Crash Course in C++

Un tutorial practic despre C++ destinat programatorilor de Java

Adrian Scoică
Observații și comentarii: adrian.scoica@gmail.com

15 februarie 2011

Cuprins

Legendă:
(*) informații cu caracter general
(**) anumite aspecte de detaliu
(***) facilități avansate și mai rar folosite ale limbajului

1 Introducere

1.1 Ce este C++?

Multă lume definește C++ ca fiind un "superset de C". Această definiție este parțial corectă, dar eșuează în a conferi limbajului credit pentru inovațiile și gradul de expersivitate superioare pe care le aduce față de limbajul C.

C++ a fost creat de către Bjarne Stroustrup pe baza limbajului C, pentru a suporta noile concepte vehiculate în programare la vremea respectivă, cunoscute colectiv sub numele de "Programare Orientată pe Obiecte". Limbajul a fost proiectat să permită exprimarea conceptelor specifice programării orientate spre obiecte, fiind construit să suporte clase, virtualizare, supraîncărcare, moștenire multiplă, template-uri și tratarea excepțiilor, printre altele, trăsături care l-ar fi distanțat în mod normal de C încă de la început. Cu toate acestea, familiaritatea programatorilor cu C-ul, nevoia de compatibilitate inversă cu codul deja existent, precum și statutul incontestatbil al C-ului au condus la dezvoltarea unui limbaj care să cuprindă în gramatica sa aceleași construcții sintactice.

C++ este unul dintre cele mai populare limbaje de programare scrise vreodată, fiind cel puțin la fel de eficient și portabil ca C-ul, dar complexitatea crescută cât și libertatea de exprimare (amintită cel mai adesea sub forma "libertatea de a greși") foarte mare au încurajat dezvoltarea de limbaje simplificate sau specializate care să implementeze doar parțial funcționalitățile acestuia și să se concentreze pe alte aspecte.

Cele mai cunoscute limbaje care au fost puternic influențate de C++ sunt Java, respectiv C#.

1.2 C++ sau Java?

Uneori se pune întrebarea: Este C++ mai bun sau mai rău decât Java? Adevărul este că un răspuns la această întrebare, din păcate, nu există. În principiu, C++ pune la dispoziția programatorului mult mai multă libertate de exprimare, dar acest lucru nu este întotdeauna ceva bun. Mai degrabă, întrebarea ar face parte din aceeași clasă (no pun intended) cu întrebarea "Este Engleza mai bună sau mai rea decât Româna?".

Este destul de evident că atât Româna cât și Engleza au în calitate de limbi vorbite, aceeași putere de exprimare. Unele idei sunt mai ușor de exprimat în Română, altele sunt mai înrădăcinate în Engleză, dar în definitiv nu există nici un concept ce se poate exprima într-una din limbi fără să se regăsească printre construcțiile celeilalte un echivalent.

În general, în Java s-au sacrificat o parte considerabilă dintre funcționalitățile C++ din dorința de a grăbi ritmul de dezvoltare al aplicațiilor de către pro-

gramatori, precum și din dorința de a standardiza mai ușor protocoalele folosite. Motivul principal stă în faptul că Java a fost inițial conceput pentru a realiza aplicații end-user bazate pe accesul la Internet, în timp ce C++ se folosește mai mult pentru a realiza aplicații de tip back-end (servere) sau aplicații care efectuează intens calcule și deci trebuie optimizate cât mai bine.

1.3 C++ vs Java. Avantaje și dezavantaje comparative.

În încheierea acestei secțiuni, voi trata comparativ câteva dintre diferențele majore dintre C++ și Java, pregătind în același timp mindset-ul necesar parcurgerii următoarelor capitole. Din dorința de a mă exprima cât mai eficient, voi considera știute aspectele legate de Java și mă voi concentra mai mult asupra celor specifice C++. Cu toate acestea, tutorialul de față nu este și nu se dorește a fi un substitut pentru un manual de C++. Mai degrabă, considerați-l un "cheat sheet" care să vă introducă în limbaj, astfel încât să vă ajute să scrieți cod corect și mult mai eficient. Pentru câteva resurse dedicate de C++, verificați secțiunea de bibliografie.

Recomand parcurgerea următoarelor pagini, mai ales dacă Java este primul limbaj de Programare Orientată Obiect pe care l-ați stăpânit. Este posibil să nu realizați la o primă lectură importanța unora dintre diferențe, dar vă veți lovi relativ repede de ele dacă veți încerca să traduceți efectiv sintaxa de Java și veți constata că de fapt codul nu funcționează așa cum v-ați fi așteptat.

1.3.1 Compilat vs. interpretat

C++ este, ca și C, un limbaj compilat. Compilatorul parsează codul sursă și produce la ieșire cod binar în formatul mașinii fizice pe care se compilează (ex: executabile Linux pe 32bit). Avantaje: executabilele sunt imagini de procese pentru mașina gazdă. Aplicațiile scrise în C++ au toate funcționalitățile celorlaltor aplicații native ale sistemului: un PID, fisiere de I/O (stdin, stdout, stderr), și accesul direct la toate celelalte resurse ale sistemului de operare (de exemplu: sistemele de fișiere, socketi, memoria virtuală, apeluri de sistem), linkarea cu biblioteci partajate pentru încărcarea dinamică de cod la rulare, etc.

Compilatorul de **Java**, pe de alta parte, produce cod intermediar care trebuie interpretat. Cu alte cuvinte, după compilarea Java, bytecode-ul obținut este rulat în interiorul unei mașini virtuale. Avantaje: deoarece aplicațiile scrise în Java nu rulează nativ în mașina fizică, ci execuția lor este "simulată" de Java Runtime Environment, acest lucru eradichează practic problemele de compatibilitate între platforme, permite încărcarea dinamică a claselor la runtime, existența blocurilor statice de inițializare și mecanismul de garbage collection.

