3. Funcţii şi domenii de accesibilitate

Un mod tipic de a face ceva într-un program C++ este de a apela o funcţie care face acel ceva. A defini o funcţie înseamnă a specifica cum trebuie făcută operaţia respectivă. O funcţie nu poate fi apelată dacă nu e *declarată* şi *definită*.

3.1 Generalităţi

Funcţiile se reprezintă prin *nume*, *listă de argumente*, *corp* (în care se defineşte funcţia) şi un *tip al valorii returnate*. O funcţie care nu returnează nimic este de tip `void`.

Exemplu:

```
inline int abs(int i)
{

   // returneaza valoarea absoluta a lui i
   return (i ? -i : i );
}

int CMMDC( int v1, int v2)
{
   // returneaza cel mai mare divizor comun a 2 intregi
   int temp;
   while(v2)
   { temp=v2;
     v2=v1%v2;
     v1=temp;
   }
   return v1;
}
```

O funcţie este evaluată atunci când operatorul de apel `()` este aplicat numelui funcţiei. Argumentele operatorului de apel `()` se numesc *argumente actuale* şi sunt transmise funcţiei.

Exemplu:

```
#include<iostream.h>
#include "cmmdc.h"

main()
{
   int i, j;
   // citeste valorile de la intrarea standard
   cout<<" valoare1 : ";     cin >> i;
   cout<<" valoare2 : ";     cin >> j;
   cout<<"\n min :" << Min(i,j) << '\n';
   i=Abs(i);
   j=Abs(j);
   cout<<"cmmdc :"<<CMMDC(i,j)<<'\n';
}
```

Printr-un apel de funcţie se pot întâmpla două lucruri:

- dacă funcţia este declarată `inline`, atunci fiecare apel este înlocuit (la compilare) cu corpul funcţiei;
- în rest, funcţia va fi invocată (apelată) în timpul execuţiei programului.

Prin apelarea unei funcţii, funcţia curentă se suspendă. Se execută funcţia apelată şi apoi se reia execuţia funcţiei apelante în punctul imediat următor apelului. Apelurile se gestionează prin intermediul stivei programului. În momentul folosirii, orice funcţie trebuie sa fie declarată. Pentru declaraţie se foloseşte *prototipul funcţiei* prin care se specifică tipul valorii returnate, numele şi lista de argumente. Prototipurile împreună cu definiţiile funcţiilor inline se pun în fişiere header.

Exemplu:

```
/* cmmdc.h */
```

```
inline Abs(int i) { return i<0 ? -i : i ); }
inline Min( int v1, int v2) { return (v1<=v2 ? v1 : v2 ); }
int CMMDC(int, int);
```

O functie se poate apela pe ea însăşi (*recursivitate*), dar este necesară definirea unei condiții de oprire.

Observație: Datorită procesului de invocare, o funcție recursivă este, în general, mai lentă decât varianta iterativă.

Exemplu:

```
int RCMMDC( int v1, int v2)
{
   // cmmdc, varianta recursiva
   if( v2==0 ) return v1;   // conditia de oprire
   return RCMMDC( v2, v1%v2 );
}
```

`inline` este doar o recomandare pentru compilator. Nu orice funcție poate fi complet expandată, de exemplu una recursivă ca `RCMMDC`. De asemenea, nu se recomandă expandarea funcțiilor foarte mari (ex 1000 linii). Mecanismul `inline` permite optimizarea funcțiilor foarte mici şi foarte frecvent apelate.

Limbajul C++ este puternic tipizat: atât lista de argumente cât şi valoarea returnată sunt verificate din punct de vedere al tipului în momentul compilării. Daca e posibil, diferențele de tip sunt rezolvate prin conversiile implicite.

