Curs 2

Funcții

<result_type> name (<parameter_list>)

return <exp> după ce rezultatul se returnează, execuția funcției se termină, o funcție void nu trebuie să returneze o valoare după expresia return.

Definiția funcției nu face parte din declarare

definiția în fișier.h

```
/**
 * Computes the greatest common divisor of two positive integers.
 * a, b integers, a,b>0
 * return the the greatest common divisor of a and b.
 */
int gcd(int a, int b);
```

declararea în fișier.c

```
/**
 * Computes the greatest common divisor of two positive integers.
 * a, b integers, a,b>0
 * return the the greatest common divisor of a and b.
 */
int gcd(int a, int b) {
    if (a == 0 || b == 0) {
        return a + b;
    }
    while (a != b) {
        if (a > b) {
            a = a - b;
        } else {
            b = b - a;
        }
    }
    return a;
}
```

Specificații:

- nume sugestiv
- scurtă descriere pentru ce face funcția

- semnificația parametrilor
- condiții asupra parametrilor (precondiții)
- ce se returnează
- relația dintre parametrii și rezultat (postcondiții)

```
/*
* Verify if a number is prime
* nr - a number, nr>0
* return true if the number is prime (1 and nr are the only dividers)
*/
int isPrime(int nr);
```

precondiții - condiții care trebuie satisfăcute de parametrii actuali înainte de execuția funcției

postcondiții - condiții satisfăcute după execuția funcției

Apelul unei funcții:

- expresiile date ca parametru sunt evaluate înainte de execuția funcției
- parametrii actuali trebuie să corespundă cu cei formali (număr, poziție, tip)

```
int d = gcd(12, 6);
```

Vizibilitate (scope):

Locul unde declarăm variabila determină vizibilitatea ei

- variabile locale:
  - vizibile doar în interiorul blocului {} unde a fost declarată
  - vizibile doar în interiorul funcției unde a fost declarată
  - ciclul de viață a variabilei începe la declararea ei și se termină când execuția iese din domeniul de vizibilitate (memoria ocupată se eliberează)

- variabile globale
  - variabilele definite în afara funcțiilor sunt accesibile în orice funcție

Transmiterea parametrilor:

- prin valoare: void byvalue(int a);
  - la apelul funcției se face o copie a parametrilor
  - schimbările făcute în interiorul funcției nu afectează

variabilele exterioare

- prin referință : void byref (int *a);
  - la apelul funcției se transmite o adresă de memorie unde se află valoarea variabilei.
  - modificările din interiorul funcției sunt vizibile și în afara funcției

```
void byValue(int a) {
	a = a + 1;
}
void byRef(int* a) {
	*a = *a + 1;
}
void testArrayParam(int a[]){
	a[0] = 3;
}
int main() {
	int a = 10;
	byValue(a);
	printf("Value remain unchanged a=%d \n", a);
	byRef(&a);
	printf("Value changed a=%d \n", a);
	 int a[] = {1,2,3};
	 testArrayParam(a);
	 printf("value is changed %d\n",a[0]);
	return 0;
}
```

Valoarea returnată de funcție:

- Built in types : (int, char, double, etc) se returnează o copie.
- Pointer : se returnează o adresă de memorie. Nu returnați adresa unei variabile locale! Memoria alocată

de compilator pentru o variabilă locală este eliberată în momentul în care se iese din funcție.

· Vector: nu se poate returna adresa unui vector (int[]) dintr-o funcție. Se poate returna un pointer int* (adresa primului element).

· Struct: se comportă exact ca valorile built-in. Dacă struct-ul conține pointeri (char*) se copiază adresa, deci cele două struct-uri vor referi aceeași adresă de memorie. Dacă struct-ul conține vectori (char[20]) se copiază vectorul (20 de caractere), deci cele două struct-uri vor conține doi vectori independenți.

Copiere de valori în C:

O valoare se poate copia:

· folosind operatorul =
· la transmiterea parametrului unei funcții
· la asignarea valorii cu ce se returnează dintr-o funcție

Excepții de la regulile de copiere - vectorii

La transmiterea unui vector ca parametru la funcție se transmite adresa de început a vectorului.

