Език за програмиране C

Характеристики на C

- Малък размер
- Използване на извиквания към функции
- Слабо типизиран
- Структурен език
- Възможности за програмиране на ниско ниво
- Използване на указатели
- Разлика между малки и главни букви

Структура на С програма

- Директиви на препроцесора
- Дефиниция на типове
- Прототипи на функции
- Дефиниране на променливи
- Дефиниране на функции
- Функция **`main()`**

Пример на С програма

```
#include <stdio.h>

int counter=0;

int main() {
      counter = 1;
      printf("Hello, counter is %d", counter);
      return 0;
}
```

Типове данни

C type	Size (bytes)	Lower bound	Upper bound
char	1	—	—
unsigned char	1	0	255
short int	2	−32768	+32767
unsigned short int	2	0	65536
(long) int	4	-2^{31}	$+2^{31}-1$
float	4	$-3.2 \times 10^{\pm 38}$	$+3.2 \times 10^{\pm 38}$
double	8	$-1.7 \times 10^{\pm 308}$	$+1.7 \times 10^{\pm 308}$

Дефиниране на променливи

Глобални променливи – дефинират се преди **main()** и имат област на видимост за цялата програма.

```
short number=1, sum;
int counter1=counter2=0;
char letter;

int main() {
...
```

Дефиниране на константи

Директива **#define**

```
#define TRUE 1
#define MAXCOUNT 100

int main() {
      ...
      if ( test == TRUE) ...
      ...
      for (i=0; i<=MAXCOUNT; i++) ..

}
```

Извеждане на екрана

Използва се форматен изход с функцията **printf().**

Форматиращ символ - %

- %c – извежда символ
- %d – извежда цяло число
- %l – извежда цяло дълго число
- %f – извежда число с плаваща запетая
- %s – извежда низ (трябва да завършва с \0)

```
printf("The counter is %d\n", counter);
printf("The letter is %c\n", letter);
```

\n – извежда символ за нов ред

\t – извежда табулация

Въвеждане от клавиатурата

Използва се форматен вход с функцията **scanf()**. Прочита въведени от клавиатурата данни в съответни променливи.

Форматиращ символ - %

- %c – въвежда символ
- %d – въвежда цяло число
- %l – въвежда цяло дълго число
- %f – въвежда число с плаваща запетая
- %s – въвежда низ

```
scanf("%d", &counter);
scanf("%c", &letter);
scanf("%d %c", &counter, &letter);
```

Присвояване на стойност на променлива

```
counter = 10;

counter += 20;

еквивалентно е на:

counter = counter + 20;
```

Аритметични оператори

- `+ - * /`

```
counter = A + B;
```

- Увеличаване/намаляване

```
counter = counter + 1
counter++

counter = counter - 1
counter--
```

Префикси и постфикси операции

- **Префиксни** – първо се увеличава/намалява стойността на променливата, после се взема стойността й.

```
(++counter) – A, аналог на:
counter=counter+1
counter-A
```

- **Постфиксни** – първо се взема стойността на променливата, после се увеличава/намалява.

```
counter++
```

Оператори за сравнение

Връщат резултат TRUE или FALSE.

- Проверка за еквивалентност **==**

```
if (counter == 0) ...
```

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

Логически оператори

За формиране на сложни логически изрази.

- Операция И - **&&**
- Операция ИЛИ - **||**
- Операция НЕ - **!**

```
if ( (counter == 0) && (test < 1) )...
```

Оператор if

Проверка на логически изрази.

```
if ( израз )
        оператор

if ( израз )
        оператор
else
        оператор

if ( израз )
        оператор
else if ( израз )
        оператор
else
        оператор
```

Оператор if

```
if ( counter == 0 )
        A = 0;

If ( counter == 0 ) {
        A = 0;
        B = 100;
}

if ( counter == 0 )
        A = 0;
else
        A = 1;

if ( counter == 0 )
        A = 0;
else if ( counter > 2 )
        A = 2;
else
        A = 100;
```

Оператор ?

