

Програмно осигуряване на Компютърните Системи (КС)

КОМПЮТЪРЪТ е универсална информационна машина, която може да извършва много и разнообразни дейности. Широкият обхват на приложение на **КС** се обуславя от универсалните електронни устройства, от които са изградени и простата система от команди (инструкции), заложени като функционални възможности на централното процесорно устройство.

Разглеждан на това ниво, компютърът е апаратна част (**Hardware**).

Решаването на разнообразните задачи в КС се осъществява чрез

многократно изпълнение на множество елементарни аритметични и логически операции, включени в системата команди на микропроцесора.

Всяка елементарна операция се изпълнява от машината със задаване на определена команда (инструкция). Следователно, за машинното решаване на коя да е задача е необходимо предварително да бъде създадена поредица от подредени и свързани помежду си инструкции, подчинени на логиката на определен **АЛГОРИТЪМ**.

Такава поредица от систематизирани и логически обвързани инструкции се нарича **програма (Software)**
и стои в основата на т. нар. Програмно Осигуряване (ПО).

Програмни езици от Първо и Второ поколение

Лесно се съобразява, че усъвършенстването на хардуерната част на КС през годините води до бързо развитие на програмното осигуряване (и в частност на програмните езици).

Първоначално програмирането се извършвало в **машинен код**.

Това означава, че програмите са съдържали само двоични инструкции (машинен код на програмите), тъй като хардуерните елементи на КС можели да работят само с такива инструкции .

Впоследствие (1950) се появяват **асемблерските езици**, при които командите се записват с мнемонически кодове (кодови съкращения). Тези кодове се обработвали от специална транслираща програма, която ги преобразувала в машинни команди (двоичен код).

При тези езици, за първи път величините се представят чрез буквени означения (идентификатори), както в математиката.

Програмни езици от Трето поколение

Оказалось се по подходящо, програмите да се записват в термините на някакво по-високо логическо ниво, което да остава независимо от усъвършенстванията на хардуера на КС, а специални програми да "превеждат" инструкциите им до по-ниско ниво (машинни инструкции).

През втората половина на 50-те години на 20-ти век се появяват първите

програмни езици от високо ниво (трето поколение).

При тях се разработва обособена съвкупност от семантични конструкции и синтактични правила, които се използват за представяне на алгоритми във форма, удобна за обработка от компютър, но независима от хардуера на КС.

Тъй като те са средство за описание на алгоритми, тези езици се наричат още алгоритмични езици. Отличават се от машинно зависимите езици, по по-високото структурно ниво на инструкциите и машинната си независимост.

Специални програми (компилатори, транслатори или интерпретатори) се използват за преобразуване в двоичен код на текста на програмите, написани с алгоритмичните езици.

Програмни езици от Трето поколение

Всеки програмен език може да се разглежда като съвкупност от три основни компонента:

- **азбука** на програмният език – съдържа набор от букви, цифри и специални символи, допустими за описание на отделните команди и оператори. Освен това, в програмният език се включва определена система от специални думи, които се наричат ключови или запазени думи.
- **Синтаксис** – система от правила, които определят допустимите отношения между обектите на езика, на които се подчиняват всички инструкции (команди).
- **Семантика** – система от правила, позволяващи точно и еднозначно тълкуване на обектите от езика.

Основна структурна единица на програмите, създавани с различните програмни езици е **(командата) инструкцията**. Обикновено инструкциите в програмните езици се разделят на три категории: инструкции за декларации, изпълними инструкции и коментарии.

Програмни езици от Трето поколение

Един от най-важните световни програмни езици е
C/C++

Програмният език **C** е разработен от Денис Ричи (Dennis Ritchie) и Кен Томпсон (Ken Thompson) в началото на 70-те години на XX век за операционната система UNIX . Той комбинира контролните структури на езиците от високо ниво със способността да манипулира информацията до ново битове, байтове и указатели (адреси), което дава почти цялостен контрол на програмиста върху КС.