Funcții de test:

#include <assert.h>

void assert (int expr);

expr - se evaluează expresia și dacă e falsă metoda assert generează o eroare și termină execuția.

```c
#include <assert.h>
/*
 * greatest common divisor .
 * Pre: a, b >= 0, a*a + b*b != 0
 * return gdc
 */
int gcd(int a, int b) {
	a = abs(a);
	b = abs(b);
	if (a == 0) {
		return b;
	}
	if (b == 0) {
		return a;
	}
	while (a != b) {
		if (a > b) {
			a = a - b;
		} else {
			b = b - a;
		}
	}
	return a;
}
```

```c
/**
 * Test function for gcd
 */
void test_gcd() {
	assert(gcd(2, 4) == 2);
	assert(gcd(3, 27) == 3);
	assert(gcd(7, 27) == 1);
	assert(gcd(7, -27) == 1);
}
```

Test Code Coverage – acoperirea testelor:

Idea: măsoară procentul de cod executat în urma rulării programului, adică numără liniile de cod efectiv executate la rularea programului.

Principii de proiectare pentru funcții:

- fiecare funcție să aibă o singură responsabilitate
- nume sugestive (nume funcție, nume parametrii, nume variabile)
- specificați fiecare funcție
- creați teste automate pentru funcții
- folosiți comentarii în cod

Programare modulară în C/C++:

Modulul este o colecție de funcții și variabile care oferă funcționalități bine definite.

Declarațiile de funcții sunt grupate-n fișiere header (.h)
Implementarea (definițiile pentru funcții) sunt grupate-n fișiere (.c)
Scopul este de separarea interfeței (.h) de cum sunt implementate funcțiile (.c)

Un modul este livrat printr-un fișier header (.h) și un fișier binar compilat (.dll pe Windows). Codul sursă (.c) nu este necesar să fie distribuit, adică implementarea poate fi ascunsă.

Directive de preprocesare:

Preprocesarea are loc înainte de compilare:

cod sursă → preprocesare → compilare → linkeditare → executabil

Preprocesarea permite includerea de fișiere header, definire de macrouri și compilare condițională.

Directiva #include:

- Pentru a avea acces la funcțiile declarate într-un modul se folosește această directivă. Prin această directivă se introduce fișierul referit în locul unde apare directiva.

Există două moduri de a referi un modul: <> sau " ":

```
#include "local.h" // caută fișierul header în directorul curent al aplicației
#include <header> // caută fișierul header în bibliotecile system
```

Principii de proiectare pentru module:

- Separați interfața de implementare
- includeți la începutul headerului o scurtă descriere a modulului
- module puternic coeziune (o singură funcționalitate)

· arhitectura stratificată : model, validation, repository, service, ui

Pointeri :

Este un tip de date folosit pentru a lucra cu adrese de memorie, stochează adresa unei variabile.

Operatori : &, *

```c
#include <stdio.h>

int main() {
	int a = 7;
	int *pa;

	printf("Value of a:%d address of a:%p \n", a, &a);
	//assign the address of a to pa
	pa = &a;
	printf("Value of pa:%d address of pa:%p \n", *pa, pa);

	//a and pa refers to the same memory location
	a = 10;
	printf("Value of pa:%d address of pa:%p \n", *pa, pa);
	return 0;
}
```

Null pointer este valoarea specială pentru a indica că pointerul nu referă o memorie validă

```c
#include <stdio.h>

int main() {
	//init to null
	int *pa1 = NULL;
	int *pa2;
//!!! pa2 refers to an unknown addres
	*pa2 = 6;

	if (pa1==NULL){
		printf("pa1 is NULL");
	}
	return 0;
}
```

```c
#include <stdio.h>
int* f() {
	int localVar = 7;
	printf("%d\n", localVar);
	return &localVar;
}

int main() {
	int* badP = f();
	//!!! *badP refera o adresa de memorie
	//care a fost deja eliberata
	printf("%d\n", *badP);
}
```

Vectori / pointeri :

· o variabilă de tip vector este un pointer la primul

element al vectorului.

