z;
}
```

Блок

Блок в C/C++ е последователност от дефиниции на променливи и оператори, заключен в {} . Блок може да се запише навсякъде в програмата, където е възможно да се запише оператор. Променливите, които се дефинират в блок са определени от мястото на дефиниринане до края на блока. Тези променливи са локални за блока. Там, където променливите са определени, е тяхната област на действие. Един блок може да съдържа в себе си друг блок. В този случай казваме, че блоковете са вградени един в друг. Променливите дефинирани в даден блок са глобални по отношение на

блоковете, съдържащи се в дадения блок. Променливите във вложените блокове могат да имат еднакви имена. В този случай променливите в даден блок скриват глобалните променливи със същите имена.

Пример:

```
void main()
{
	float x,y;
	{
		int k;
		// локална променлива k
		// глобални променливи x и y
	}

	float z;
	{
		int x;
		// локална променлива x
		// глобална променлива z и y;
	}

	{
		int y;
		// локална променлива y
		// глобална променлива z и x;
	}
}
```

Note

Добра практика:
При малки блокове е добра практика променливите да се дефинират в началото на блока.
При големи блокове е препоръчително дефинирането на променлив да става възможно най-близко до мястото на тяхното използване.
Това води до по-добра читаемост на кода.

Глава 17

Класове памет

В C/C++ всяка променлива се характеризира с 3 атрибута:

1. тип на променливата
2. област на действие (видимост)
3. период на активност.

Типът на променливата определя:

1. колко памет се разпределя за променливата
2. как се кодира стойността на промеливата

В следствие получава диапазон на стойсти, които променлива от даден тип може да приема.

Областта на действие зависи от това къде е дефинирана променливата.

Период на активност за една променлива - части от времето при при изпълнение на програмата, през което се съхранява последните текущи стойности на променливата.

Периодът на активност на една променлива може да бъде времето за цялостно изпълнение на програмата или времето, през което управлението се намира в областта на нейно действие.

В зависимост от периода на активност за променливата се разпределя памет на различни места. По друг начин можем да кажем, че променливата има различен клас памет.

В C/C++ има 4 класа памет.

- extern
- auto
- static
- register

Те се задават при дефиниране на променливата.

Общ вид:

```
клас_памет тип променлива;
```

extern не може да се задава при дефиниране.

По подразбиране външните променливи имат клас памет extern. Ако една променлива има клас памет extern, то за нея се разпределя памет в областта за статични данни. Тази област се разпределя преди да започне изпълнението на програмата и се освобождава след края на изпълнение на програмата.

Клас памет auto по подразбиране имат вътрешните и формалните параметри на функция. Променливи от този клас се наричат автоматични. За тези променливи се разпределя памет в програмния стек по време на изпълнение на програмата след влизане във функцията или блока, в който са дефинирани, когато управлението за първи път достигне мястото на тяхната дефиниция. Поради това отначало вътрешните променлив имат произволни стойности. Период на активност на автоматичните променливи съвпада с областта им на действие, т.е. стойностите им се пазят само докато се изпълнява функцията или блокът в нея. При повторно влизане стойностите на вътрешните променливи отнов са произволни.

Обикновено компилаторите анулират параметри, за които се разпределя памет в областта на статичните данни.

Note

Добра практика е задаването на начална стойност на променливи.

Пример за дефиниране на променливи от клас auto:

```
auto float x;
auto int i;
```

static

Задължително се задава явно при дефинирането на променлива. За променлива, дефинирана с клас памет static, се разпределя памет в областта за статични данни. Ако вътршна променлива се дефинира с клас памет static, то се променя нейният период на активност. Той става равен на времето за изпълнение на цялата програма.

Пример:

```
void f()
{
        static int i;
        float z;
        .........
        i=i+2;
        z=z+2;
        z=1;
}
```

Какво се случва? Началната стойност на i преди изпълнението е 0; След извикване на f() :

За z е разпределено място в програмния стек. Следователно не знаем каква е стойността на z.

i получава стойност 2.

За z получаваме неопределено число.

z получава стойност 1.

Свършва изпълнението на f. Следователно паметта за z в програмния стек се освобождава, т.е. губим последната стойност на z = 1.

При следващо извикване на f i=2 първоначално.

Когато външна променлива се опише като static, то нейният период на активност няма как да бъде увеличен.

Явно записан extern, то може променливата да се ползва в други модули.

Явно записан static външна промелива, то тя не може да се ползва в други модули.

register

Register е частен случай на auto.

В централния процесор има регистри. Тази памет е най-бързата памет.

При задаване на клас register "молим" компилаторът да разпредели регистър в централния процесор.

Такъв тип памет се използва за променливи, с които често се извършват операции.

Глава 18

Инцилиализиране на променливи

Инцилиазиране - присвояване на начална стойност на променливи.

За променливите може да се разпредели динамична памет:

В C чрез стандартни функции

В C++ чрез стандартни функции и чрез операции за разпределение и освобождаване на динамична памет.

Всяка променлива може да бъде инициализирана при нейното дефиниране по следния начин:

```
име_на_тип име_на_променлива = израз;
```

Променлива се инициализира като след името ѝ се постави символ за присвояване и желаната начална стойност:

```
double pi = 3.1415927;
```

Ако за променливата се разпределя памет в областта за статични данни, то изразът трябва да бъде константен.

Ако паметта се заделя в програмния стек, то изразът може да бъде произволен.

Ако променливата е вътрешна пресмятането на израза е всеки път, а static int i=5; става веднъж и след това променливата е инициализирана.

Ако вътршната променлива е автоматична, то тя се инициализира всеки път, когато се извиква функцията.

Ако за променлива се отделя памет в областта за статични данни, то тя запазва своите стойности докато не се промени явно с израз на присвояване. Примерно вътрешните статични промелниви се инициализират само 1 път и запазват своята стойност само веднъж. Ако се налога тази стойност да се промени става чрез израз за присвояване.

Вътрешните променливи могат да се инцилиазират при дефиниция или чрез израз за присвояване.

Пример:

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

Еквивалентно на:

```
void f1(int x, int y)
{
        int i;
```

```
        int j;
        i=1;
        j=x*y;
        ........
}
```

Note

Препоръчва се ползването на 1вия вариант.
Той е по-четим и разбираем. А и компилаторът създава по-ефективна програма, която работи по-ефективно.

Масивите могат да бъдат инициализирани при тяхното дефиниране с помоща на списък от начални стойности на елементите. Списъкът се поставя в {}.

```
int list[5] = { 0, 1, 2, 3, 4};
```

Еквивалентно на:

```
int list[] = {0,1,2,3,4};
```

Подреждането на началните стойности съответства на реда за разполагане на елементи на масиви в оперативната памет.

```
int mat[3][3] = { 0,1,2,3,4,5,6,7,8 };
```

Възможно е да се направи инициализиране по редове:

```
int mat[3][3] = {
                        {0,1,2},
                        {3,4,5},
                        {6,7,8},
                };
```

Кодът от предните два примера е напълно еквивалентен.

Инициализиране на символен масив - общо правило - чрез списък от символи:

```
char name[6]={'P','E','T','E','R','\0'};
```

Възможно е и инициализация от следния вид:

```
char name[6] = "PETER";
// еквивалентно с:
// char name[] = "PETER";
```

В този случай:

1. не се използват "{}"
2. не се задава '\0' (нулев бит). Той се добавя автоматично от компилатора.

Последнота налага следното правило:

Броят на елементите на масива трябва да е поне с 1 по-голям от броя на символите в низовите константи.

Ако броят на началните стойности е по-малък от броя на елементите на масива, то останалите 'неизползвани' елементи се анулират.

Ако броят на началните стойности е по-голям от броя на елементите на масива, то получавам съобщение за грешка.

Възможно е броят на елементите да не се задава. В този случай компилаторът приема, че броя на елементите е равен на броя на началните стойности.

При многомерни масиви може да не се зададе само първата размерност.

В C++ чрез ключовата дума const може да се дефинира променлива от тип const. Тя задължително се инициализира при дефинирането ѝ и тази стойност не може да бъде променяна.

Примери:

```
const double eps=0.001;
const double pi=3.14
const int initial = 100;
const int array[] = {1,2,3};

void f( const int a)
{
        // Тяло на функцията;
}
```

Note

Формален параметър на функция може да се декларира от тип const!
Това означава, че този формален параметър не трябва да се променя в тялото на функцията.
Компилаторът може да открива някои грешки в този случай.

При така дефинирани параметри прилагаме принципът на най-малките привилегии. С такава дефиниция показваме, че нещо не трябва да се случи.

C++ дава възможност по този начин ясно да се заяви, че нещо не може (бива) да се случва.

Глава 19

Указатели, основни операции

Указателят е променлива, чиято стойност е адрес от паметта.

Дефниране на указател:

