Programare Orientată pe Obiect
Lucrarea de laborator Nr. 2

Concepte de bază ale limbajului C++. Programare procedurală.

Limbajul de programare C++ este o versiune extinsă a limbajului C al cărui scop este programarea orientată pe obiecte. C++ păstrează eficiența, flexibilitatea şi concepția de bază a limbajului C. C++ prevede mai multe facilități şi mai puține restricții decât limbajul C, astfel încât majoritatea construcțiilor din C sunt legale şi au aceeaşi semnificațiie şi în C++.
Pentru a exemplifica simplu ultima afirmație considerăm noțiunea de comentariu. Comentariile introduse într-un program sunt fragmente de text care sunt folosite pentru a explica, a traduce în limbaj natural fragmente de cod de program. Nefiind cod de program comentariile sunt ignorate de către compilator. În C un comentariu se introduce în program între perechile de caractere `/*` şi `*/`, poate apare oriunde în program şi se poate întinde pe mai multe linii de program:

```
/* un comentariu
în C++ */
```

Metoda din C poate fi deasemenea folosită în C++, dar C++ aduce o nouă metodă de a introduce comentarii în program prin folosirea unui „dublu slash", `//`, ce precede comentariul care poate începe oriunde pe o linie, fiind valabil până la sfârşitul liniei respective:

```
//un comentariu
//în C++
```

O bună practică de programare este de a utiliza noua metodă pentru toate comentariile şi de a o folosi pe cea veche pentru a izola fragmente de cod în timpul depanărilor (se preferă comentarea unui bloc de instrucțiuni din program pentru ca să nu mai fie luat în considerarea de către compilator decât ştergerea definitivă a liniilor de program şi rescrierea lor ulterioară, dacă este cazul). Totuşi este bine să nu se folosească abuziv comentarii când o claritate a definițiilor din program poate fi obținută prin denumiri sugestive a elementelor componente (variabile, constante. funcții, etc.).

1. Variabile

În afară de cuvintele-cheie ale limbajului (char, const, double, else, for etc., rezervate pentru anumite utilizări) într-un program se declară şi se definesc entități care servesc algoritmului particular implementat prin program. Fiecare entitate poată un *nume*, care este introdus în program printr-o *declarație*. *Definiția* entității rezervă cantitatea necesară de memorie şi execută inițializările corespunzătoare. Fiecare nume într-un program C++ are un *tip* asociat lui care determină ce operații se pot aplica entității la care se referă numele.
Tipurile fundamentale în C++ sunt tipuri de întreg, pentru definirea numerelor întregi de diferite dimensiuni (ca număr de octeți ocupați în memorie: `char` – 1 octet, `short int` – 2 octeți, `int` – 2 sau 4 octeți, `long` – 4 sau 8 octeți), tipuri de numere flotante, pentru definirea numerelor reale (reprezentate ca numere cu virgulă flotantă: `float` – 4 octeți, `double` – 8 octeți, `long double` – 12 sau 16 octeți). Aceste tipuri sunt denumite împreună tipuri aritmetice. Pentru tipurile întreg, există variante de declarație *cu semn* (`signed`) şi *fără semn* (`unsigned`).
Datele manipulate în cadrul unui program sunt stocate în *variabile*, care sunt identificate printr-un nume şi un tip. Numele pentru variabile pot fi secvențe de lungime oarecare de litere şi cifre, cu condiția ca primul caracter să fie literă sau caracterul de subliniere (*underscore*) şi cu condiția să nu fie cuvinte cheie.

Literele mici sunt diferite de literele mari corespunzătoare. În plus, cu câteva excepții pe care le vom indica ulterior o declarație este şi o definiție. Exemple de declarații şi definiții de variabile:

```
int a; // declararea şi definirea de variabile sunt instrucțiuni, deci se încheie cu „punct şi virgulă” (;)
int a; // eroare, redefinire; trebuie să existe o singură definiție a unei entități în program
float real=5.7; // într-o declarație a unui nume se poate introduce opțional o expresie de
                //inițializare pentru  acel nume
float Real=5.7;
char caracter='x'; // caracterul care inițializează o variabilă de tip char este cuprins între
                   //ghilimele simple, (‘ ‘)
```