Проверка на логически изрази в единичен оператор.

```
израз_1 ? израз_2 : израз_3
```

Ако *израз_1* е TRUE, то се изпълнява *израз_2*, в противен случай се изпълнява *израз_3*

```
z = (a > b) ? a : b
```

е еквивалентно на

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

Оператор switch

Вариантен оператор.

```
switch ( израз ) {
        case позиция_1:
                оператор;
                break;
        case позиция_2:
                оператор;
                break;
        ...
        default:
                оператор;
                break;
}
```

Оператор switch

```
switch ( letter ) {
        case 'A':
        case 'B':
                counter++;
                break;
        case 'C':
                counter = 100;;
                break;
        ...
        default:
                counter = 0;
                break;
}
```

Оператор за цикъл for

```
for ( израз_1; израз_2; израз_3 )
        оператор;
```

Инициализира се с *израз_1*, *израз_2* съдържа условието за завършване на цикъла, *израз_3* е модифициращ.

```
int i;

for ( i = 0; i <= 10 ; i++ ) {
        counter = A * 10;
        B = A + 20;
}
```

Оператор за цикъл while

```
while ( израз )
        оператор;
```

Проверява се *израз,* ако има стойност TRUE се изпълнява *оператор*, в противен случай цикълът завършва .

```
int i = 0;

while ( i <= 10 ) {
        counter = A * 10;
        B = A + 20;
        i++;
}
```

Оператор за цикъл do-while

```
do
  оператор;
while ( израз )
```

Изпълнява се *оператор*, накрая се проверява *израз,* ако има стойност FALSE, цикълът завършва.

```
int i = 0;

do
  counter = A * 10;
  B = A + 20;
  i++;
while ( i <= 10 );
```

Оператор break и continue

Възможност за контролиране на цикъла:

- **break** – изход от цикъл или *switch*
- **continue** – прескача към следваща итерация в цикъла

```
while ( value < 100 ) {
        if ( value = 50 ) {
                printf("Reach 10\n);
                break;
        }
        if ( value = 5 ) {
                printf("Reach 5\n");
                continue;
        ...
}
```

Масиви и низове

- **Масиви** – последователност от еднакви елементи. Елементите се индексират от 0.

```
int count_arr[10];  // едномерен масив
int table[10][20];  // двумерен масив

count_arr[2] = 100; // обръщение към елемент
                        на масив
count_arr[i] = 0;
```

- **Низ** – масив от символи. Последният елемент съдържа 0 за край на низа (символ \0).

```
char name[10];
name = "Ivan";
printf("Name is %s\n", name);
```

Функции

```
тип_връщан_резултат име ( парам1, парам2, ... ) {
        локални променливи

        код_на_функцията;
}

int sum ( int x, int y ) {
        int res;

        res = x + y;

        return res;
}
```

Извикване на функция:

```
the_sum = sum ( A, B );
```

Функция тип void

Функция която не връща резултат:

```
void име ( парам1, парам2, ... ) {
        локални променливи

        код_на_функцията;
}

void show ( int x) {

        printf("The value is %x\n", x);

}
```

Извикване на функция:

```
show ( A );
```

Прототип на функция

Преди една функция да се използва, компилаторът трябва да има информация какъв тип връща и какви параметри има.

Прототип е декларираме на функция само с тази информация. В последствие трябва да има пълната дефиниране на функцията (и нейния код).

```
int sum ( int x, int y );

int main () {
        ...
        rez = um ( A, B );
        ...
}

int sum ( int x, int y ) {
        return x + y;
};
```

Тип структура

```
struct gun {
        char name[50];
        int magazinesize;
        float caliber;
}

struct gun mygun; // променлива от тип структура
```

Инициализиране на променлива от тип структура:

```
mygun.name = "Uzi";
mygun.magazinezise = 20;
mygun.calibre = 7;

struct gun mygun = { "Uzi", 20, 7 };

mygun.magazinesize = 100;
```

Дефиниране на нов тип - typedef

```
typedef struct gun {
      char name[50];
      int magazinesize;
      float caliber;
} agun;

agun mygun;


agun mygun = { “Uzi”, 20, 7 };
```

Преобразуване на типове

Възможност променлива от един тип да бъде от друг тип.

```
int integernumber;
char letter = 'A';

integernumber = ( int ) letter;
```

Ще присвои стойност 65 (‘A’) на променливата **integernumber**.