· vectorul este transmis prin referință (se transmite adresa de memorie, nu se face copie.

· folosind funcția sizeof(var) se poate afla numărul de bytes ocupat de var.

Aritmetica pointerilor:

Se folosesc operații de adăugare / scădere pentru a naviga în memorie.

```c
#include <stdio.h>

int main() {
	int t[3] = { 10, 20, 30 };
	int *p = t;
	//print the first elem
	printf("val=%d adr=%p\n", *p, p);

	//move to the next memory location (next int)
	p++;
	//print the element (20)
	printf("val=%d adr=%p\n", *p, p);
	return 0;
```

Memory managment pe stivă

Pentru variabilele declarate, compilatorul alocă memorie pe stivă - o zonă de memorie gestionată de compilator

```c
int f(int a) {
   int *p;
   if (a>0){
      int x = 10;
      p = &x;
   }
   //here p will point to a memory location that is no longer reserved
   *p = 5; //!!! undefined behavior, the program may crash
   return 0;
}
```

Memoria este automat eliberată de compilator în momentul în care execuția părăsește domeniul de vizibilitate.

Alocare dinamică:

Folosind funcțiile `malloc(size)` și `free(pointer)` programatorul poate aloca memorie pe **heap** – o zonă de memorie gestionată de programator.

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

int main() {
	//allocate memory on the heap for an int
	int *p = malloc(sizeof(int));

	*p = 7;
	printf("%d \n", *p);
	//Deallocate
	free(p);
   //allocate space for 10 ints (array)
	int *t = malloc(10 * sizeof(int));
	t[0] = 0;
	t[1] = 1;
	printf("%d \n", t[1]);
	//dealocate
	free(t);
	return 0;
}
/**
 * Make a copy of str
 * str - string to copy
 * return a new string
 */
char* stringCopy(char* str) {
	char* newStr;
	int len;
	len = strlen(str) + 1; // +1 for the '\0'
	newStr = malloc(sizeof(char) * len); // allocate memory
	strcpy(newStr, str); // copy string
	return newStr;
}
```

`void *malloc(int n);`

– alocă n bytes de memorie, memoria este neinițializată

`void *calloc(int n, int size);`

– alocă n * size bytes de memorie, memoria este inițializată cu 0

`void *realloc(void *address, int newsize);`

– realocă newsize bytes de memorie, dacă spațiul unde urmează să se realoce memorie este ocupat, realloc alocă un nou bloc de memorie într-un alt loc, copiază conținutul vechi și eliberează blocul vechi.

void free (void * address);

- eliberează memoria (marchează blocul corespunzător adresei ca disponibil pentru reutilizare)

Memory leak:

```c
int main() {
        int *p;
        for (int i = 0; i < 10; i++) {
                p = malloc(sizeof(int));
                //allocate memory for an int on the heap
                printf("p\n", p);
        }
        free(p); //deallocate memory
        //leaked memory - we only deallocated the last int
        return 0;
}
```

Void and void *:

O funcție care nu returnează nimic:

```c
void f() {
}
```

Nu există variabile de tip void dar putem folosi pointeri la void *.

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

int main() {
       void* p;
       int *i=malloc(sizeof(int));
       *i = 1;
       p = i;
       printf("%d /n", *((int*)p));
       long j = 100;
       p = &j;
       printf("%ld /n", *((long*)p));
        free(i);
       return 0;
}
```

Se pot folosi structuri de date care funcționează cu orice tip de elemente dacă folosim `void *`. Problema este că nu se poate folosi `==` între elemente de tip `void *`.

Pointeri la funcții:

```
void *funcptr();      // a function that returns a pointer
void (*funcptr)();    // a pointer to a function
```

```c
void myFunction() {
    printf("Hello, world!\n");
}

int main() {
    void (*funcPtr)(); // Pointer la funcție
    funcPtr = myFunction; // Atribuire
    funcPtr(); // Apelare funcție prin pointer
    return 0;
}
```