2. Funcții

Programarea procedurală este prima modalitate de programare care a fost şi este încă frecvent folosită. În programarea procedurală accentul se pune pe descompunerea programului în proceduri (funcții) care sunt apelate în ordinea de desfăşurare a algoritmului. Limbajele care suportă această tehnică de programare prevăd posibilități de transfer a argumentelor către funcții şi de returnare a valorilor rezultate.
Un program C++ este format din funcții. În afara funcțiilor nu pot exista în program decât directive de preprocesor, de exemplu, pentru includere de fişiere header sau declarații şi definiții de variabile globale.
Într-un program C++ înstrucțiunile sunt grupate în blocuri de instrucțiuni cuprinse între acolade (`{}`). Mai mult, blocurile de instrucțiuni pot fi doar blocuri de definire a corpului unei funcții sau parte componentă a altor blocuri de instrucțiuni (blocuri imbricate).
Funcțiile reprezintă un tip derivat din tipurile fundamentale. Exemple de declarații de funcții:

```
void f1(); //funcția cu numele f1 nu are de returnat nici o valoare – void este un tip fundamental
           //care specifică o mulțime vidă de valori – şi nu are nici un parametru
           //  se poate scrie şi void f1(void);
float functie(int a); //funcția returnează o valoare de tip float şi primeşte la apel
                      //un argument formal (parametru) de tip întreg
double g(int, float); //declarațiile de funcții se mai numesc şi prototipuri
                      // în prototipuri, numele argumentelor formale sunt opționale
```

Definiția unei funcții include după tipul returnat, numele funcției şi lista de argumente formale, blocul de definiție al funcției. Un exemplu de definiție a unei funcții:

```
int f (int x)
{
      return x+1;
}
```

Funcțiile sunt apelate în cadrul altor funcții. La apelul unei funcții, argumentele de apel inițializează argumentele formale din declarația funcției, în ordinea din declarație. Argumentele unei funcții se pot transfera în două moduri: apelul prin valoare şi apelul prin referință. Apelul prin referință va fi prezentat mai târziu. În apelul prin valoare, se copiază valoarea argumentului real în argumentul formal corespunzător al funcției iar modificările efectuate asupra argumentului funcției (asupra copiei) nu modifică argumentul real. Un exemplu de apel al unei funcții în cadrul altei funcții şi transferul parametrilor:

```
#include <iostream.h>
int f(int); // prototipul este un model limitat al unei entități care urmează a fi definită mai
```

```
                // târziu (după apelul ei în funcția main)
                // prototipul permite compilatorului să verifice numărul şi tipurile argumentelor de apel
void g(int x){cout << "Rezultatul este" << x;} //funcția g afişează valoarea primită
void main()
{
      int a=10;
      a= f(a); // se apelează funcția f căreia i se transmite valoarea 10
                // iar funcția returnează valoarea 11 care este memorată în variabila a
      g(a); // se apelează funcția g căreia i se transmite valoarea 11
}
int f (int x)
{
      return x+1;
}
```

O funcție poate avea argumente implicite astfel încât ea poate fi apelată cu un număr de argumente mai mic:

```
int produs (int a, int b, int c=1, int d=1)
{
      return a*b*c*d;
}
void main()
{
      cout<<produs(2,7,4,5)<<endl; //a=2 b=7 c=4 d=5
      cout<<produs(2,7,4)<<endl; // a=2 b=7 c=4 d=1
      cout<<produs(2,7); // a=2 b=7 c=1 d=1


}
```

Numai argumentele de la sfârşitul listei pot fi argumente implicite, sau altfel spus, odată ce un parametru are o valoare implicită în lista parametrilor formali, toți cei ce urmează trebuie să aibă valori implicite. Nu este posibil să lăsăm găuri în mijlocul unei liste, decât dacă restul valorilor vor fi implicite. Valorile implicite se dau în prototip sau în headerul din definiția funcției şi nu în ambele, caz în care compilatorul ar trebui să verifice dacă ambele valori sunt identice, lucru care ar complica şi mai mult scrierea unui compilator pentru C++.

```
int produs (int a, int b, int c=1, int d); //eroare la compilare
```

În limbajul C, fiecare funcție definită în program trebuie să aibă un nume diferit. În C++, mai multe funcții pot avea acelaşi nume, dacă se pot diferenția prin numărul sau tipul argumentelor de apel. Mecanismul se numeşte supraîncărcarea funcțiilor (*overloading*):

```
int f(int);
long f(int, int);
void main()
{
      f(2); //apelează f(int)
      f(2,3); //apelează f(int, int)
}
```

Tipul returnat nu este folosit pentru a determina care funcție va fi apelată. Deasemenea, exemplul de mai sus face referire la acelaşi domeniu de definiție.