Exemplu:

```
CMMDC(3.14, 6.29);        // cmmdc(3,6);
CMMDC(24312);             // eroare
CMMDC("hello", "world");  // eroare
```

De unde ştie compilatorul cu cine să compare tipul pentru parametrii transmişi? Din prototipul funcției. De accea, nici o funcție nu poate fi apelată fără să fi fost declarată. Valoarea returnată poate fi de orice tip predefinit, definit de utilizator sau derivat din acestea (pointer sau referință), exceptând tablourile şi funcțiile. Totuşi, pot returna pointeri la tablouri sau la funcții. O funcție care nu returnează nimic ar trebui să fie de tip `void`. O funcție care nu are specificat tipul valorii returnate se presupune că returnează un întreg (`int`). Funcțiile de tip `void` nu trebuie neaparat să conțină o instrucțiune `return`. `Return` poate fi folosită pentru a întrerupe execuția funcției curente şi întoarcere la funcția apelantă.

O funcție poate returna o singură valoare, dar dacă avem nevoie să returneze mai multe valori, atunci:

- se pot folosi variabile globale (dar cu riscul unor efecte colaterale)
- datele returnate pot fi organizate în clase (se pot returna clase, pointeri la clase sau referințe la clase).

3.2 Lista de argumente

Lista de argumente nu poate să lipsească. O funcție care nu primeşte argumente trebuie reprezentată cu o lista de argumente vidă sau cu o listă conținând ca singur argument cuvantul cheie `void`.

Exemplu:

```
int Fork();
int Fork(void); // declaratii echivalente
```

Lista de argumente se mai numeşte şi *semnătura funcției* pentru că este folosită pentru a distinge funcțiile cu acelaşi nume. La definire nu pot fi două sau mai multe argumente cu acelasi nume. La declarare, numele argumentelor este ignorat.

Exemplu:

```
Min( int v1, v2);       // Eroare
```

```
Min( int v1, int v2 );  // Ok
Min(int, int);          // Ok
```

3.2.2 Signatura specială: lista de argumente cu valori implicite

Rezolvarea acestor situații se face pozițional.

Exemplu:

```
char *ScreenInit(int line=24, int col=80, char backGround=' ');
char *cursor;
cursor = ScreenInit();  // cursor=screenInit(24,80,' ');
cursor = ScreenInit(66);// cursor=screenInit(66,80,' ');
```

Prin convenție, valorile implicite se specifică o singură dată la declararea funcției (într-un header).

3.3 Transmiterea parametrilor

Lista de argumente descrie argumentele formale. În momentul apelului, fiecărei funcții i se alocă o zonă în stiva programului, numită *structură de activare*. Fiecărui argument formal i se alocă un spațiu în structura de activare care este inițializat prin intermediul argumentului actual corespunzător. Metoda implicită de inițializare este prin copierea valorii argumentului actual (transmiterea parametrilor prin valoare). Funcțiile manipulează copii locale ale parametrilor actuali, aflate în structura de activare. Funcțiile nu modifică valoarea parametrilor actuali, deci nu e necesar ca programatorul să memoreze valoarea veche şi s-o restaureze.

Transmiterea prin valoare nu este adecvată în urmatoarele situații:

- argumentele sunt clase de dimensiuni mari
- funcția modifică valoarea argumentului.

Exemplu:

```
void Swap( int v1, int v2)
{
   // se apeleaza Swap(i,j), parametri transmisi prin valoare
   int temp=v2;
   v2=v1;
   v1=temp;
}
void PSwap( int *pv1, int *pv2 )
{
   // se apeleaza PSwap(&i,&j)
   int temp=*pv2;
   *pv2=*pv1;
   *pv1=temp;
 }
```

C++ permite şi transmiterea parametrilor prin referință.

Exemplu:

```
void RSwap( int& v1, int& v2)
{
   // se apeleaza RSwap(i,j);
   int temp=v2;
   v2=v1;
   v1=temp;
}
```

Referințele se vor folosi cu atenție, pentru că în cazul unor neconcordanțe de tip se generează obiecte temporare (este nevoie de *exact match*).

Exemplu:

```
int i=10;
unsigned int ui=20;
RSwap(i, ui); // se genereaza int T1=(int)ui;
              // RSwap(i,T1); nu este transmis ui !!
```

```
// daca i=10, ui=20
// Swap(i,ui) => i=20, ui=20.
```

Referințele se folosesc mai degrabă pentru tipurile utilizator decât pentru cele predefinite. În cazul în care există pericolul de a modifica valorile unor parametri actuali, aceştia pot fi declarați `const`.