```
име_на_тип *име_на_указателя;
```

На мястото на име_на_тип може да стои всеки валиден тип в C/C++.

Име_на_указател е идентификатор.

Note

За да дефинираме указател името на указателя трябва да има префикс $*$

Дефинираният тип на указателя трябва да съответства на типа на обекта, към който сочи указателя.

Ако това не е изпълнено програмистът трябва да знае какво точно ще се случи. В подобна ситуация повечето компилатори предупреждават за несъответстието.

Примери за дефиниране на указатели:

```
int *p;
float *p1;
double z, *p2;
// Тук само p2 е указател!
```

След като дефинираме даден указател той съдържа произволна стойност и трябва да бъде използван едва след като е инициализиран, т.е. след като ми присвоена дадена стойност, която е валиден адрес от паметта на компютъра. По определение указателите са променливи и следователно могат да бъдат инициализирани по общите правила за инициализиране на променливи.

Note

Ако указателят има стойност 0, тогава той е свободен и не посочва нищо.

Инициализиране на свободен указател:

```
p = NULL;
// или:
p = 0 ;
```

Използването на указател, който има стойност "0" е опасна. Не се знае какво се случва. Стойност 0 може да се използва като маркер (флаг) за идентификатор за грешка и др.

Note

Указателите не са цели числа и не трябва да се използват като цели променливи!

Операции с указатели:

& операнд -> дава адреса на операнда. Операндът трябва да бъде променлива.

операнд -> дава променливата, чийто адрес се определя от операнда. Операндът е израз, който има като стойност адрес.

Двете операции по-горе са едномерни (унарни) операции. Следователно и двете се изпълняват от тясно на ляво и са с висок приоритет.

Пример за работа с указатели:

```
float x,y, *px, *py;
x = 3.14;
// на x се присвоява стойност 3.14

px = &x;
// На указателя px се присвоява адреса на променливата x.

y = *px+10;
// 1. *px се замества със стойността на клетката намираща се на адреса сочен от px, т.е. стойността на x.
// 2. Към 3.14 ( стойността на x ) се прибавя 10.

py = px;
// Адресът на px се присвоява от py.
```

Чрез x се осъществава пряк достъп до x, а чрез ∗px косвен достъп.

Забележка:

```
y = ++ *px;
// равносилно с:
y = ++x;

y = (*px)--;
// равносилно с:
y = x--;
```

Пример:

```
float x,y;
int *px;
x = 3.14;

px = &x; // px = (int * ) ( &x);
y = *px;
```

При извикването на px = се взимат двата младши байта и те се разглеждат като допълнителен код на цялото число.

Чрез указателите се осъществява няколкократен косвен достъп до променливи.

```
int x, *px, **pxx;
```

x - пряк достъп до променлива

px - еднократен косвен достъп до променлива

pxx - двукратен косвен достъп до променлива

На указател не може да се присвои адрес на променлива от тип const, защото това би било непряк начин за промяна на стойността ѝ.

```
const int initial = 100;
int *ptr = &initial;  // този ред дава грешка при компилиране!
```

Възможно е дефиниране на указател, който сочи към променлива от тип const. Това става по следния начин:

```
const int *ptr = & initial;
```

Указателят към константа може да получи друга стойност, но променливата извлечена с него не може да бъде променяна:

```
const int k=1;
ptr = &k;
*ptr = 3; // дава грешка при компилация!
```

Възможно е да дефинираме указател, който е константа:

```
int * const ptr1 = &k;
// Самият указател е const.
*ptr1 = 5;
int a=2;
ptr1 = &a; - грешка;

int *const ptr2 = &initial;
// указател, който е константа и сочи към константа.
```

Глава 20

Адресна аритметика (аритметика с указатели)

При адресната аритметика са валидни само операциите събиране и изваждане на указател с цяло число. При тези условия могат да се използват ++, –, += и -= . При добавяне или изваждане на цяло число към или от указател вскя единица е равна на броя байтове, отделени в паметта за дадения тип на елементи от данните, към който сочи указателя.

