1 Introducere

C++ este un limbaj de programare care a fost dezvoltat la AT&T Bell Laboratories în anii '80 de catre Bjarne Stroustroup. El reprezintă o dezvoltare a limbajului C prin introducerea unui suport pentru tipuri abstracte de date (tipuri abstracte utilizator), pentru programarea orientată pe obiecte şi alte imbunătăţiri. Prima sa implementare s-a făcut în 1985 la AT&T iar în 6 luni a devenit disponibil pe 24 de sisteme, de la PC-uri până la mainframe-uri. În prezent, există medii de programare complexe, inclusiv biblioteci de obiecte. Învăţarea acestui limbaj presupune mai mult decat invăţarea unei semantici sau sintaxe noi pentru că reprezintă un nou mod de abordare a programării.

2 Tipuri de date în C++

C++ promovează un set redus de tipuri de date, operatori care manipulează aceste date şi instrucţiuni predefinite. Forţa limbajului constă în mecanismele care permit utilizatorului să definească tipuri abstracte de date.

Tipurile de dată cunoscute sunt: constantele literale şi cele simbolice, variabilele, referinţele, tablourile.

Vom studia, în continuare, tipurile enumerare, pointer şi clasă.

2.1 Tipul enumerare

Tipul de dată *enumerare* reprezintă un set de constante intregi. Diferenţa faţă de constantele întregi constă în faptul că elementele unei enumerări nu pot fi adresate. Se declară cu cuvântul cheie `enum`:

Exemplu:

```
enum {FALSE, TRUE}; // FALSE=0, TRUE=1;
```

Implicit, elementele au valori consecutive incepând de la `0`, dar sunt posibile şi asignări explicite.

Exemplu:

```
enum { FALSE, FAIL=0, PASS, TRUE=1 };
```

Fiecare enumerare poate fi etichetată definind astfel tipuri unice de date.

Exemplu:

```
enum testStatus { NOT_RUN=-1, FAIL, PASS };
enum boolean { FALSE, TRUE };
main()
{
     const testSize=100;
     testStatus test[testSize];
     boolean found = FALSE;
     for(int i = 0; i < testSize; ++i)
          testStatus[i]=NOT_RUN;
}
```

Variabilele de tip enumerare pot fi iniţializate şi li se pot atribui numai constante din tipul respectiv.

Exemplu:

```
main()
{
     testStatus test = NOT_RUN;
     boolean found = FALSE;
```

```
test = -1;      // avertisment TestStatus = int
test = found;   //avertisment TestStatus = Boolean
test = FALSE;   // TestStatus = const Boolean
int status = test; // Ok, conversie implicita
}
```

Tipului de dată enumerare asigură faptul că o variabilă de acest tip (ex. `test`) va lua numai una dintre valorile menționate în mulțime.

2.2 Pointeri

O variabilă de tip *pointer* conține adresa unui obiect din memorie. Ea permite referirea indirectă a obiectelor. Un exemplu tipic de utilizare a pointerilor este crearea listelor înlănțuite şi administrarea obiectelor alocate pe timpul execuției programului.

Fiecare pointer are un tip asociat. Tipul este dat de tipul variabilei la care se referă. De exemplu, un pointer *int* va pointa la un obiect de tip *int*.

Spațiul care se alocă unui pointer este constant şi este cel necesar pentru a memora o adresă (2 octeți sau 4 octeți, dependent de maşină sau de modelul de memorie folosit).

Tipul pointerului arată cum trebuie interpretat conținutul şi dimensiunea zonei de memorie pe care o adresează.

Exemplu:

```
int i = 1024;
int *ip = &i;  // &  - operatorul adresa
int *ip2 = ip;
int *ip3 = i;  // eroare
```

C++ este un limbaj puternic tipizat, cu alte cuvinte, orice inițializare şi asignare este verificată la compilare. Tipurile implicate trebuie fie să corespundă, fie să fie obiectul unor reguli de conversie.