Развитието на **C++** започва през 1985, когато Бьорн Строустръп (Bjarne Stroustrup) създава обектно-ориентиран вариант на **C** , чрез който се цели да се моделират обектите от реалния свят. Днес може да се приеме, че този език съществува като отделен програмен език, който изграден върху основата на **C** е увеличил почти двойно размера му и е един от най-мощните компютърни езици, създадени досега.

Структура на С/С++ програма

```
#include <stdio.h>
#include <iostream>
#define MAX 100

int total_sum;
int func1(int par1,int par2);
float func2(float par1, float par2);

int main()
{
int a,b,c; float f,g,h;
.....
c=func1(a,b);
.....
f=func1(g,h);
.....
return(0);
}

int func1(int par1,int par2)
{
.....
}
float func2(float par1, float par2)
{
.....
}
```

Секция за директиви на предпроцесора на С

Секция за глобални декларации – на глобални променливи и на прототипи на функции

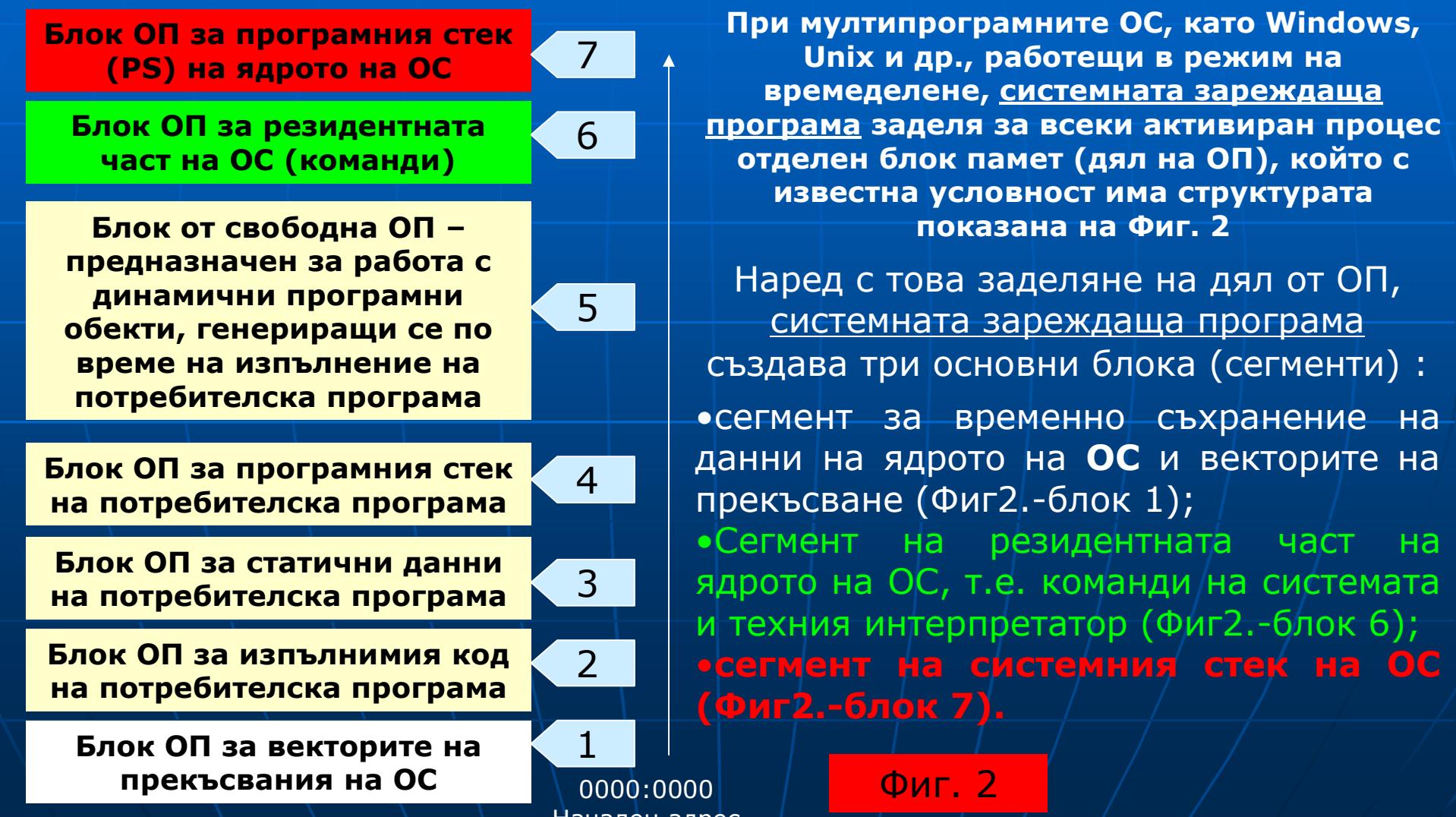
Секция за дефиниране на тялото от инструкции на главната функция main на програмата на С