1.3.2 Timpul de execuție

Compilatoarele de C++ sunt, în general, capabile să producă cod binar extrem de optimizat. Mai mult, faptul că acest cod se rulează nativ pe mașina gazdă reduce la minimum overhead-ul. De asemenea, C++ nu vă adaugă singur în cod funcționalități suplimentare pe care nu le-ați cerut și care vă trag aplicația înapoi. Nu în ultimul rând, C++ permite, opțional, specificarea de directive care să instruiască compilatorul despre cum să vă genereze codul de asamblare.

Dacă nici asta nu este suficient, puteți oricând să introduceți în proiectul vostru de C++ implementări de funcții critice scrise direct în limbaj de asamblare (care daca sunt scrise bine, pot accelera semnificativ aplicația). Deși ar putea părea un gest extrem, circumstanțele speciale impun măsuri speciale. Nu este recomandat totuși să introduceți în C++ cod de asamblare, pentru că pierdeți portabilitatea (și categoric nu înainte de finalizarea aplicației).

Viteza de execuție a fost mereu primul și cel mai important dezavantaj pentru Java. Deoarece bytecode-ul trebuie interpretat de mașina virtuală, acest lucru înseamnă că pentru fiecare operație elementară de bytecode, mașina fizică ar trebui teoretic să execute două instrucțiuni: una pentru interpretare și una pentru execuție. În cazul noilor mașini virtuale, acest lucru nu este valabil, și performanțele aplicațiilor Java s-au îmbunătățit constant de la apariția limbajului: dacă la început, o aplicație Java rula de până la de 20 de ori mai lent decât echivalentul C++, acum s-a ajuns la un un factor de timp de aproximativ 1,5 - 2.

Lupta pentru performanță a dus la apariția mașinii virtuale Java HotSpot, scrisă în C++ (:D). Ideea din spatele HotSpot este cu adevărat state-of-the-art în ceea ce privește știința compilatoarelor: crearea de optimizări adaptive ale codului sursă în funcție de statisticile de runtime ale codului, și poate coduce în cazuri particulare la cod la fel de rapid ca și echivalentul nativ C++ fără optimizări specifice. Alte inovații cu adevărat valoroase aduse de mașinile virtuale Java sunt Just-In-Time compilation, Garbage Collection sau Class Data Sharing.

1.3.3 Memoria și accesul la memorie

Gestiunea memoriei este poate ultima diferența cu adevărat importantă dintre C++ și Java. În C++, executabilul final este imaginea unui proces. Cu alte cuvinte, veți avea acces la un spațiu de memorie virtuală propriu, din care puteți aloca/citi orice, conform cu nevoile aplicației (ca în C). La fel ca în C, puteți accesa memoria fie prin intermediul unu obiect, fie prin numele variabilei care ocupă acea zonă de memorie, fie prin intermediul unui pointer. Alocarea, eliberarea, precum și accesul corect la memorie sunt în întregime **responsabilitatea programatorului**. Mai multe despre asta veți înțelege din ciclul de viață al obiectelor. Mai mult, C++ pune la dispoziție și facilități mai avansate

de acces la memorie:

- referințele: sunt doar dubluri de nume pentru obiecte existente în spațiul de memorie, și funcționează puțin diferit decât în Java: ele trebuie inițializate la instanțiere, și se comportă precum obiectul referențiat. Nu le puteți distruge în mod explicit, și nici atribui, dar vă scapă de o parte din greșelile lucrurlui cu pointeri.

- iteratorii: iteratorii în C++ sunt pur și simplu clase care abstractizează un pointer (permit operații precum (*it) sau it++ prin supraîncărcarea operatorilor specifici). Spre deosebire de Java, iteratorii sunt strongly-typed și mai intuitiv de folosit, dar la fel ca și pointerii, vă pot crea probleme dacă nu sunteți atenți.

- smart-pointerii: există doar în C++, deoarece în Java nu aveți acces la memorie. Funcționează ca niște pointeri obișnuiți, cu mențiunea că atribuirea se face prin "transferul" complet al obiectului referențiat, pentru a evita memory leak-urile. Cel mai probabil nu veți avea prea curând de a face cu ei.

Deoarece gestiunea memoriei în C++ este foarte complicată, și este probabil sursa numărul 1 de erori pentru programele scrise în acest limbaj, Java a venit cu o soluție radicală: nu vi se permite accesul la memorie! Toate obiectele sunt manipulate prin referințe (mai apropiate ca funcționalitate de pointerii din C++ decât de referințele de acolo). Se tolerează și tipuri de dată scalare (int, double), dar acestea nu pot fi integrate în colecțiile de date ale limbajului. Acest lucru poate fi uneori extrem de inconvenabil, pentru că duce la scrierea de cod foarte ineficient! De asemenea, Java vă descurajează să contați pe ciclul de viață al obiectelor, și nu vă lasă să controlați consumul de memorie (Garbage Collecterul este nedeterminist). Aparent, acest lucru este convenabil pentru programator, dar în realitate vă obligă să apelați in logica aplicației cod explicit pentru eliberarea resurselor (fișiere, coenxiuni), ceea ce poate complica lucrurile.

Totuși, decizia Java de a nu vă permite accesul la memorie are și alte justificări. Deoarece codul se execută în mașina virtuală, memoria nu este mapată ad literam în RAM, iar dacă se dorea accesul la memorie, atunci ar fi trebuit simulat acest lucru. De asemenea, dacă vi s-ar fi permis accesul la memorie, atunci ați fi putut compromite transparența serializării și se pierdea din ușurința limbajului de a scrie aplicații network-oriented. În cele din urmă, finalități diferite justifică decizii de implementare diferite!