Указатели

Указател – променлива, съдържаща адрес от паметта на друга променлива.

- Оператор **&** - дава адреса на променливата (референциране)

```
&counter
```

- Оператор * - дава стойността на променливата, сочена от указателя (дереференциране)

```
*pointer_to_counter
```

Указатели

```
int x = 1; y = 2;
int *pointer;

pointer = &x;   // pointer приема адреса на x
y = *pointer;   // y приема съдържанието, сочено от
                pointer
x = pointer;    // x приема стойността на pointer
*pointer = 3;   // паметта, сочена от pointer,
                приема стойност 3
```

Указатели

```
int x = 1; y = 2;
int *pointer;

pointer = &x;
```

x	y	pointer
1	2	100
100	200	1000

```
y = *pointer;
```

x	y	pointer
1	1	100
100	100	1000

```
x = pointer;
```

x	y	pointer
100	1	100
100	100	1000

```
*pointer = 3;
```

x	y	pointer
3	1	100
100	100	1000

Предаване на параметри към функция

Предаване по стойност – промяна на формалните параметри на функцията не променя оригиналите.

```
#include <stdio.h>
int a;
int main() {
    a = 3;
    func( a, 4 );
    printf("%d\n", a);
    return 0;
}
```

```
func( int x, int y ) {
   // x е 3 като изпратен параметър
   x += a;   // a се добавя към x, x е 6
   a += 2*y; // a остава 3, резултатът е 11
 }
```

Предаване на параметри към функция

Предаване по адрес – промяна на формалните параметри на функцията променя оригиналите.

```
#include <stdio.h>
int a;
int main() {
    a = 3;
    func( &a, 4 );
    printf("%d\n", a);
    return 0;
}
```

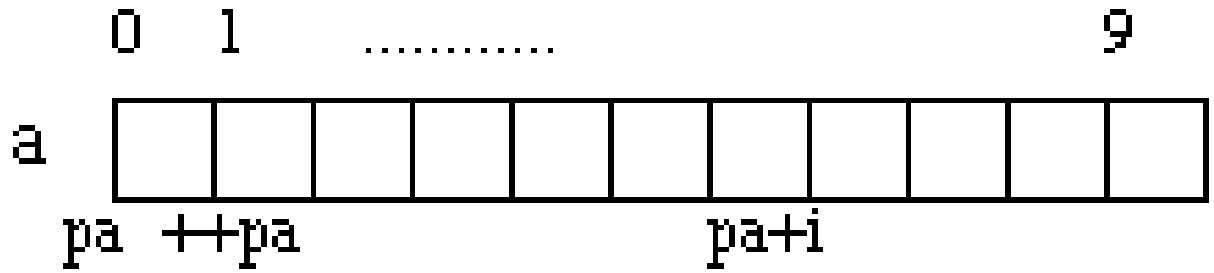
```
func( int *x, int y ) {
   // x е 3 като предаден параметър
   *x += a; // a е 6
   a += 2*y; // резултатът е 14
 }
```

Указатели и структури

```
struct gun {
      char name[50];
      int magazinesize;
      float caliber;
}

struct gun mygun;

struct gun *pointer;

pointer = &mygun;
pointer->name = “Uzi”;
pointer->magazinesize = 100;
pointer->caliber = 7;
```

Указатели и масиви

```
int a[10], x;
int *pa;

pa = &a[0];  // pa съдържа адреса на a

x = *pa;     // x = съдържанието на pa ( a[0] )
```



Работа с динамична памет

- Заемане на памет – **malloc()** . Паметта не се инициализира (запълва с 0). Изисква се размер.

```
int *ip;

ip = malloc( 4 ) ;

ip = malloc( sizeof(int) );

*ip = 100;
```