Секция за дефиниции на потребителски функции

C/C++ програма и ОП

За по-добро разбиране на механизмите за взаимодействие между компонентите на една програма е необходимо да се разгледа как се изпълнява изпълнимия код в средата на операционната система (**ОС**) и как се разпределя наличната оперативна памет (**ОП**).

Разпределението на ОП се осъществява от два основни модула в ядрото на **ОС** – системна зареждаща програма и програма за динамично разпределение на ОП.



C/C++ програма и ОП

С композирането на сегментите 1, 6 и 7 се създават условия за работа на програмите (процесите) от Приложното Програмно Осигуряване (ППО).

При това, изпълнимите модули на ППО се изграждат от съответните свързващи редактори. Самото им стартиране се изпълнява отново от системната зареждаща програма на ОС на няколко етапа:

Етап 1a: Зареждане на изпълнимия код в сегмент, който съдържа асемблерския код на програмата и включва кода на функцията **main** и на всички функции използвани в нея (Фиг2.- блок 2);

Етап 1b: Заделяне на сегмент на статичните (глобални) данни, (в зависимост от типа им – т.е броя байтове с които се представят), които данни съществуват и са достъпни по време на изпълнението на потребителската програма (Фиг2.-блок 3)

Етап 1c: Заделяне сегмент на програмния стек (Фиг2.-блок 4)

Етап 1d: Заделяне сегмент на свободната памет (heap-куп), за динамичните обекти (Фиг2.-блок 5).

Етап 2: Обработка на изпълнимия код за относително преадресиране спрямо началната точка (адрес) на конкретното зареждане.

Етап 3: Настройка на възвратния адрес за връщане към ядрото на **ОС**, след завършване на потребителската програма (**command.com**)

Елементи на езика C/C++

Градивни символи: големи и малки латински букви, цифри, _

Служебни символи: .,:;?!<>()[]+-*/\=%&|~"{}#\$

Резервирана дума: специални комбинации от градивни символи.

Идентификатори (имена): комбинации от градивни символи.

Коментари: //текст или /*текст*/

Основни обекти

Операнд-Константа = именована (с идентификатор) или неименована постоянна величина

Операнд-Променлива = именована непостоянна величина (атомарна или структурирана), с принадлежност към даден тип, с възможност да приема стойност

Оператор = комбинация от служебни символи

Израз = комбинация от операнди и оператори

Инструкция = комбинация от резервирана дума и/или изрази, която завършва с ";". Остойностяването ѝ зависи от стойностите на operandите в изразите, а типът - от типа на същите operandи

Структурирана инструкция (клауза) = комбинация от резервирана дума и инструкции

Зона-инструкции = линейна последователност от една или много инструкции и/или клаузи и/или коментари

Блок-инструкции = зона-инструкции, заградени от { и }

Функция = самостоятелно обособена и логически завършена, именована програмна конструкция от инструкции, клаузи и коментари

Основни типове данни. Атомарни операнди в С/C++

5 основни типа данни в С

char символ

int цяло число

float число с плав. точка

double двойно число с плав. точка

void без стойност



2 типа данни в С++

bool булев (true/false)

Wchar_t широко-символен

Някои типове могат да се модифицират чрез т.нар. модификатори за тип:

signed

unsigned

short

long

const

Модификаторът предшества типа, а ако липсва тип, се подразбира **int**

Деклариране на operand-променливи

type varname1,varname2,...,varnameN;

type=тип на данни, **varnamei**=име на променлива, i=1,N;

Деклариране на operand-константи

const type constname1,...,constnameN;

const= модификатор за константа, забраняващ въздействие върху него

type=тип на данни, **constnamei**=име на константа

Инициализиране на operandи

constnamei = израз ; varnamei=израз;

Забележки: Всяка променлива трябва да бъде декларирана преди употреба ! Глобални или static локални променливи се инициализират само чрез изрази от константи !