2 Clase

2.1 Declarație

Cel mai important concept din OOP este conceptul de clasă. Desigur, toată lumea care a programat în Java are deja o intuiție despre ce anume e o clasă, dar nu este corect să confundăm un limbaj cu paradigma pe care trebuie să o exprime. Eu propun să uităm o clipă de programare ca să putem gândi mai elegant: **Ce este în realitate o clasă?**

În principiu, cred că vom fi cu toții de acord că o clasă descrie o mulțime de lucruri care au ceva în comun. Asemănările dintre două obiecte din aceeași clasă pot să fie:

- structurale (oamenii au neuroni)
- comportamentale (oamenii gândesc)

Cu alte cuvinte, dacă știi despre un obiect că se încadrează într-o clasă, atunci știi să îl descrii la nivel general.

Haideți să reținem acestă idee și să revenim la C++. În C++, o clasă seamnănă mult cu un tip de date. Ca să folosim o clasă, trebuie mai întâi să o descriem. Putem face asta în două moduri: folosind keyword-ul **class** sau folosind keyword-ul **struct**.

```
1  class Student{
2
3          char * name;
4          double grade[3];
5
6          void setMark(int subject, double newMark);
7  }; // ATENTIE! Nu uitati de ";". Aceasta este o greseala foarte frecventa!
8
9  // sau asa:
10
11 struct Student{
12
13         char * name;
14         double grade[3];
15
16         void setMark(int subject, double newMark);
17 }; // ATENTIE! Nu uitati de ";". Aceasta este o greseala foarte frecventa!
```

2.2 Modificatorii de acces

Ca și în Java, membrii din interiorul unei clase, fie ei date sau funcții, pot avea modificatori de acces:

- **public** - Datele sau funcțiile se pot folosi atât din interioul cât și din afara clasei.
- **private** - Datele sau funcțiile se pot folosi doar în interiorul clasei. Clasele care o derivă nu au acces la ele.
- **protected** - Datele sau funcțiile pot fi folosite în interiorul clasei curente și al tuturor claselor care o derivă.

În C++, există o singură diferență între **class** și **struct**: nivelul de acces implicit, care este

- **private** - pentru clasele declarate cu **class**
- **public** - pentru clasele declarate cu **struct**

În Java, accesul trebuie specificat pentru fiecare membru în parte. În C++, este suficient să scriem cuvântul cheie o singură dată urmat de doua puncte și el rămâne valabil până la întâlnirea unui alt modificator de acces, sau până la finalul clasei. De exemplu, pentru clasele de mai sus, putem modifica accesul la membri adăugând doar puțin cod:

GREȘEALĂ FRECVENTĂ! Dacă veți încerca să scrieți ca în Java, înainte de numele membrului fără ":" veți primi eroare de compilare!

```
1  class Student{
2
3      char * name; //private (implicit pt class)
4  protected:
5      double grade[3]; //protected
6  public:
7      void setMark(int subject, double newMark); //public
8
9  };
10
11 // sau de exemplu cu struct
12
13 struct Student{
14
15     char * name; //public (implicit pt struct)
16 private:
17     double grade[3]; //private
18     void setMark(int subject, double newMark); //also private
19
20 };
```

2.3 Constructori

Atunci când vrem să descriem o clasă, un aspect important este să definim [cel puțin] un constructor pentru acea clasă. Ca și în Java, constructorii nu returnează valori și au numele identic cu al clasei.

De reținut este că dacă nu declarăm un constructor pentru o clasă, atunci compilatorul va considera din oficiu un constructor care nu face nimic (nu are nici o instrucțiune).

În exemplul anterior, ar fi bine să ne putem asigura cumva în program că la orice moment de timp, numele unui student este un pointer valid dacă este privit din afara clasei (pentru ca daca reușim asta, niciodată nu o să avem SegFault folosindu-l). Soluția din C era să creăm o funcție care inițializează o structură, dar în C++ **singurul mod corect** de a face asta este să declarăm constructori.

A folosi o funcție separată de inițializare ar însemna să avem încredere că oricine ar declara vreodata un obiect de tip **Student** va apela conștiincios funcția înainte de a folosi obiectul. Asta e cea mai sigura cale catre ore în șir de debugging.

```
1  class Student{
2  //private: este implicit, pentru ca am folosit "class"
3          char * name;
4          double grade[3];
5  public:
6          void setMark(int subject, double newMark);
7
8          Student(); // Constructor fara parametri
9          Student(char * name); // Stim numele dinainte
10 };
```

2.4 Definire

Chiar dacă nu am amintit decât pe scurt despre câteva din conceptele relevante pentru declararea claselor în C++, relativ la Java acestea ar trebui să fie suficiente pentru a avea o înțelegere elementară a sintaxei. Vom trece în continuare la definirea de clase.

Ce este, totuși definirea unei clase și cum diferă ea de declarare? Putem să ne gândim astfel:

- **Declararea** unei clase reprezintă o descriere generală a acesteia. Compilatorul află că există tipul de dată corespunzător, știe ce metode sunt specifice și ce variabile ar trebui să încapsuleze clasa, cum este ea înrudită cu alte clase, dar nu știe nimic concret despre clasă.
- **Definirea** unei clase, pe de altă parte, este procesul prin care se detaliază exact codul din metode care trebuie executat, dar și construirea membrilor statici în memorie. Un alt termen echivalent pentru definire este **implementare**.

În Java, clasele erau organizate în pachete, iar metodele erau definite obligatoriu în același fișier cu declarația clasei din care făceau parte. Acest mod de a scrie codul poate părea mai simplu din anumite puncte de vedere, dar este destul de inflexibil, motiv pentru care în C++ este folosit doar în situații speciale pe care le vom aminti mai târziu.

Abordarea corectă în C++ este de a separa declarația și definiția unei clase în fișiere distincte:

- **Headere (*.h, *.hpp)**, care conțin numai declarații
- **Implementări (*.cpp)**, care conțin numai definiții

Avantajul evident este că putem înlocui rapid o implementare scurtă dar ineficientă a unei clase cu una mai lungă dar mai optimizată fără a trebui să mai editǎm codul, specificând doar alt fișier la compilare. Pe de altă parte, separându-le, putem evita erorile de definire multiplă atunci când avem lanțuri mai complexe de includeri.