3. Domeniu de definiție, domeniu de vizibilitate

Domeniul de definiție al unui nume prin care se identifică o entitate a unui program este zona din program în care numele este cunoscut şi poate fi folosit. Dat fiind că un nume este făcut cunoscut printr-o declarație, domeniile numelor se diferențiază în funcție de locul în care este introdusă declarația în program. Un nume declarat într-un bloc este *local* blocului şi poate fi folosit numai în acel bloc, începând din locul declarației şi până la sfârşitul blocului şi în toate blocurile incluse după punctul de declarație. Argumentele formale ale unei funcții sunt tratate ca şi când ar fi declarate ân blocul cel mai exterior al funcției respective. Un nume declarat în afara oricărui bloc are ca domeniu fişierul în care a fost declarat şi poate fi utilizat din punctul declarației până la sfârşitul fişierului. Numele cu domeniu fişier se numesc nume *globale*.
Un nume este vizibil în întregul său domeniu de definiție dacă nu este redefinit într-un bloc inclus în domeniul respectiv. Dacă într-un bloc interior domeniului unui nume se redefineşte (sau se redeclară) acelaşi nume, atunci numele inițial (din blocul exterior) este parțial ascuns, şi anume în tot domeniul redeclarării. Un nume cu domeniu fişier poate fi accesat într-un domeniu în care este ascuns prin redefinire, dacă se foloseşte operatorul de rezoluție pentru nume globale (::). Redefinirea unui nume este admisă numai în domenii diferite. Dacă unul din domenii este inclus în celălalt domeniu, atunci redefinirea provoacă ascunderea numelui din domeniul exterior în domeniul interior. Redefinirea în domenii identice produce eroare de compilare. Un exemplu:

```
# include <iostream.h>
int index=13; // variabilă globală
void main()
{
float index=3.1415;//variabilă locală
cout<<"valoarea indexului local este" << index << endl;
cout<<"valoarea indexului global este" << ::index << endl;
}
```

O entitate este creată atunci când se întâlneşte definiția sa (care este unică) şi este distrus (în mod automat) când se părăseşte domeniul ei de definiție. O entitate cu nume global se creează şi se inițializează o singură dată şi are durata de viață până la terminarea programului. Entitățile locale se creează de fiecare dată când execuția programului ajunge în punctul de definire a acestora, cu excepția variabilelor locale declarate de tip `static`, care se inițializează o singură dată la prima execuție a instrucțiunii. O variabilă globală sau statică neinițializată explicit, este inițializată automat cu 0. Un exemplu:

```
static unsigned i; //  inițializată automat cu valoarea 0
```

4. Constante simbolice

O constantă simbolică (sau constantă cu nume) este un nume a cărui valoare nu poate fi modificată în cursul programului:

```
# include <iostream.h>
void afisare(const int data);//parametrul formal data este o constantă pe parcursul
                             //  întregii funcții şi orice încercare de a asigna o nouă
                             // valoare acestei variabile va produce eroare la compilare

void main()
{
      const int START=3;  //compilatorul nu va permite ca accidental sau intenționat
                          //să fie schimbată valoarea START pentru că a fost declarată constantă
```

```
	afisare(START);
}
void afisare(const int data)
{
	cout<<data<<endl;
}
```

Orice nume de funcție este o constantă simbolică.