Основни типове данни. Атомарни операнди в С/C++

Размер и обхват на типове данни

<i>bool</i>	true/false	1B
<i>char</i>	$[-2^7, +2^7 - 1]$	1B
<i>unsigned char</i>	$[0, 2^8 - 1]$	1B
<i>Wchar_t</i>	$[-2^{15}, +2^{15} - 1]$	2B
<i>int</i>	$[-2^{15}, +2^{15} - 1]$	2B
<i>unsigned int</i>	$[0, 2^{16} - 1]$	2B
<i>Long int</i>	$[-2^{31}, +2^{31} - 1]$	4B
<i>float</i>	$[-3.4e^{-38}, +3.4e^{+38}]$	4B
<i>Double</i>	$[-1.7e{-308}, +1.7e{+308}]$	8B

Примери:

```
int a,b,c; float f=10.5; char ch='A';  
const double pi=3.14;
```

Escape последователности

(Двусимволни константи)

\b	Backspace
\f	Връщане в началото на реда
\n	Нов ред
\r	Връщане на каретката
\t	Хоризонтална табулация
\"	Двойни кавички
'	Единична кавичка
\\"	Обратно наклонена черта
\v	Вертикална табулация
\a	Сигнал от високоговорителя
\N	Осмична константа N
\xN	Шестнайсетична константа N

Обхват на operandите

Операндите биват локални или глобални. Глобални са тези операнди, които са деклариирани в сурс-кода на програмата извън всякакви функции. Локални са онези операнди, които са деклариирани в тялото на функция или блок-инструкции. Глобалните операнди могат да се извикват навсякъде в кода на програмата. За разлика от тях, локалните операнди разпростират своето присъствие най-много в блока-инструкции, заграден с { и } , в който те са деклариирани.

Оператори в C/C++

Аритметични оператори

+	Събиране
-	Изваждане
*	Умножение
/	Деление
%	Делене по модул

Релационни оператори

<	По-малко
<=	По-малко или равно
>	По-голямо
>=	По-голямо или равно
==	Равно
!=	Не равно

Присвояващи оператори

=	Присвоява (директно)
+=	Увеличава с число и присвоява
-=	Намалява с число и присвоява
*=	Умножава с число и присвоява
/=	Дели на число и присвоява
%=	дели по модул и присвоява
<<=	Измества в ляво и присвоява
>>=	Измества в дясно и присвоява
&=	Двоично AND с присвояване
^=	XOR с присвояване
=	Двоично OR с присвояване

Логически оператори

&&	Логическо AND
	Логическо OR
!	Логическо NOT

Оператори в C/C++

Побитови оператори

&	Побитово AND
^	Побитово XOR
	Побитово OR
<<	Преместване наляво
>>	Преместване надясно
~	По битово NOT

Адресни оператори

&	Адрес на променлива
*	Деклариране на указател
**	Деклариране на указател към указател
:>	Базов адрес

Инкрементиращи/ декрементиращи оператори

++	Инкрементиране
--	Декрементиране

Специални оператори

?:	Логическо присвояване
,	списък от изрази
()	Групиране в изрази или описание на функции
[]	Индексиране на масиви
.	Избор на елемент от структура
->	клас чрез указател
(type)	Промяна на тип
sizeof	Размер в байтове
::	Деклариране на принадлежност
&	Псевдоним
.*	Указател към член-функция
->*	Указател към член-функция