Pentru a defini o metodă dintr-o clasă, trebuie să prefixăm metoda cu numele clasei, urmat de operatorul de rezoluție de scop (::). Facem acest lucru pentru a informa compilatorul că de fapt definim o metodă dintr-o clasă, și nu o funcție globală cu acel nume. Un exemplu valoreaza cât o mie de cuvinte.

Pentru clasa următoare:

```
// Student.h

#pragma once

#ifndef __STUDENT_H__
#define __STUDENT_H__

class Student{
//private: este implicit, pentru ca am folosit "class"
        char * name;
        double grade[3];
public:
        void setMark(int subject, double newMark);

        Student(); // Constructor fara parametri
        Student(char * name); // Stim numele dinainte
};

#endif
```

Un exemplu de implementare ar putea fi următoarul:

```
// Student.cpp

#include <cstring> // Pentru strdup()

#include "Student.h"

// din clasa Student, definim functia Student()
Student::Student(){
        name = strdup("Name Unknown");
}

// din clasa Student, definim Student(char*)
Student::Student(char * name){
        // ATENTIE! In C++, "this" este un pointer, nu o referinta!
        this->name = strdup(name);
}

// din clasa Student, definim setMark(int, double)
void Student::setMark(int subject, double newMark){
        grade[subject] = newMark;
}

```

În declarația de mai sus, ați remarcat două elemente noi, care nu sunt obligatorii în orice situație, dar care au devenit "standard practice" în C++:

- **#pragma once** - această directivă este specifică C++ și informează compilatorul să nu includă același fișier de mai multe ori într-o ierarhie de clase.
- **#ifndef ... #endif** - această directivă este specifică C și îndeplinește același rol. De obicei se pun amândouă din motive de compatibilitate cu compilatoarele care nu înțeleg una din directive, dar o interpretează corect pe cealaltă.

O altă observație importantă se referă la constructori. Practic în exemplul de mai sus, constructorii inițializează membrii claselor prin atribuiri. Acest mod de a scrie codul seamănă cu sintaxa din Java, dar nu funcționează mereu. În mod corect, în C++, constructorii sunt niște funcții speciale care se apelează automat înainte ca un obiect să înceapă efectiv să existe, deci trebuie invocați corespunzător. Acest lucru se specifică folosind operatorul ":" dupa antetul constructorului, urmat de o serie de construcții separate de virgulă. Rescriind fișierul de mai sus, obținem:

```
1  // Student.cpp
2
3  #include <cstring> // Pentru strdup()
4
5  #include "Student.h"
6
7  // invocarea corectă a constructorului pentru membrul "name"
8  Student::Student() : name(strdup("Name Unknown"))
9  {
10 }
11
12 // invocarea corectă a constructorului pentru membrul "name".
13 // Compilatorul știe să facă diferența între:
14 // * membrii clasei, pe care se apelează constructorii
15 // * parametri, care pot să apară doar în paranteze ca valori
16 Student::Student(char * name) : name(strdup(name))
17 {
18 }
19
20 // din clasa Student, definim setMark(int, double)
21 void Student::setMark(int subject, double newMark){
22         grade[subject] = newMark;
23 }
24
```

Nu în ultimul rând, C++ vă lasă să definiți o clasă în momentul declarării (ca în Java). Mai mult, acest stil de definire devine obligatoriu pentru anumite optimizări (funcții inline, respectiv headere precompilate). Acest mod de definire poartă tot numele de "inline". Clasa de mai sus definită inline ar arăta astfel:

```
1  // Student.h
2
3  #pragma once
4
5  #ifndef __STUDENT_H__
6  #define __STUDENT_H__
7
8  #include <cstring>
9
10 class Student{
11         char * name;
12         double grade[3];
13 public:
14         void setMark(int subject, double newMark) {
15             grade[subject] = newMark;
16         }
17
18         Student() : name(strdup("Name Unknown")) { }
19         Student(char * name) : name(strdup(name)) { }
20 };
21
22 #endif
```

ATENTIE! În C++ nu puteți reapela alt constructor din interiorul unui constructor pentru un obiect (cum se întâmpla în Java, când scriați this(0,0), de exemplu). De asemenea, nu există cuvântul cheie "super", mecanismul de instanțiere pentru clasele derivate fiind explicat mai jos, la capitolul aferent.

2.5 Ciclul de viață al obiectelor. Destructori.

Ciclul de viață al obiectelor este probabil aspectul cel mai divergent al programării orientate pe obiecte, așa cum este ea exprimată în C++, respectiv Java. În Java există mai multe tipuri de obiecte, unele cu scope explicit (de exemplu, scalarii: int, double sau referințele în sine) și altele fără scope explicit, care se crează cu operatorul "new" și a căror spațiu ocupat este "reciclat" de către garbage collecter în momentul în care nu mai sunt referite din nici un thread.

În ciuda aparențelor, în C++ gestionarea memoriei este mult mai simplă conceptual (dezavantajul fiind faptul că o mare parte din ea cade în sarcina programatorului).

În C++, **orice** variabilă este declarată într-un scope. Exemple de scope-uri sunt:

- Spațiul unei funcții/metode
- Spațiul global de nume
- Interiorul blocurilor de program (cuprinse între { })

În momentul în care se iese din interiorul unui scope, trebuie eliberată stiva. Pentru fiecare obiect care este scos din stivă, C++ apelează automat o funcție specială care se numește **destructor**. Destructorul oferă obiectului șansa de a elibera resursele pe care le ținea ocupate. Ca și în cazul constructorilor, dacă nu definim nici un destructor, C++ generează din oficiu un destructor vid pentru clasele pe care le scriem.

