4 Memoria liberă şi supraîncărcarea funcţiilor

În acest capitol vom aborda în detaliu două concepte fundamentale:

- alocarea memoriei pe parcursul execuţiei programului
- utilizarea aceluiaşi nume pentru mai multe funcţii.

4.1 Alocarea dinamică a memoriei

Fiecare program are la dispoziţie o rezervă de memorie nealocată care poate fi utilizată pe parcursul execuţiei. Aceasta se va numi memoria liberă a programului.

Exemplu:

```
IntArray::IntArray( int sz )
{
   size=sz;
   ia=new int[size]; // ia va adresa o zona alocata in
   for( int i=0; i<sz; ++i ) // memoria libera
   ia[i]=0;
}
```

Obiectele alocate în memoria liberă vor fi manipulate prin intermediul pointerilor. Memoria liberă se alocă prin intermediul operatorului `new` aplicat unui specificator de tip, inclusiv nume de clasă. Se pot aloca fie câte un singur obiect, fie tablouri de obiecte. Operatorul `new` returnează un pointer de acelasi tip ca şi cel specificat.

Exemplu:

```
int *pi=new int;
IntArray *pia = new IntArray(1024); // 1024 = argument pentru
                                    // constructor
IntArray *pia2 = new IntArray; // exista un constructor fara
                               // argumente
```

Se pot aloca tablouri de obiecte adăugând după specificatorul de tip dimensiunea între `[]`. Operatorul va returna un pointer la primul element.

Exemplu:

```
#include<string.h>
char *copyString(const char *s)
{
   char *ps=new char[strlen(s)+1];
   strcpy(ps,s);
   return ps;
}

// tablou de IntArray-uri cu aceeasi dimensiune
IntArray *pia=new IntArray[someSize];
```

Alocarea memoriei libere pe parcursul execuţiei se numeşte alocare dinamică a memoriei. Alocarea memoriei în urma compilării programului se numeşte alocare statică.

Exemplu:

```
ia = alocat static
tabloul adresat de ia = alocat dinamic
```

Putem vorbi astfel de existenţă unui obiect (variabile), ca fiind perioada din timpul de execuţie a programului în care obiectului îi este alocata memorie.

- Variabilele globale au existenţă statică. Ele sunt alocate în momentul lansării programului şi rămân alocate pe toată perioada de execuţie a lui.
- Variabilele locale au existenţa locală. Ele sunt alocate în momentul activării blocului/funcţiei în care sunt definite şi rămân alocate până în momentul terminării

blocului/funcției respective. O variabilă locală statică are existența identică cu cea a variabilelor globale, deci statică.

- Obiectele alocate în memoria liberă au existență dinamică. Ele rămân alocate până când sunt explicit dealocate, prin operatorul `delete` aplicat unui pointer care adresează un obiect dinamic.

Exemplu:

```
// redimensionarea obiectelor din clasa IntArray
void IntArray::grow()
{
  int *oldia=ia; // oldia are o existenta locala dar nu si obiectul
           // pe care il adreseaza
   int oldSize=size;
   size+=size/2;
   ia=new int[size];
   // copiaza elementele adresate de oldia in noua locatie
  for(int i=0; i<olfia; ++i)
    ia[i]=oldia[i];
  // initializeaza elementele ramase cu 0
  for( ; i<size; ++i )
    ia[i]=0;
  // eliberam spatiul alocat obiectului adresat de oldia
  delete oldia;
}
```

Operatorul `delete` apelează destructorul tipului respectiv, dacă acesta exista. Din acest motiv, dealocarea tablourilor de obiecte trebuie specificată explicit.

Exemplu:

```
IntArray *pia = new IntArray[size];
delete [size] pia;
```

Aplicarea operatorului `delete` unor pointeri care nu adresează obiecte dinamice poate avea consecințe imprevizibile. Operatorul `delete` poate fi aplicat pointerilor nuli.

Exemplu:

```
void f()
{
  int i;
  char *str="desen";
  int *pi=&i;
  IntArray *pia = 0;
  double *pd=new double;
  delete str;     // periculos
  delete pi;      // periculos
  delete ia;      // corect
  delete pd;      // corect
}
```

Memoria liberă nu este infinită. În cazul epuizării ei, operatorul `new` returneaza `0`;

4.2 Supraîncărcarea funcțiilor

În general, spunem că un identificator este supraîncărcat când are 2 sau mai multe înțelesuri distincte în funcție de contextul în care este utilizat.