Изрази в C/C++

Израз = комбинация от операнди и оператори, даваща възможност да променяме или сравняваме operandите, които сме декларирали и искаеме да ползваме.

```
a = 4*6 // присвоява на a стойност 24
b < 7 // проверява дали b е по-малко от 7
c == 5 // Не присвоява 5 на c, а проверява дали c е равно на 5.
!( 1 || 0 ) // стойността на израза е 0
!( 1 || 1 && 0 ) // стойността на израза е 0 (AND е преди OR)
!(( 1 || 0 ) && 0 ) // стойността на израза е 1 (Скобите са полезни)
```

Инструкции в C/C++

Инструкция = комбинация от резервирана думи и/или изрази, която завършва с ";". Остойностяването ѝ зависи от стойностите на operandите в изразите, а типът - от типа на същите operandи.

```
a = 4*6; // присвоява на a стойност 24
a = a+5; // присвоява на a старата ѝ стойност, увеличена с 5 (a=29)
d = !( 1 || 1 && 0 ); // присвоява 0 на d.
d++; // инкрементира d с единица
cout<<"Hi !\n"; // изобразява на экрана текст "Hi !" и отива на нов ред
delete ptr; // освобождава паметта, алокирана от new, с адрес ptr
```

Клаузи (структурирани инструкции) за логическо разклонение

if (израз)

инструкция;

или

if (израз)

{

инструкция1;
инструкция2;

...

инструкцияn;
}

{

инструкция1;
инструкция2;

...

инструкцияn;
}



Клауза **if**

(“1-алтернативен превключвател”)

Изчислителният процес, преминавайки през тази клауза, ще изпълни зоната с инструкции (една инструкция или блок-инструкции), следваща **if (израз)**, само ако логическата стойност на **израз** е истина, (т.е. $\neq 0$). Ако логическата стойност на **израз** е лъжа, (т.е. $= 0$), изчислителният процес ще прескочи цялата клауза и ще продължи нататък.

Блок-инструкции

Представлява **зона-инструкции**, поместена между служебните символи { и }, които се изпълняват последователно и се разглеждат като една сложно-съставна инструкция.

Клаузи за логическо разклонение

if (израз)

инструкция1;

else

инструкция2;

или

if (израз)

{

инструкция1;

инструкция2;

...

инструкцияn;

}

else

{

инструкция1;

инструкция2;

...

инструкцияn;

}

Клауза **if-else** ("2-алтернативен превключвател")

Изчислителният процес, преминавайки през тази клауза, ще изпълни зоната с инструкции (една инструкция или блок-инструкции), следваща **if (израз)**, само ако логическата стойност на **израз** е истина (т.е. $\neq 0$). Ако логическата стойност на **израз** е лъжа (т.е. =0), изчислителният процес ще изпълни зоната с инструкции (една инструкция или блок-инструкции), следваща резервираната дума **else**.

Клаузи за логическо разклонение

if (израз1)

инструкция1;

else if (израз2)

инструкция2;

else if (израз3)

инструкция3;

.....

else

инструкцияN;

или

if (израз1)

инструкция1;

else if (израз2)

{

1инструкция2;

2инструкция2;

...

Nинструкция2

}

else if (израз3)

инструкция3;

.....

else

инструкцияN;

Клауза **if-else if**

(“N-алтернативен превключвател”)

Преминавайки през тази клауза, изчислителният процес ще изпълни зоната с инструкции (една инструкция или блок-инструкции), следваща резервирана дума **if** или **else if**, ако логическата стойност на съответния им **изрази** е истина (т.е. $\neq 0$). Ако логическата стойност на всеки **изрази** е лъжа (т.е. $= 0$), то изчислителният процес ще изпълни зоната с инструкции (една инструкция или блок-инструкции), следваща резервирана дума **else**.

Забележка: Изчисляването на логическите условия става последователно.

Клаузи за логическо разклонение

```
switch (ключ-израз)
{
    case константа1:
        инструкция1;
        break;
    case константа2:
        1инструкция2;
        2инструкция2;
        ...
        пинструкция2;
        break;
    .....
    case константап:
        инструкцияп;
        break;
    default:
        инструкция(n+1);
        break;
}
```

Забележка1: Константите **константа i** са от целочислен или символен тип и са уникални.