Câteva situații în care ar putea fi nevoie de destructori sunt:

- Atunci când clasa conține resurse pe care le-a alocat dinamic și care trebuie dezalocate.
- Atunci când clasa a deschis fișiere pe care trebuie să le închidă.
- Atunci când clasa a deschis socketi pe care trebuie să îi închidă.
- Atunci când clasa deține lock-uri pe care trebuie să le elibereze (nu veți întâlni la materiile din anul 2).

Destructorii se definesc asemănător cu constructorii, doar că numele lor trebuie precedat de caracterul tildă și nu pot primi parametri! De asemenea, deși limbajul permite, în teorie nu este corect să apelați niciodată explicit destructori! Ei se apeleaza automat la ieșirea din scop a obiectelor, sau la dezalocarea memoriei alocate dinamic.

Vom reveni din nou la exemplu pentru a explica mai bine conceptul:

```cpp
#include <iostream>
#include <cstring>

class Student{
        char * name;
        double grade[3];
public:
        void setMark(int subject, double newMark) {
            grade[subject] = newMark;
        }

        char * getName() {
            return name;
        }

        Student() : name(strdup("Name Unknown")) { }
        Student(char * name) : name(strdup(name)) { }
};

void functie1(){
    // In declaratia de mai jos se apeleaza constructorul fara parametrii pentru a
    // crea un obiect care este valabil incepand de aici pana la sfarsitul functiei.
    Student student1;

    std::cout << student1.getName() << "\n"; // Se afiseaza "Name Unknown"
}

void functie2(Student student2){
    // student2 este un obiect care se creaza in momentul apeului si este valabil
    // pana la sfarsitul functiei.
    std::cout << student2.getName() << "\n"; // Se afiseaza "Eric Cartman"
}

int main()
{
    functie1();
    functie2(Student("Eric Carman"));
    return 0;
}
```

Pe parcursul execuției acestui program, se vor construi, pe rând, două obiecte de tip **Student**. Fiecare instanțiere se face cu apelarea constructorului pentru clasa Student, iar constructorii aloca memorie în care copie un șir de caractere care să reprezinte numele studentului. Dar ce se întâmplă cu această memorie? Nu apare nicăieri nici apel către "free". În realitate, această memorie nu se eliberează odată cu obiectele (se distruge doar pointerul, nu și memoria pe care

o pointează), ci produce leak. Pentru a rezolva această problemă, vom declara în clasă un destructor care să elibereze memoria atunci când obiectele ies din scop. Programul corect este:

```
#include <iostream>
#include <cstring>
#include <cstdlib>

class Student{
        char * name;
        double grade[3];
public:
        void setMark(int subject, double newMark) {
            grade[subject] = newMark;
        }

        char * getName() {
            return name;
        }

        Student() : name(strdup("Name Unknown")) { }
        Student(char * name) : name(strdup(name)) { }

        // Cand obiectul trebuie distrus, se executa codul din destructor!
        ~Student() { free(name); }
};

void functie1(){
    // In declaratia de mai jos se apeleaza constructorul fara parametrii.
    Student student1;
    std::cout << student1.getName() << "\n"; // Se afiseaza "Name Unknown"
    // AICI se apeleaza destructorul pentru student1. Variabila e locala functiei.
}

void functie2(Student student2){
    // student2 este un obiect care se creaza in momentul apeului.
    std::cout << student2.getName() << "\n"; // Se afiseaza "Eric Cartman"
    // AICI se apeleaza destructorul pentru student2. Parametrul e local functiei.
}

int main()
{
    functie1();
    functie2(Student("Eric Carman"));
    return 0;
}
```

2.6 Membrii statici

În C++, membrii statici ai claselor sunt resurse pe care le au în comun toate obiectele clasei. La fel ca în Java, ei se declara folosind cuvântul cheie **static** în fața membrului ce trebuie **declarat** (NU definit), și funcționează asemănător, dar există câteva deosebiri.

- Variabilele membre statice terbuie **definite** în fișierele de implementare. Acest lucru este necesar deoarece prin simpla declarare, lor nu li se aloca efectiv memorie în segmentul de date!
- Spre deosebire de Java, nu este posibilă crearea unor blocuri statice de inițializare. Cel mai bine este să considerați că la începutul execuției programului, variabilele statice au valori nedefinite (sau, în anumite compilatoare, initializate cu "0"). Variabilele statice de tip clasă sunt construite automat folosind un constructor fără parametri, dacă nu se specifică altfel în cod.
- Funcțiile statice, dacă au permisiunile corecte, pot fi folosite ca orice alte funcții globale, și se pot deduce pointeri din ele astfel încât să poată fi date ca parametri în locul funcțiilor globale. Pentru funcțiile membre nonstatice, acest lucru NU este valabil din cauza încapsulării. Această problemă nu apărea în Java oricum, deoarece acolo nu putem avea pointeri la funcții.

Pentru a apela o funcție statică, trebuie să îi specificăm și spațiul de nume (prefixând apelul cu numele clasei, urmat de operatorul ::).

Următorul exemplu ilustrează folosirea de membrii statici.

```
1 #include <iostream>
2 #include <cstring>
3 #include <cstdlib>
4
5 class Student{
6         char * name;
7
8         // Declaram un obiect static
9         static int instanceCount;
10 public:
11     // Declaram o functie statica
12         static void printInstanceCount() {
13             std::cout << instanceCount << "\n";
14         }
15
16         Student() : name(strdup("Name Unknown")) {
17             instanceCount++;
18         }
19
```

```
20          Student(char * name) : name(strdup(name)) {
21              instanceCount++;
22          }
23
24          ~Student() {
25              free(name);
26              instanceCount--;
27          }
28  };
29
30  // DEFINIM fizic obiectul instanceCount (i se va aloca memorie)
31  // si il initializam explicit cu 0 la lansarea in executie
32  int Student::instanceCount = 0;
33
34  void functie()
35  {
36      Student s2;
37      Student::printInstanceCount(); // Se afiseaza 2: "s1" si "s2"
38      // AICI se apeleaza destructorul lui "s2", care decrementeaza instanceCount
39  }
40
41  int main()
42  {
43      Student s1;
44      Student::printInstanceCount(); // Se afiseaza 1: "s1"
45      functie();
46      Student::printInstanceCount(); // Se afiseaza 1: "s1"
47      return 0;
48      // AICI se apeleaza destructorul lui "s1", care decrementeaza instanceCount
49  }
50
```

2.7 Derivarea claselor

Printre cele mai importante avantaje ale programării orientate pe obiecte îl reprezintă mecanismul de moștenire. Din nou, acesta este un capitol la care există diferente notabile între C++ și Java.