La dorința explicită a programatorului, există posibilitatea unei conversii explicite a tipului de dată (inclusiv pentru pointeri), numita *cast*. Iată două din regulile de conversie pentru pointeri:

- li se poate asigna valoarea `0` (`NULL`)
- li se pot aisgna *pointeri generici* (*void**)

Pentru a accesa un obiect prin intermediul unui pointer trebuie aplicat operatorul de indirectare *.

Exemplu:

```
int i=1024;
int *ip = &i;
int k = *ip;    // k are val 1024
int k = ip;     // eronat
*ip = k;        // i=k;
*ip = abs(*ip);// i=abs(i);
*ip = *ip+1;    // i=i+1;
```

Aritmetica pointerilor

Valoarea unui pointer (adresa conținută) poate fi adunată sau scăzută cu o valoare întreagă `i`. O asemenea operație modifică adresa cu dimensiunea a `i` obiecte de tipul pointerului si NU cu `i` octeți.

Exemplu:

```
char *cp; cp+2      ->    adresa + 1*2octeti   (1 char = 1 octet)
int *ip; ip+2       ->    + 2*2octeți   (1 int = 2 octeți)
double *dp; dp+2    ->    + 4*2octeți   (1 double = 4 octeți)
```

Pointerii cel mai des utilizați sunt pointerii la şiruri de caractere, `char*`.

Tipul şir de caractere (`char*`)

În C++, manipularea şirurilor de caractere se face prin pointeri. Tipul unui şir constant este un pointer la primul caracter din şir. Putem defini o variabilă de tip `char*` şi s-o initializăm cu un şir:

Exemplu:

```
char *st="Cultivarea spiritului\n";
```

Următorul program inţentionează să calculeze lungimea lui `st`, folosind aritmetica pointerilor pentru avansare în şir.

Exemplu:

```
// determinarea lungimii unui sir
# include<iostream.h>
char *sir="Cultivarea spiritului\n";

     main()
{
     int lungime = 0;
     char *pSir = sir; // pointer de manevra
     while(*pSir++ != '\0')
          ++lungime;
     cout << lungime << ":" << sir;
     return 0;
}
```

Să modificăm acest program astfel încât munca noastră să poată fi folosită şi de altcineva. Plasăm codul într-o functie separată, astfel el devine disponibil pentru oricine.

Exemplu:

```
int stringLength( char *sir)
     {
// returneaza lungimea lui sir
     int lungime = 0;
     while( *sir++ )
          ++lungime;//echivalent cu *sir++ !=0
     return lungime;
}
```

Exemplu:

```
// utilizarea externa a acestei functii
     #include<iostream.h>
char *sir = "Cultivarea spiritului\n";
int stringLength(char *);

void main()
{
     int lungime = stringLength(sir);
     cout << lungime << ":" << sir;
}
```

Funcţia `stringLength(char *)` nu modifică pointerul `st`, ci doar o copie locală a acestuia, care dispare în momentul terminării funcţiei. În această situaţie, spunem că `st` este transmis prin valoare funcţiei `stringLength`.

2.3 Tipuri abstracte de date. Tipul clasă

Pentru a putea lucra cu noțiuni complexe, mintea umană recurge adeseori la *abstractizare*. Aceasta înseamnă separarea calitățile esențiale ale unei idei sau obiect de detaliile legate de modul în care lucrează sau de alcătuire.

Prin abstractizare, ne focalizăm asupra întrebării *ce*, dar nu şi pe *cum*. De exemplu, gradul nostru de înțelegere a funcționării unui autoturism se bazează în mare măsură pe abstractizare. Mulți dintre noi ştiu *ce* face motorul unui autoturism, dar puțini ştiu sau vor să ştie *cum* lucrează un motor. Abstractizarea ne permite să discutăm despre automobile şi să le folosim, fără să ştim, însă, detalii legate de modul lor de funcționare.