Exemplu:

```
static int x; // poate fi static local, sau static la nivel de fisier
```

Dacă acest context lipseşte sau este insuficient pentru a se putea alege unul din înțelesuri, se consideră că este vorba despre o ambiguitate. În limbajul nostru natural folosim de multe ori deliberat exprimări ambigue. Aceste *jocuri de cuvinte* nu sunt acceptate de compilator. Dacă din context nu rezultă clar sensul unui identificator, compilatorul ne va

semnala o eroare. În C++, supraîncărcarea funcțiilor se foloseşte pentru a reuni sub acelaşi nume anumite operații aplicate unor tipuri de date diferite.
Exemplu:
```
int max(int, int);
double max(double, double);
complex& max( const complex&, const complex& );
int i=max(j,k);
complex c=max(a,b);
```

Ambiguitatea se elimină pe baza semnăturii funcțiilor respective, valoarea returnată nu se ia în considerare. Cum să supraîncărcăm un nume de funcție? Când un nume de funcție este declarat mai mult decât o data în program, compilatorul va interpreta a 2-a declarație astfel:
1) dacă tipul şi semnătura funcției se potrivesc exact, a 2-a declarație este tratată ca redeclarare a primeia.
Exemplu:
```
void print(int*, int size);
void print(int *array, int sz);
// numele argumentelor nu sunt relevante, nu se semnaleaza eroare
```

2) dacă funcțiile au semnături egale şi tipuri diferite, a doua este interpretată ca o redeclarare eronată a primei declarații.
Exemplu:
```
unsigned int max( int*, int sz);
int max(int *ia,int);   // eroare
```

3) dacă semnăturile funcțiilor diferă fie prin tipul, fie prin numărul argumentelor, atunci cele doua funcții se consideră supraîncărcate.
Exemplu:
```
void print( int*, int );
void print( double* da, int sz);
```

4) în rezolvarea ambiguității, tipurile definite prin `typedef` nu sunt considerate tipuri noi.
Exemplu:
```
typedef char *string;
int search(string);
char *search(char *);   // eroare, redeclarare eronata, cazul 2
```

Apelul funcțiilor supraîncărcate

Se realizează printr-un proces numit *potrivirea argumentelor* (*argument matching*), prin care fiecare argument actual se compară cu argumentul formal corespunzător. Rezultatele acestor comparări pot fi:
Exemplu:
```
void print(unsigned int);
void print(char *);
void print(char);
void print(int);
```

1) potrivire totală
```
unsigned a;
print('a');     // apel print( char );
print("a");     // apel print( char* );
print(a);       // apel print( unsigned );
```

2) nepotrivire
```
int *ip;
print(ip);      // eroare
```

3) potrivire ambiguă

```
// mai mult decat o functie poate rezolva problema
unsigned long ul;
print(ul);      // eroare - ambiguitate (unsigned sau int)
```

În cazul unor nepotriviri la o prima cautare, unele tipuri numerice sunt promovate la tipuri superioare:

```
char, unsigned char, short -> int
unsigned short -> int sau unsigned int (dependent de masina)
float -> double
```

Exemplu:

Fie următoarele prototipuri de funcții:

```
ff(int);
ff(short);
ff(long);
ff('a');        // apel ff(int)
```

Prin aplicarea conversiilor standard unui argument actual,

1) orice tip numeric se va potrivi unui argument formal de tip numeric
2) tipul enumerare se va potrivi unui tip numeric (chiar şi `double`)
3) valoarea 0 se va potrivi unui tip numeric sau unui pointer
4) un pointer de orice tip se va potrivi unui argument formal de tip `void*`.

Exemplu:

Fie următoarele prototipuri de funcții:

```
ff(char *);
ff(double);
ff('a');        // apel ff(double)
```

Pentru conversia unor referințe se generează variabile temporare (vezi capitolul precedent). În cazul mai multor argumente, comparările şi conversiile se fac de la stânga la dreapta, fără reveniri.

4.3 Pointeri la funcții

Să presupunem acum că trebuie să scriem o funcție de sortare cu caracter general, `sort(aray, lowBound, highBound);`. Există însă mai multe metode de sortare (`mergeSort`, `quickSort`), care se pretează la o situație sau alta. Deci funcția noastră generalizată ar trebui să permită specificarea/schimbarea algoritmului de sortare, fără ca acest lucru să presupună modificare de cod. Rezolvarea acestei probleme stă în lucrul cu pointeri la funcții.