Paradigma impusă de Java este aceea că "Totul este un obiect". Prin urmare, absolut orice clasă pe care am scrie-o derivă (sau folosind terminologia Java, "extinde") din clasa **Object**. Mai mult, în Java o clasă nu poate extinde decât o singură altă clasă. Cu alte cuvinte, dacă avem o clasă care extinde java.util.Vector, atunci ea nu poate extinde nici o altă clasă. Din cauza că asta ar fi limitat limbajul foarte tare, cei care au proiectat Java au permis doar un singur tip de moștenire multiplă: moștenirea de interfețe. Diferența este că o interfață cuprinde doar funcții și de aceea nu pot apărea conflicte de instanțiere multiplă a membrilor de date.

Pe de altă parte, în C++ nu există nici o restricție din acest punct de vedere, iar lucrurile stau mult mai complicat (deși este extrem de improbabil să aveți de-a face cu altceva decât derivare de tip direct și public).

Pentru a deriva o clasă, este suficient să înșiruiți clasele pe care aceasta le extinde după numele clasei, cu virgulă între ele. Numele claselor de bază ("părinte") trebuie precedat cu tipul de derivare. În acest sens, derivarea publică este aproape universală.

O excepție notabilă este că în C++, nu există o clasă de bază din care să derive toate celelalte clase (echivalentă cu Object din Java)! Unul dintre motivele pentru care în Java există clasa Object este tratarea uniformă a tipurilor, dar în C++ există pentru acest scop pointerii generici **(void*)**.

Un exemplu de sintaxa prin care se specifica derivarea de clase regăsim în continuare:

```
1  #include <iostream>
2  #include <cstring>
3  #include <cstdlib>
4
5  class Om {
6  protected:
7      char * nume;
8  public:
9      Om(const char * nume) : nume(strdup(nume)) { }
10     virtual ~Om() { free(nume); }
11 };
12
13 class Student : public Om {
14     double nota;
```

```
15 public:
16     Student(const char * nume, double nota) : Om(nume), nota(nota) { }
17     void display() {
18         std::cout << "(" << nume << "," << nota << ")\n";
19     }
20 };
21
22 int main()
23 {
24     Student s("Eric Cartman", 10.0f);
25     s.display();
26     return 0;
27 }
```

În exemplul de mai sus, există trei elemente de sintaxă care trebuie explicate:

- Apelarea constructorului clasei de baza in constructorul clasei Student, pe linia 16. Un Student este, în primul rând, un caz particular de Om, deci înainte ca obiectul Student să poată fi explicit construit, trebuie să apelăm constructorul clasei de bază folosind sintaxa specifică constructorilor. Acest lucru este echivalent cu apelurile către apelurile la constructori de tipul **super()** din Java.
- Se remarcă de asemenea că a trebuit să declarăm numele ca fiind protected, pentru ca funcția **display()** din clasa Student să aibă acces la el.
- Nu în ultimul rând, destructorul din clasa de bază este declarat ca fiind **virtual**. În acest caz, acel cuvânt cheie nu produce diferențe, dar în cazul general este **extrem** de recomandat ca destructorii claselor să fie declarați ca virtuali. Vom discuta acest lucru la virtualizare.

Atunci când o clasă derivează altă clasă, aceasta moștenește din clasa de bază toți membrii, mai puțin membrii statici. Trebuie de ținut minte că membrii statici nu participă la mecanismul de moștenire, deoarece acest lucru nu are sens (ei sunt partajați doar între obiectele unei anumite clase, dar nu se pot replica pentru clasele derivate). Setarea drepturilor de acces astfel încât clasele derivate să aibă acces la ei crează uneori o falsă impresie de moștenire, dar aceasta este o greșeală de exprimare des întâlnită la programatori.

Un alt lucru care trebuie reținut este că în C++, supradefinirea unei metode dintr-o clasă de bază duce cu sine la pierderea **tuturor** metodelor cu același nume moștenite din clasa de bază, indiferent de semnătura parametrilor. Metodele din clasele de bază se pot apela prin specificarea namespace-ului corect (numele clasei, urmat de operatorul ::).

Ca și curiozitate, ar trebui menționat că în C++ derivarea poate să fie și de tip protected/private (sunt rare situațiile când așa ceva este cu adevărat necesar în cod), dar și virtuală (pentru soluționarea problemei moștenirii "în romb").

2.8 Polimorfism. Funcții virtuale

Polimorfismul în C++ este la început o sursă continuă de confuzie pentru programatorii Java. În Java, un singur comportament este posibil: toate funcțiile sunt implicit polimorfice. Cu alte cuvinte, dacă o clasă derivează o clasă de bază și suradefinește o metodă, atunci în toate situațiile în care folosim o referință la obiecte derivate, se va apela noua metodă.

În realitate, acest comportament impune să ținem evidența tipului real al tuturor obiectelor instanțiate, și încetinește foarte mult toate apelurile de funcție din nevoie întreținerii unei tabele de virtualizare.

Prin urmare, în C++ virtualizarea nu se face automat. Ea trebuie specificată în mod explicit cu ajutorul cuvântului cheie **virtual** în declarație (NU definiție). Odată definită ca fiind virtuală o metodă dintr-o clasă rămâne virtuală și în clasele derivate, chiar dacă nu mai specificăm acest lucru în noile declarații.