Клауза **switch** ("N-алтернативен превключвател")

Преминавайки през тази клауза, изчислителният процес най-напред изчислява **ключ-израз**. След това се открива онази резервирана дума **case**, след която имаме **константа i** , съвпадаща с **ключ-израз**. Тогава започва да се изпълнява зоната с инструкции (една инструкция много инструкции и/или блок-инструкции) от нейната секция, докато се срещне резервираната дума **break**. Ако стойността на **ключ-израз** не съвпадне с никоя **константа i** , то се изпълнява зоната с инструкции от секцията **default**.

Забележка2: Ако не се използва **break** за край на дадена секция, се преминава към следваща секция, докато се срещне оператор **break** или се достигне края на клаузата **switch**.

Пример с логическо разклонение if-else if

```
#include <iostream>           // Прикачва заглавен файл iostream
using namespace std;         // Разрешава ползване на std

int main()                  // Най-важната част на програмата !
{
    int age;                // Декларира променлива...

    cout<<"Input your age: "; // Пита за възраст
    cin>> age;              // Въвежда годините в age
    cin.ignore();            // Очаква enter
    if ( age < 50 ) {        // Ако в age има число < 50
        cout<<"Pretty young!\n"; // Само показва, че if-else if работи...
    }
    else if (age>=50 && age<100) // Ако в age има число от 50 до 100
    {
        cout<<"You are old\n";   // Само показва, че if-else if работи...
    }
    else {
        cout<<"You are very old\n"; // Само показва, че if-else if работи...
    }
    cin.get();               // Задържа екран-прозореца
}
```

Клаузи за многократно повторение (цикли)

while (израз)

инструкция;

while (израз)

```
{  
    зона-инструкции;  
}
```



Фиг. 3

Клауза *while*
(цикъл с пред-условие)

Цикълът тип *while* е основна конструкция.

Действието му е следното (Фиг. 3):

1. Оценява се **израз**.
- 2a. Ако логическата стойност на **израз** е *False*, се реализира Край на цикъла и управлението се предава на инструкцията след клаузата **while**.
- 2b. Ако логическата стойност на **израз** е *True* се изпълнява тялото на цикъла.
3. Преминава се към точка 1.

Клаузи за многократно повторение (цикли)

Клауза **for** (цикъл с пред-условие)

for (израз-инициализация; израз-условие; израз-модификация)
{ зона-инструкции; }



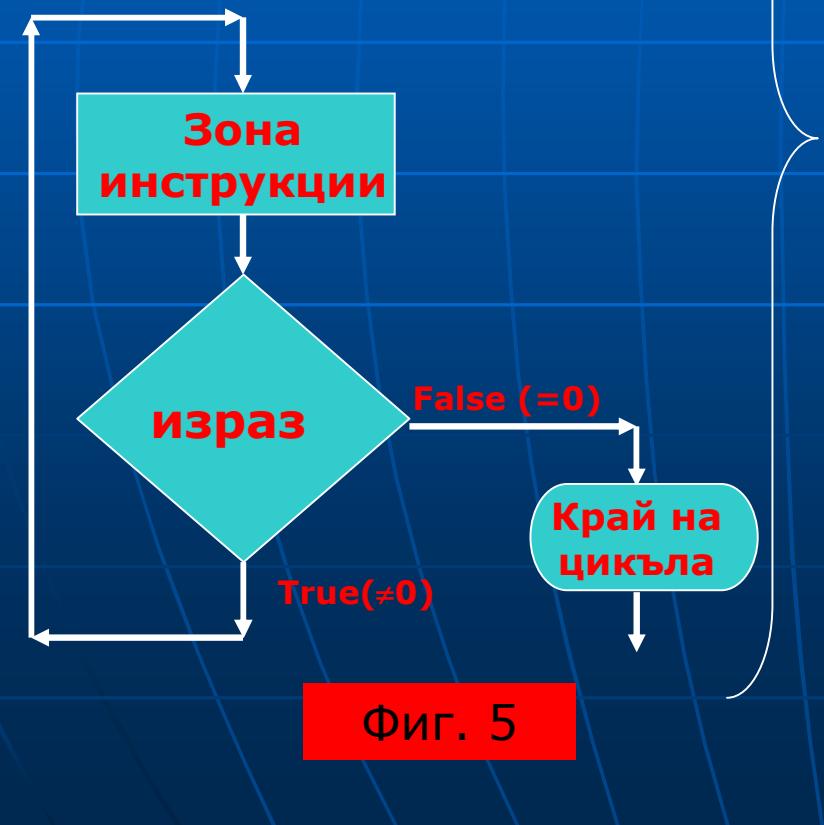
Цикълът **for** е много гъвкава форма на реализация на цикъл, аналогичен на **while**, в основата на който стои принципът на броенето. В него няма ограничения за типове изрази или за ролите, които те изпълняват. (Фиг. 4)