Exemplu:

```
int Inc(int& i)
{
   return ++i;
}

int G(const int& x)
{
   return inc(x);  // Eroare
}
```

Funcțiile care returnează referințe pot fi obiectul unor atribuiri.

Exemplu:

```
char& Element( char *s, int i)
{
   return s[i];
}
void F()
{
   char *text="mac";
   char x = element(text,1);
   element(text,1)='i';
}
```

Tablouri

În C++, tablourile nu sunt transmise niciodată prin valoare, ci prin adresa primului element.

Exemplu:

```
// urmatoarele declaratii sunt echivalente
void PrintT( int*, int size);
void PrintT( int[], int size);
```

Pentru tablourile multidimensionale trebuie specificate toate dimensiunile, exceptând-o pe prima.

Exemplu:

```
void PrintM( int m[][10], int rowSize );
void PrintM( int (*m)[10], int rowSize );
// pointer la un tablou cu 10 elemente
// m+1 = adresa liniei 2
```

3.4 Domenii de accesibilitate

S-a mai spus că un identificator trebuie să fie unic. Asta nu înseamnă că un nume poate fi folosit o singură dată într-un program, ci că în funcție de context trebuie să se poată face distincția între diversele instanțe ale numelui respectiv. Fiecare identificator definit/declarat poate fi utilizat într-o zonă bine determinată a programului numită *domeniu de accesibilitate* al identificatorului respectiv.

În C++ există trei categorii de domenii de accesibilitate:

- la nivel global (fişier)
- la nivel local
- la nivel de clasă.

Domeniul global este cel mai cuprinzător. El conține tot ceea ce nu este conținut într-o definiție de funcție sau clasă dintr-un fişier sursă.

Observație: listele de argumente ale funcțiilor nu fac parte din domeniul global.

Operatorul `::` este operatorul domeniu global.

Domeniul local este determinat de fiecare definiție de funcție şi de fiecare bloc de instrucțiuni, inclusiv unele în altele.

Asupra *domeniului clasă* vom reveni cu detalii. Deocamdată trebuie reținut că:

- fiecare clasă reprezintă un domeniu de accesibilitate distinct
- fiecare membru este tratat ca având domeniul de accesibilitate interiorul clasei.

Domeniile de accesibilitate sunt imbricate : un identificator declarat într-un anume domeniu va putea fi utilizat în toate subdomeniile acestuia.

Observație: Fiecare identificator trebuie să fie unic în domeniul lui de accesibilitate.

Exemplu:

```
// analizati urmatorul exemplu
void Swap( int *ia, int, int);
void Sort( int *ia, int sz);
void PutValues( int *ia, int sz);
int ia[]={4,7,0,9};
const sz=sizeof(ia)/sizeof(int);

void main()
{
   int i,j;
   Swap(ia,i,j);
   Sort(ia,sz);
   PutValues(ia,sz);
}
```

O variabilă locală poate reutiliza numele unor variabile globale. În acest caz, variabila globală este ascunsă de cea locală. Operatorul domeniu permite evitarea acestei situatii.

Exemplu:

```
const max=65000;
void F(int max)
{
   int c;
   for( int i=0; i<max; ++i )      // max local
   {
   ...
      if( c>= ::max ) break;       // max global
   }
}
```

O variabilă globală este vizibilă întregului program. Ea va fi definită într-un singur fişier sursă, iar pentru a putea fi folosită şi în alte fişiere sursă, trebuie declarată în fiecare folosind cuvântul cheie `extern`. Vizibilitatea unui identificator global poate fi limitată la fişierul sursa în care este definit folosind cuvântul cheie `static`.

Exemplu:

```
extern int var; // in alt fisier sursa este definit int var;
static void Swap( int *ia, int i, int j)
                // in alt fisier nu poate opera
{
   int temp=ia[i];
   ia[i]=ia[j];
   ia[j]=temp;
}
extern void PutValues( int*, int);
// <extern> folosit pentru claritate, nu se obisnuieste sa se puna
```