Cel mai clar este să folosim un exemplu care să justifice efectele lipsei virtualizării:

```
1  #include <iostream>
2  #include <cstring>
3  #include <cstdlib>
4
5  class Animal {
6  protected:
7      char * name;
8  public:
9      void makeSound() {
10         std::cout << "No sound\n";
11     }
12     Animal(const char * name) : name(strdup(name)) { }
13     virtual ~Animal() { free(name); }
14 };
15
16 class Bird : public Animal {
17 public:
18     void makeSound() {
19         std::cout << "Cra-Cra\n";
20     }
21     Bird(const char * name) : Animal(name) { }
22 };
23
24 class Dog : public Animal {
25 public:
26     void makeSound() {
27         std::cout << "Whoof-Whoof\n";
```

```
28      }
29      Dog(const char * name) : Animal(name) { }
30  };
31
32  void forceToSpeak(Animal * animal){
33      // Se va afisa mereu "No sound", deoarece compilatorul nu poate decide
34      // tipul real al obiectului la care se refera pointerul "animal" daca nu
35      // specificam noi sa tina evidenta printr-o tabela de virtualizare
36      animal->makeSound();
37  }
38
39  int main()
40  {
41      Bird bird("Tweetie");
42      Dog dog("Pluto");
43      std::cout << "// Apeluri directe: \n";
44      bird.makeSound();
45      dog.makeSound();
46      std::cout << "// Apeluri prin pointeri: \n";
47      forceToSpeak(&bird);
48      forceToSpeak(&dog);
49      return 0;
50  }
51
```

În exemplul de mai sus, putem forța compilatorul să țină evidența tipului real folosind cuvântul cheie **virtual** în fața metodei **makeSound()**. De asemenea, din moment ce nu vom avea niciodată o instață de Animal în program, după ce facem metoda makeSound virtuală, putem să îi scoatem implementarea default și să o facem **pur virtuală**. O metodă pur virtuală este o metodă fără implementare, ceea ce face clasa respectivă să fie abstractă. Exact ca și în Java, o clasă abstractă nu se poate instația. Putem face o metodă să fie pur virtuală specificând în cadrul declarației că nu i se va atașa nici un pointer de funcție (este o funcție cu pointerul NULL). Astfel, este suficient să modificăm următoarele în clasa Animal:

```
1  class Animal {
2  protected:
3      char * name;
4  public:
5      // Am declarat metoda makeSound ca fiind virtuala si am specificat
6      // ca aceasta clasa nu o poate implementa folosind "= 0;"
7      // Acest lucru face din Animal o clasa abstracta.
8      virtual void makeSound() = 0;
9      Animal(const char * name) : name(strdup(name)) { }
10     virtual ~Animal() { free(name); }
11 };
```

Este recomandat ca destructorii claselor să fie mereu declarați ca virtuali, deoarece altfel este posibil ca atunci când vrem să distrugem un obiect referențiat de un pointer să nu se apeleze destructorul corect, ci destructorul unui părinte al clasei, producând astfel comportamente neașteptate în program.

2.9 Supraîncărcarea operatorilor

Un motiv pentru care C++ este mai expresiv și produce cod mai clar decât Java îl reprezintă posibilitatea de a supraîncărca operatori. Supraîncărcarea operatorilor se referă la interpretarea semnificației lor în funcție de tipurile operanzilor. Această facilitate este foarte puternică în C++, permițând implementarea de către utilizatori a grupurilor algebrice direct în limbaj! Deși supraîncărcarea operatorilor nu necesită mecanisme de parsare foarte complicate, ea nu a fost implementată mai departe în Java din motive de claritate a codului final. Supraîncărcarea operatorilor exista totuși intr-o formă rudimentară și în C, unde operatorul plus, de exemplu, are semnificație diferită în funcție de tipul operanzilor:

- **(int)(1)+(int)(2) == (int)(3)**, pe când
- **(int*)1+(int)2 == (int*)(1+sizeof(int)*2)**

În C++, se poate supraîncărca oricare dintre operatorii existenți ai limbajului: **unari** (, !, ++, etc.) sau **binari** (+, -, +=, -=, ==, <, etc.). Prin supraîncărcare, vom defini o funcție care să fie invocată automat atunci când întâlnim la stânga și [eventual] la dreapta tipurile de date corespunzătoare. Din motive de consistență a limbajului, nu se pot redefini operatori care există deja bine definiți, de exemplu operatorul "+" între două numere întregi va însemna mereu numai adunare.

Ceea ce putem redefini însă sunt operatorii pe tipuri de date definite de utilizator. Cel mai des supraîncărcați operatori sunt operatorii **mai mic** și **atribuire**. Trebuie reținut că supraîncărcarea operatorilor nu afectează prioritățile lor. Atunci când dorim o ordine anume de aplicare a operațiilor, este indicat să folosim paranteze pentru siguranță. O pereche de paranteze se tastează în sub o secundă, dar lipsa lor poate provoca ore întregi de debugging.

Operatorii în C++ seamănă din punct de vedere sintactic (și semantic) cu niște funcții. Ei pot fi definiți la nivel global, sau pot fi încapsulați în clase. Cu toate acestea, a doua practică nu este încurajată deoarece ascunde niște probleme de implementare (deoarece operandul din stânga este implicit obiectul referit de **this**).