```
float x, *px, *py;

px = &x;
px = px + 3; // добавят се 3 по 8 байта към адреса запазен в px

py = --px ; // В py се записва стойността на px минус 8 байта.
```

Note

Забележка:
При използване на адресна аритметика за обхождане на масиви могат да възникнат проблеми поради естеството на масивите и това как се заделя памет за тях.

Няколко операции с указатели:

1. ∗p++ - първо се извлича стойността сочена от p и след това се увеличава указателя p . Така той сочи следващ елемент.
2. ∗p– - извлича стойността сочена от p и след това се намалява указателя p така, че да сочи предходен елемент.
3. ∗++p - увеличава се стойността на указателя p така че да сочи следващ елемент и след това се извлича стойността на този елемент
4. ∗–p - намалява се p да сочи към предишен елемент и след това се извлича новата стойност сочена от p

Указателите могат да бъдат изваждани един от друг. Ако p1 и p2 са указатели към елементи на един и същ масив, то p2 - p1 ще определи броя на елементите между двата елемента, към които сочат p1 и p2.

Възможно е да проверява дали някой указател не е празен указател, т.е. със стойност null

```
if ( p == NULL )
        cout<<"Празен указател"<<endl;
else
        cout<<"Better luck next time"<<endl;
```

Глава 21

Предаване на резултати чрез формални параметри

Note

Параметър на функция може да бъде указател

```
void f( int x, int *p )
{
        ........
        *p = 10;
        x = x + 5;
        ........
}

void main()
{
        int a=15, b=5;

        f( a, &b);
        .........
}
```

p е фактически параметър. Той се взима в програмата от главната функция. Т.е. чрез този формален параметър се променя фактически параметър. Т.е. след извикването на f(); b вече има стойност 10, а не 5. Докато а остава не променена.

Note

При указатели напрактика заместването става по име.

В С винаги заместването е по стойност, а стойността на указателя е адрес. Следователно изпълнението на f се отразява по следния начин: а остава 15, а b се променя на 10.

Задача:

Да се състави програма, в която се чрез отделна функция се пресмята лицето на кръг и дължината на окръжност.

Решение:

```
#include<iostream>
#define PI 3.141592

using namespace std;

void fcircle( double r, double *s, double *p)
{
        *s = PI * r * r;
        *p = 2 * PI * r;
```

```
}

int main()
{
        double a,b,c;
        cout<<"Въведете радиуса: ";
        cin>>a;

        fcircle( a, &b, &c );

        cout<<"Лицето е " << b << "Дължината е " << c << endl;

        return 0;
}
```

В C++ може да се използва и заместване по име.

За целта се използват псевдоними:

```
int n;
int &nn = n;
// nn се нарича псевдоним на n;

double a[10];
double &last = a[9];
// можем да използваме last вместо a[9];

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

Псевдонимите се използват при заместване на формални с фактически параметри.

Формалния параметър може да се опише като псевдоним на фактическия.

```
void fcircle( double r, double &s, double &p )
{
        s = PI * r * r;
        p = 2 * PI * r;
}

int main()
{
        double a,b,c;
        cout<<"Въведете радиуса: ";
        cin>>a;

        fcircle( a, b, c);

        cout<<"Лицето е " << b << "Дължината е " << c << endl;

        return 0;
}
```

Така написана програмата изглежда по-естествено.

Note

Ако използваме псевдоними, то се извършва заместване по име.

В по-горния пример формалния параметър s е псевдоним на фактическия b, а формалния параметър p е псевдоним на фактическия c.

Напрактика формалния параметър е заместен с името на фактическия.

Глава 22

Формални параметри и масиви

До тук можем да подаваме масиви на функци по следния начин:

```
void f(int i, int a[] )
{
        // Some code goes in here.....
}

void main()
{
        int k, array[10];
        // more code goes in here....

        f(k, array);
        // this is also some code ....
}
```

Разглеждането на масив като указател при предаването му към функция разкрива някои нови възможности.

Например във функцията да се подаде част от масива, т.е. да се подаде подмасив:

```
void f1(int i, int *p )
{
        // Some code goes in here.....
}

void main()
{
        int k, array[10];
        // more code goes in here....

        f1(k, array[4);
        // еквивалентно на:
        f1(k, array + 4 );
        // Предава като указател в f1 подмасив на arrau като започва от 5тия елемент на масива.

        // this is also some code ....
}
```

В f1 можем да работим с подмасив по същия начин, по които бихме работили и с цял масив.

Нека разгледаме двумерен масив:

```
int  arr[5][10];
```

В C/C++ двумерните масиви всъщност се разглеждат като едномерен масив, чиито елементи са едномерни масиви (по-точно указател към едномерен масив).