5. Alte tipuri derivate

Tablourile (vectori, matrice) de entități de unul din tipurile fundamentale, cu excepția tipului `void` pot fi unidimensionale sau multidimensionale:

```
int x[]={1,2,3,4};
char str[]="abcde"; //vector de caractere ->şir

int y[5][5];
for(int i=0;i<5;i++)     //spre deosebire de C, în C++ variabilele se pot declara/defini
                         //cât mai aproape de locul în care sunt folosite
	for(int j=0;j<5;j++) // dacă în bucla for care execută de 5 ori operația de inițializare
                             //se execută mai multe intrucțiuni, acestea formează
                             // un bloc de instrucțiuni ce trebuie delimitat prin acolade
		//{
			y[i][j]=i+j;
		//}
```

Pointer-ul în C şi C++ reprezintă o adresă către o variabilă de un tip oarecare şi se declară cu operatorul * în fața numelui. Acesta realizează operația de dereferențiere, adică accesarea obiectului a cărei adresă o reprezintă pointer-ul. Un exemplu:

```
char c1='a'; //variabila c1
char* p1=&c1; //p1 memorează adresa lui c1; operatorul & - obține adresa lui c1
char c2=*p1; //dereferențiere, c2='a'
```

Definiția C++ permite ca un pointer către o constantă să fie definit astfel încât valoarea către care indică pointerul să nu poată fi schimbată, dar pointerul în sine să poată fi mutat către o altă variabilă sau constantă:

```
const char *nume1= "John"; //valoarea nu poate fi schimbată
```

Se pot declara şi pointeri constanți, care nu pot fi schimbați, nu pot indica altă variabilă:

```
char const *nume2= "John"; //pointerul nu poate fi schimbat
```

Numele unui tablou poate fi folosit ca pointer la primul element al tabloului:

```
char sir[]="abc";
char* p=sir;
char* q= &sir[0]; //p=q; p+1 indică elementul următor al tabloului, sir[1],
                  //deci p+1 este cu sizeof(char) octeți mai mare decât valoarea lui p
```

Pointerul la void (`void*`) permite programatorului să definească un pointer care să poată fi folosit pentru a accesavariabile de diferite tipuri în cadrul unui program:

```
int* pt_int;
float* pt_float;
void* general;
general=pt_int;
general=pt_float;
```

Variabilele *referință* se declară folosind operatorul ampersand în fața numelui şi se deosebesc de toate celelalte variabile pentru că acționează ca un pointer autoreferențiabil. După inițializare, variabila referință devine un sinonim pentru variabila de inițializare, astfel încât schimbând valoarea variabilei se schimbă şi valoarea referinței, deoarece, de fapt, ele referă aceeaşi valoare:

```
int i=1;
int& r=i;
i=i+1; // i++; i şi r au valoarea 2
```

Variabile referință nu se folosesc uzual aşa cum s-a arătat mai sus ci, mai degrabă, pentru apelul prin referință a unei funcții, atunci când se accesează direct variabila din argumentul real transmis funcției şi nu o copie a ei. Apelul prin referință permite transmiterea unei variabile unei funcții şi a modificărilor efectuate de aceasta în funcția apelantă:

```
#include <iostream.h>
void f(int in1, int& in2)
{
      in1= in1+10;
      in2= in2+10;
      cout<<in1<<in2;
}
void main()
{
      int a=7;
      int b=8;
      cout<<a<<b; // afişează 7 8
      f(a,b); // afişează 17 18
      cout<<a<<b; // afişează 7 18
}
```

6. Alocare şi eliberare dinamică de memorie

În limbajul C, se pot aloca dinamic zone în memoria liberă (*heap*) folosind funcții de bibliotecă (`malloc()`, `calloc()`, `realloc()`) şi se pot elibera folosind funcția `free()`. La aceate posibilități care se păstrează în C++, se adaugă operatorii de alocare şi eliberare dinamică a memoriei, `new` şi `delete`. În timpul creării C++, s-a simțit nevoia ca, din moment ce alocările şi aliberările dinamice de memorie sunt o parte intens folosită în programe, ele să devină o parte a limbajului în loc de o librărie adăugată, astfel că operatorii `new` şi `delete` sunt de fapt parte a limbajului C++, la fel cum sunt cei de adunare sau de atribuire. Exemple de utilizare:

```
int* pi=new int(3);// alocare int şi inițializare
double* pd=new double;// alocare fără inițializare
delete pi;
delete pd;
```

O situație care trebuie evitată:

```
int *p1, *p2;
*p2=7;
p1=new int;
p2=p1; // p2 va referi acum variabila la care pointează şi p1,
       //în acest caz orice referință către variabila la care pointa anterior p2 este pierdută
       //şi nu mai poate fi eliberată (dealocată);
       //ea este pierdută în heap până la întoarcerea în sistemul de operare
delete p1;// p1 este dealocat, iar p2 nu mai poate fi eliberat
```