Cum arată un pointer la funcții şi de ce tip este?

Să luăm în considerare următoarea declarație de funcție:

```
void quickSort( int*, int, int);
```

Tipul unei funcții este dat de tipul valorii returnate şi semnătura funcției (lista de argumente). Un pointer la `quickSort` trebuie să specifice aceeaşi signatură şi acelaşi tip al valorii returnate:

```
void (*pf)(int*, int, int);
```

Şi alte funcții pot avea acelaşi tip de funcție.

Exemplu:

```
void mergeSort(int*, int, int);
void heapSort(int*, int, int);
```

Pointerii la funcții pot fi inițializați cu funcții sau pointeri de acelasi tip (`quickSort` este o expresie de tip `void (*)(int*, int, int);`)

Exemplu:

```
void (*pfv)(int*, int, int) = quickSort;
void (*pfv2)(int*, int, int);
    sau
pfv=quickSort;
pfv2=pfv;
```

Fie acum declarațiile de funcții:

```
int min(int*, int sz);
int max(int*, int sz);
```

Un pointer la aceste funcții se defineşte în felul următor:

```
int (*pfi)(int*, int);
```

Şi pentru pointerii la funcții sunt valabile toate regulile enunțate în *argument matching*.

```
void (*pfv)(int*, int, int)=0;
```

Pointerii pot adresa funcții supraîncărcate:

Exemplu:

```
void ff(char);
void ff(unsigned);

void (*pf)(char)=ff;    // corect, apel void ff(char)
void (*pfv2)(int)=ff;   // eroare, ambiguitate

void main()
{
  pfi=min;
  pfv=min;        // eroare
  pfv=pfi;        // eroare
}
```

La invocarea funcțiilor prin pointeri nu este obligatoriu operatorul de indirectare.

Exemplu:

```
extern min(int*, int);
int (*pf)(int*, int)=min;
min(ia,iaSize);
(*pf)(ia,iaSize);
pf(ia,iaSize);          // toate trei sunt forme echivalente
```

Se pot folosi şi tablouri de pointeri la funcții de acelasi tip.

Exemplu:

```
void (*SortFunc[])(int*, int, int)={quickSort, mergeSort, heapSort};
(*SortFunc[2])(ia,0,iaSize-1);  // apel heapSort
```

Un pointer la o funcție poate fi declarat ca tip returnat de o altă funcție.

Exemplu:

```
int (*ff(int))(int*, int);
                    // declara ff ca fiind o functie cu 1
                    // argument de tip int
               // ff returneaza un poiter la o functie de tip
               // (int) (*)(int*, int);
```

Notațiile pot fi substanțial simplificate prin `typedef`:

Exemplu:

```
typedef int (*PFI)(int*, int, int);
PFI ff(int);    // declaratie echivalenta a functiei ff de mai sus
```

Observație: În biblioteca standard C++ (`<new.h>`), este conținut un pointer la funcții, `void (*new_handler)();`

El este setat implicit cu 0 şi este un pointer la o funcție care returnează `void` şi nu are lista de parametri. Acest pointer permite accesul la o funcție care să fie apelată când este

epuizată memoria liberă. Dacă operatorul `new` eşuează, testează acest pointer; dacă este nenul, se apelează funcția respectivă; în caz contrar, returnează 0.

Exemplu:

```
extern void freeStoreException();
_new_handler=freeStoreException;
```

4.4 Link-Editarea şi supraîncărcarea funcțiilor

Orice identificator global nestatic trebuie să fie unic. În cazul funcțiilor supraîncărcate, fiecare identificator de funcție este modificat (intern) prin adăugarea numelui propriu-zis al unei secvențe codificate a semnăturii.

Exista 2 cazuri în care această codificare generează erori la link-editare.

1) declarația funcției este inconsistentă între fişiere

Exemplu:

```
addToken(unsigned char) { ... }    // definita intr-un fisier
extern addToken(char);             // definita in alt fisier
```

2) apeluri de funcții scrise în alte limbaje de programare (de exemplu C)

În acest caz trebuie înhibat mecanismul de codificare a numelui funcției, cu ajutorul directivei de link-editare `extern`.

Exemplu:

```
extern "sir_de_caractere" prototip_functie
extern "sir_de_caractere" { prototip functii }
extern "C" strlen( const char* );
extern "C" { double sqrt(double); double fabs(double); }
```

Observație: Această directivă poate opera numai la nivel de fişier, deci global, nu local.