Работа с динамична памет

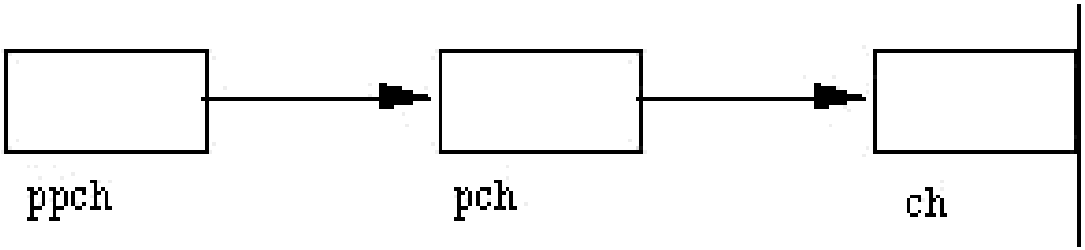
- Заемане на памет с инициализиране – **calloc()** . Паметта се запълва с 0. Изисква се брой елементи и размер.

```
int *ip;

ip = calloc( 1, sizeof(int) ) ;

*ip = 100;
```

Указатели към указатели

```
char ch;        // символ
char *pch;      // указател към символ
char **ppch;    // указател към указател към символ
```

- Указател към указател



Указатели към указатели

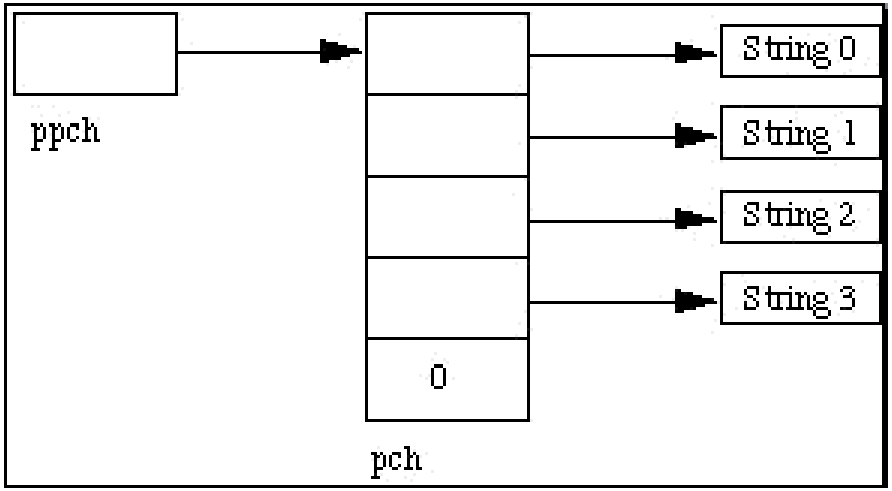
```
char ch;        // символ
char *pch;      // указател към символ
char **ppch;    // указател към указател към символ
```

- Указател към низ (низът завършва с \0)

Указатели към указатели

```
char ch;        // символ
char *pch;      // указател към символ
char **ppch;    // указател към указател към символ
```

- Указател към множество низове



Вход от командния ред

- Дава възможност за предаване на параметри при стартиране на програма.
- Функцията **main()** може да има параметри.

```
int main ( int argc, char **argv) {

        ...

}
```

- **argc** – брой аргументи в командния ред
- **argv** – масив от низове, всеки елемент съответстващ на един низ от командния ред (argv[0] е името на програмата)

Вход от командния ред

```
#include <stdio.h>
int main ( int argc, char **argv) {
      int i;

      printf("argc=%d\n\n", argc);
      for (i=0; i<argc; ++i) {
            printf("argv[%d]: %s\n, i, argv[i]);
      }
}
```

Ако е компилирана за изпълнение като **testarg**:

```
testarg f1 "f2" f3 4 stop!
```

Компилиране на програма

Използва се компилатора **gcc**.

```
# gcc -o име_на_изпълним_файл файл1.c файл2.c …
```

```
# gcc -o testprog main.c func.c
```

Изпълнение на програма:

```
# ./testprog
```

Въпроси?