Следователно arr и arr[i] са масиви.

arr[i][j] е елемент на двумерния масив.

arr[i] е указател към едномерен масив и като такъв може да се предаде във функция по вече познатия начин.

Примерна програма:

```
#include < ctype.h>
#define ROW 100
#define COL 200

void trans( char *p );

void main()
{
        char str[ROW][COL];
        int i;
        .........

        for ( i = 0; i < ROW; i++ )
                trans(str[i]);
        .......
}

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

Глава 23

Рекурсивни функции

Един обект е рекурсивен, ако частично се съдържа в сеже си или е дефиниран с помощта на себе си.

C/C++ поддържа две езикови конструкции, които позволяват да се реализират рекурсивни алгоритми - рекурсивни функции и структори с рекурсия.

Функция, която се обръща пряко или косвено към себе си , се нарича рекурсивна.

При всяко рекурсивно обръщение в програмния стек се разпределя памет за параметрите и вътрешните променливи. Използването на рекурсия води до увеличване производителността на програмисткия труд (обикновено рекурсивните алгоритми са по-кратки).

Но рекурсията не води до ефиктивни програми, защото при нея се използва повече памет и е необходимо повече време за изпълнение на програмата.

За да бъде една функция рекурсия, то трябва тя да бъде дефинирана рекурсивна.

Пример за нерекурсивна функция пресмятащата n! .

```
long fact ( int n)
{
        int i;
        long f = 1;

        for ( i = 2; i<=n; i++ )
                f = f * i;

        return f;
}
```

Пример за рекурсивна функция пресмящата n! :

```
long fact( int n )
{
        if ( n == 0 )
                return 1;

        else
                return ( n * fact( n - 1 ) );
}
```

Note

> При ползване на рекурсия има възможност за безброй много рекурсивни обръщения.
> За това е необходимо да има 1 или няколко "прости" случаи (без рекурсивно обръщение) и 1 или няколко случая с рекурсивни обръщения.

Да се състави програма, в която чрез рекусивна функция се определя НОД на две цели неотрицателни числа, които не са едновременнно равни на 0.

За пресмятане ще използваме следната рекурсивна дефиниция:

НОД (m, n) =

1. m, ако n = 0
2. НОД (n, m), ако n > m
3. НОД (n, m % n) в останалите случаи

Код:

```
#include <iostream.h>

int nod ( int m, int n)
{
	int result;

	if ( n == 0 )
		result = m;
	else if ( n > m )
		result = nod ( n, m ) ;
	else
		result = nod ( n, m % n);

	return result;
}

void main()
{
	int m, n;

	do{
		cout<<" m= ";
		cin>> m;
		cout<<"n = ";
		cin>> n;
	} while ( m < 0 || n < 0 || ( m == 0 && n == 0 ) );

	cout<<"\n НОД = "<< nod ( m, n ) << endl;
}
```

Глава 24

Конспект

24.1 Увод в програмирането

1. Алгоритми - определение, примери, свойства, начини на изразяване (словесно, блок-схеми, алгоритмични езици)
2. Променливи и типове данни
3. Константи (литерали). Видове
4. Коментари, структура на програмата и етапи на нейната обработка
5. Операции и изрази
6. Преобразуване на типовете
7. Условни изрази и условни оператори
8. Опертори за цикъл while и do while
9. Масиви. Низ от символи
10. Оператор за цикъл for
11. Оператор за избор на вариант switch. Оператори break, continue и goto
12. Функции - общ вид. Оператор return
13. Обръщение към функция
14. Прототипи на функции
15. Области на действие на променливите
16. Класове памет
17. Инициализиране на променливи
18. Указатели, основни операции
19. Адресна аритметика (аритметика с указатели)
20. Масиви и указатели
21. Представяне на резултати чрез формални параметри
22. Формални параметри и масиви
23. Рекурсивни функции
24. Стандартни функции за обработка на низове

25. Структури. Основни операции
26. Вложени структури. Рекурсивно използване на структури
27. Функции и структури
28. Структури и масиви
29. Дефиниране имена на типове
30. Изброен тип
31. Списък, стек, опашка, дек
32. Последователен списък и стек. Основни операции
33. Последователна опашка. Основни операции
34. Структури и модулно програмиране. Същност на обектно-ориентираното програмиране