În lumea proiectării software, este recunoscut faptul că abstractizarea este o necesitate absolută pentru coordonarea proiectelor complexe, foarte mari. În cursurile introductive, programele sunt de obicei mici, de 50-200 de linii de cod şi inteligibile în întregul lor de către o singură persoană. Pe de altă parte, produsele software comerciale alcătuite din sute de mii sau milioane de linii de cod nu pot fi realizate fără folosirea abstractizării în diverse forme. Există două importante tehnici de abstractizare: *abstractizarea controlului* şi *abstractizarea datelor*.

Abstractizarea controlului înseamnă separarea proprietăților logice ale unei acțiuni de implementarea sa. Procedăm la abstractizarea controlului atunci când scriem o funcție care reduce un algoritm complicat la o acțiune abstractă realizată printr-un apel de funcție. Prin invocarea unei funcții de bibliotecă:

```
4.6 + sqrt(x)
```

ne vom folosi de numele funcției fără a fi nevoie să îi cunoaştem implementarea.

Tehnicile de abstractizare se aplică, de asemenea, datelor. Fiecare tip de dată constă dintr-un set de valori (domeniul) şi o colecție de operații permise asupra lor. Abstractizarea datelor intervine atunci când avem nevoie de un tip de dată care nu este implementat în limbajul de programare folosit de noi. Definim noul tip de dată ca un *tip abstract de dată*, concentrându-ne doar pe peoprietățile sale logice şi făcând abstracție de detaliile legate de implementare.

Ca şi în cazul abstractizării controlului, şi pentru datele abstracte avem o specificație (acel *ce*) şi o implementare (*cum*). Specificația unui tip abstract de dată descrie caracteristicile valorilor datei şi modul în care funcționează fiecare operație asupra acestor valori. Pentru a îl putea folosi, utilizatorul unui tip abstract trebuie să înțeleagă doar specificația, nu şi implementarea. Iată o specificare neformală a unui tip abstract:

TIPUL
    IntArray
DOMENIUL
    Fiecare IntArray conține o colecție de numere întregi. Colecția poate avea oricâți întregi, dar implicit sunt 24.
OPERAȚII
    Atribuirea unui IntArray altui IntArray (atribuire între tablouri)
    Indexarea (returnează un element din IntArray folosind indicele său)
    Returnează dimensiunea tabloului (IntArray-ului)

Se remarcă absența totală a detaliilor de implementare. Nu am menționat cum sunt stocate datele sau cum se implementează operațiile. Evitând aceste detalii, reducem complexitatea pentru utilizatorii clasei şi le creăm o independență de modificările de implementare.

Specificarea unui tip abstract de dată defineşte valorile datelor şi operațiile. În final, un tip abstract de dată trebuie implementat în program. Pentru aceasta, programatorul trebuie să țină cont de două lucruri:

- să aleagă o reprezentare concretă a datelor abstracte, folosind tipuri de dată deja existente
- să implementeze fiecare operație în termenii unor instrucțiuni de program.

Pentru a implementa tipul abstract de dată IntArray, putem alege o reprezentare concretă a datelor constând din două elemente: un tablou de numere întregi şi o variabilă întreagă care păstrează dimensiunea tabloului. Pentru a implementa operațiile din IntArray, trebuie să creăm nişte funcții bazându-ne pe reprezentarea concretă a datelor.

În C++, un tip de dată introdus de utilizator se numeste *clasă*. O clasă este un ansamblu format din elemente de diferite tipuri şi din funcțiile destinate manipulării lor. De regulă, o clasă introduce un nou tip, care dacă este bine proiectat, poate fi ulterior la fel de comod de utilizat ca şi tipurile predefinite. Tipul clasă va fi descris în detaliu ulterior. Fiind conceptul central în limbajele orientate pe obiecte, va fi introdus în cele ce urmează printr-un exemplu detaliat: **o clasă de tablouri întregi**. De ce o clasă de tablouri întregi când avem tipul predefinit de tablouri întregi? Motivele pentru care tipul predefinit de tablouri întregi nu este suficient şi care au dus la nevoia introducerii unei clase de tablouri întregi sunt urmatoarele:

- dimensiunea tablourilor trebuie să fie o constantă;
- incadrarea indicilor în dimensiune nu este verificată;
- când sunt transmise ca parametri unor funcții, tablourile trebuie însoțite de dimensiunea lor;
- nu sunt suportate atribuirile la nivel de tablou;

Să analizăm împreună următorul exemplu:

Exemplu:

```
const int arraySize = 24; // dimensiune implicita
     class IntArray // clasa de tablouri intregi
{
public:
     IntArray ( int sz=araySize ); // constructor
     IntArray ( const IntArray& ); // constructor
     ~IntArray () { delete ia; } // destructor
     IntArray& operator = ( const IntArray& );
                    //operator de atribuire
     int& operator [] (int index ); // operator de indexare
     int getSize () { return size; }
                    //returneaza dimensiunea tabloului
protected:
     int size; // dimensiunea tabloului
     int *ia; // adresa tabloului de intregi
};
```

Definiția unei clase constă din 2 părți:

- *antetul clasei* (*class head*), compus din cuvântul cheie `class` urmat de numele clasei
- *corpul clasei* (*class body*), compus din definiția ei cuprinsă între `{...}` urmată de `;`

Numele clasei este numele unui nou tip de dată. Exemple de utilizare a acestui nou tip ar fi:

Exemplu:

```
     const int sz=10;
     int mySize;
     int ia[sz]; // tip predefinit de tablou
     IntArray myArray(mySize), ia(sz);
     IntArray *pA = &myArray;
     IntArray iA2; // 24 elemente implicit
```

Corpul claselor conține definițiile membrilor, şi anume operațiile permise şi datele necesare pentru a reprezenta clasa. Cuvintele cheie `protected` şi `public` controlează accesul la membrii unei clase. (*information hiding*). Membrii din secțiunea `public` sunt accesibili întregului program.

Membrii din secțiunea `protected` sunt accesibili doar din funcțiile membre în clasa respectivă. În general, datele unei clase sunt protejate pentru că:

- schimbându-le trebuie modificate doar funcțiile membre, nu şi programele utilizator
- se restrânge aria erorilor de manipulare la funcțiile membre.

Observați în definiția clasei cele 3 funcții care au acelasi nume, mai exact numele clasei. În cadrul unei clase distingem următoarele funcții membre speciale:

- *constructorii* (au numele clasei)
- *destructorii* (au numele clasei precedat de caracterul ~).

Aceste funcții sunt apelate automat de către compilator. Cum arată un constructor şi care sunt sarcinile sale? Constructorul se ocupă de inițializarea datelor membre ("construieşte obiectul"). Constructorul `IntArray( int sz=arraySize);` este apelat pentru declaratiile obişnuite, de tip `IntArray iA2;` unde `arraySize` reprezintă dimensiunea implicită.

Exemplu:

```
IntArray :: IntArray ( int sz )
     {
          size = sz;
          ia = new int[size];
                    // aloca un tablou de intregi de
                         //dimensiune size şi depune adresa
                    sa în
                         //pointerul ia;
          for (int i=0; i<size; ++i ) // se initializeaza
tabloul
                    ia[i] = 0;
     }
```

Ce noutăți au apărut în acest cod? Operatorul `new` este un operator pentru alocarea dinamica a memoriei. Operatorul `::` este operatorul domeniu şi indică în acest caz că funcția este membră a clasei `IntAray`. Funcțiile membre pot accesa direct datele clasei.

Prin operatorul `.` se accesează membrul unei anumite clase (cunoscătorii limbajului C îşi vor aminti analogia cu accesul membrilor unei structuri). Operatorul `->` se foloseşte în acelaşi scop, doar că lucrează printr-un pointer la clasa respectivă.

Deoarece funcția `getSize()` are o definiție foarte simplă, ea a fost definită odată cu clasa. Prin convenție, definiția clasei şi toate constantele implicate sunt date printr-un fişier prefix, având acelaşi nume ca şi clasa, iar definițiile funcțiilor membre într-un fişier C++ cu acelaşi nume ca şi clasa. Pentru clasa `IntArray` vom avea fişierele `IArray.h` şi `IArray.cpp`.