Folosirea operatorilor `new` şi `delete` pentru alocarea dinamică şi dezalocarea unui vector unidimensional este următoarea:

```
char* pv=new char[20];
delete []pv;// este corect şi delete pv; pentru că char este un tip predefinit
            //  dar nu şi pentru tipuri definite de utilizator
```

7. Tipuri definite de utilizatori

Ca şi în C, în C++ se pot folosi tipul enumerare, uniunile şi structurile. Un exemplu:

```
struct complex{
      int real;
      int imaginar;
};
void main()
{
      complex nr1;//nu mai este necesară folosirea cuvântului struct la definirea variabilei ca în C
      nr1.real=1;// inițializarea variabilei tip structură prin inițializarea fiecărui membru în parte
      nr1.imaginar=1;
}
```

8. Operatori şi instrucțiuni

Se folosesc în C++ operatorii din C:

- `+,-,*,/ , %` - operatori matematici;
- `++/--` - incrementare/decrementare;
- `!` negație, `&&` SI logic, `||` SAU logic, etc.,

şi deasemenea, instrucțiunile din C:

- `if-else`
- `do-while`
- `for`
- `switch-break-continue`, etc.

9. Exerciții

1. Scrieți un program care să afişeze dimensiunea tipurilor fundamentale de date cu ajutorul operatorului `sizeof`.
2. Creați un tablou bidimensional de dimensiune 5 x 5 de numere de tip întreg, în care numerele din prima coloană sunt multipli consecutivi ai lui 10, iar elementele consecutive din linii sunt crescătoare cu o unitate. Afişați elementele tabloului.
3. Selectați elementele (2,3), (4,4), (4,0) din tabloul definit la exercițiul precedent cu ajutorul pointerilor şi afişați conținutul acestora prin dereferențiere. Afişați conținutul întregului tablou de mai sus folosind pointeri, operații asupra acestora şi dereferențierea.
4. Creați o functie numită `uppercase()` care să transforme literele mici din şirul cu care a fost apelată în litere mari. Demonstrați funcționarea sa într-un program.(Indicaţtie: utilizați funcția `toupper()` pentru a converti literele mici în litere mari).
5. Scrieți o functie numită `medie()` care să calculeze media unui şir de valori de tip `float`. Funcția va avea două argumente.Primul este un pointer catre un tablou conținând numerele, al doilea este o valoare întreagă ce reprezintă dimensiunea tabloului.Demonstrați folosirea ei într-un program.
6. Scrieți 3 versiuni ale funcției `suma()` cu următoarele prototipuri:
   ```
   int suma(int , int)
   float suma(float , float)
   double suma(double , double)
   ```
   Demonstrați folosirea lor într-un program.
7. Să se scrie o funcție în care se detectează eroarea de alocare a memoriei cu operatorul de alocare `new`.
8. Să se rescrie următoarea instrucțiune for printr-o instrucțiune echivalentă while:

   ```
   for (i = 0; i < max_length; i++)
   if (line[i] == '?') count++;
   ```

   Să se dea o nouă implementare folosind un pointer.

9. Să se rescrie funcțiile `strlen()`, care returnează lungimea unui şir, `strcpy()`, care compară două şiruri şi `strcpy()` care copiază un şir în alt şir. Să se considere ce fel de tipuri de argumente sunt necesare, după aceea să se compare cu versiunea standard declarată în `<string.h>`.
10. Să se scrie o funcție `strcat()` care concatenează două siruri. Se va folosi operatorul `new` pentru alocarea spațiului necesar rezultatului.
11. Să se scrie o funcție `strrev()` care inversează poziția caracterelor într-un şir dat.
12. Să se scrie o funcție `atoi()` care returnează valoare de tip întreg a unui şir de cifre zecimale. Să se scrie o funcție `itoa()` care generează şirul de caractere care reprezintă un număr întreg dat.
13. Să se scrie declarațiile pentru următoarele tipuri de variabile: pointer la un caracter, un vector de 10 valori întregi, pointer la un vector de 10 valori întregi, un pointer la un pointer la un caracter.