Например, не е задължително да се ползва израз-инициализация, за да се инициализира променлива за управление на цикъла, нито дори да се ползва такава, защото има много други средства за управление или излизане от този цикъл. Не е задължително и израз-модификация да модифицира някаква променлива. Технически погледнато, той е само един израз, който просто се изчислява.

Забележка: Възможно е изразите във **for** да са празни, с което може да се създаде безкраен цикъл, излизането от който е грижа на логиката в тялото му.

Клаузи за многократно повторение (цикли)

```
do  
{  
    зона-инструкции;  
} while (израз);
```



Клауза **do-while**

(цикъл с пост-условие)

Цикълът **do-while** е много полезна конструкция за действия, които трябва да се повторят поне веднъж.

Действието му е следното (Фиг. 5):

1. Изпълнява се тялото на цикъла.
- 2a. Ако логическата стойност на **израз** е **False**, се реализира Край на цикъла и управлението се предава на инструкцията след клаузата **do-while**.
- 2b. Ако логическата стойност на **израз** е **True** се преминава към т. 1.

Забележка: За разлика от другите цили, този с **DO.. WHILE** трябва да завърши с **;**.

Пример с цикъл *do-while* и логическо разклонение *switch*

```
#include <iostream>
using namespace std;

int main()
{
    float a,b;
    int izbor;

    cout<<"\nEnter first number a: ";
    cin>>a; cin.ignore();
    cout<<"\nEnter second number b: ";
    cin>>b; cin.ignore();

    cout<<"\n1.Add\n2.Subtract";
    cout<<"\n3.Multiply\n4.Divide\n";
    cout<<"\nChoose: ";
    do {
        cin>>izbor;
        cin.ignore();
    } while (izbor<1 || izbor>4);
```

```
switch (izbor) {
    case 1:
        cout<<"\n" <<a <<"+" ;
        cout<<b <<"=" <<a+b;
        break;
    case 2:
        cout<<"\n" <<a <<"-" ;
        cout<<b <<"=" <<a-b;
        break;
    case 3:
        cout<<"\n" <<a <<"*" ;
        cout<<b <<"=" <<a*b;
        break;
    case 4:
        cout<<"\n" <<a <<"/" ;
        cout<<b <<"=" <<a/b;
        break;
    default: break;
}
```

}

```
cin.get();
```

Пример за цикъл *for* и инструкции *break*; и *continue*;

```
#include <iostream>
#include <conio.h>
using namespace std;

int main()
{
    int i,a=10,b;
    char ch;
    for(i=1; ; i++) {
        if(!(i%4))
        {
            cout<<"\n";
            cout<<"Psswrd number: ";
            cin>>b;
            if(a!=b) continue;
            cout<<"\n" <<i
            cout<<", More ?<y/n> ";
            ch=getch();
            if (ch=='n') break;
        }
    }
}
```

Инструкция *break*; за служебно излизане от клауза

Тази инструкция прекратява незабавно изпълнението на клаузата (било то клауза за логическо разклонение или за циклично повторение) и предава управлението на инструкцията, следваща тази клауза.

Инструкция *continue*; за служебно прекратяване на текущата итерация на клауза

Тази инструкция предизвиква изпълнение на следващата итерация на цикъл, като пропуска кода, намиращ се между нея и проверката на условието за край на цикъла (за цикли *while* и *do-while*) или между нея и израз-модификацията с последваща проверка на условието за край на цикъла (за цикъл *for*).