Pentru a exemplifica sintaxa prin care se pot supraîncărca operatorii în C++, ne vom folosi de un exemplu matematic foarte simplu: vom defini o clasă Complex care să rețină numere complexe și vom implementa operațiile cu numere complexe folosind operatorii nativi din limbaj, în loc de funcții.

```
1 #include <iostream>
2
3 class Complex{
4 public:
5     // Membrii de date
```

```
6     double real, imag;
7     // Constructori
8     Complex(double real, double imag) : real(real), imag(imag) { }
9     Complex() : real(0), imag(0) { }
10 };
11
12 // Spunem ce executa "+" intre doua obiecte Complex
13 Complex operator+ (Complex& left, Complex& right){
14     return Complex(left.real+right.real, left.imag+right.imag);
15 }
16
17 // Spunem ce executa "+" intre un obiect Complex si un obiect int
18 Complex operator+ (Complex& left, int right){
19     return Complex(left.real+right, left.imag);
20 }
21
22 // Spunem ce executa "+=" intre doua obiecte Complex
23 Complex& operator+= (Complex& left, Complex& right){
24     left.real += right.real;
25     left.imag += right.imag;
26     return left;
27 }
28
29 // Spunem ce executa "<<" intre un obiect ostream si un obiect Complex
30 std::ostream& operator<< (std::ostream& out, Complex right){
31     out << "(" << right.real << "," << right.imag << ")";
32     return out;
33 }
34
35 int main()
36 {
37     Complex a(1,1), b(-1,2);
38     std::cout << "A: " << a << "\nB: " << b;
39     std::cout << "\nA+B: " << (a+b) << "\n";
40 }
41
```

Atunci când scriem operatori este bine să încercăm să îi privim mai degrabă ca pe niște aplicații matematice pentru a nu greși. Se observă din exemplul de mai sus că operatorii **+=** și **<<** întorc referințe către obiectul din stânga. Vă veți întreba de ce acest lucru nu este redundant.

Răspunsul se află în modul în care compilatorul interpretează codul. Noi scriem **std::cout << a << b;** și ne-am aștepta ca programul să execute pe rând, codul de afișare pentru obiectul **a** și apoi pentru obiectul **b** pe ecran. În realitate însă, ceea ce se execută arată mai degrabă în felul următor (ținând cont că or-

dinea de evaluare este de la stânga la dreapta): **operator<<(operator<<(std::cout, a), b)**. Dacă operatorul << nu ar fi returnat mai departe referința către obiectul de tip ostream în care se face afișarea, atunci nu am fi putut îmbrica apelurile către operatori, sau cu alte cuvinte nu am fi putut înlănțui operatorii. Înlănțuirea este mai ales relevantă atunci când scriem operatori pentru expresii matematice, deoarece este un caz particular destul de rar ca o expresie matematică să conțină un singur operator. De altfel, atunci când s-au studiat operatori la algebră, acești operatori erau de fapt funcții care luau valori în mulțimea peste care era definită algebra.

Dintre toți operatorii, doi sunt mai des supraîncărcați și pun probleme mai serioase programatorilor:

- Operatorul =, numit și **assigment operator**
- Operatorul <

Operatorul de atribuire se sintetizează din oficiu de către compilator dacă nu îl supraîncărcați voi explicit. Operatorul din oficiu pur și simplu realizează atriburiea între toți membrii din clasă (proces numit "shallow copy"). Acest lucru devine problematic atunci când aveți pointeri în clase, deoarece riscați memory leak-uri la fiecare atribuire. În acest caz, va trebui să implementați voi explicit atribuirea. Un amănunt destul de important este de asemenea că operatorul de atribuire nu trebuie confundat cu copy constructor-ul unei clase, chiar dacă se pot nota amândoi cu simbolul =. Vom discuta despre copy constructor la momentul potrivit. De asemenea, sunt situații în care nu vi se poate sintetiza operatorul de atribuire pentru o clasă, deocare clasa conține referințe ca membrii (iar referințelor în C++ nu li se pot atribui valori după instanțiere).

Operatorul < este important mai ales pentru că el este cerut implicit de anumite containere din biblioteca standard de template-uri. El are rol similar cu funcția din interfața Comparable din Java, doar că în C++ verificarea existenței operatorilor corecți se face abia la linking. Cu alte cuvinte, dacă folosiți o structură de date sortată (de exemplu, un "map" în care cheile sunt obiecte definite de voi), va trebui în mod obligatoriu să definiți operatorul < pentru chei, altfel nu veți putea compila codul. Pentru a respecta semnătura impusă de containerele din C++, este de asemenea necesar ca operatorul < să primească argumente de tip "const", pentru că altfel nu se garantează corectitudinea algoritmului de sortare. Lipsa specificării parametrilor de tip "const" va duce la erori de compilare.

Un alt operator interesant este operatorul funcțional **()**, care permite obiectelor să fie apelate ca funcții. Aceste obiecte poartă numele de **functori** și nu au nici un concept echivalent în Java. Functorii pot face anumite programe greu de înțeles, dar sunt interesanți din punct de vedere al logicii limbajului, deoarece

șterg practic granița între clase și funcții.

În final, ar trebui amintit că o serie dintre coding guide-urile consacrate (de exemplu, cel al Google sau cel al Mozilla) sfătuiesc vehement împotriva supraîncărcării operatorilor limbajului, doarece semnificația acestora poate deveni ambiguă foarte repede. Excepție fac în principiu operatorii de atribuire și cei logici (declarați cu cuvântul cheie **const** imediat după lista de argumente. Această sintaxă restricționează implementarea să nu producă efecte laterale (un concept despre veți mai auzi mai ales la programarea funcțională). Vă veți convinge repede că abuzul de facilitățile C++ poate conduce dacă nu sunteți atenți la cod imposibil de întreținut.

3 Input/Output

3.1 Fișiere și stream-uri

În C, metoda cea mai folosită de I/O era folosirea de fișiere. Pentru a simplifica lucrul cu fișiere, C-ul definea o structură wrapper peste descriptorii de fișier pe care să o folosim (structura **FILE**). În C++, însă, s-a dorit abstractizarea lucrului cu resursele de I/O. Astfel, deși se pot în continuare folosi funcțiile și structurile din C, C++ pune la dispoziție stream-uri.

Stream-urile din C++ seamănă conceptual (și de fapt, constituie sursa de inspirație) pentru cele din Java, dar există câteva diferențe care le separă.

- Stream-urile din biblioteca standard derivă toate fie din clasa **ostream** (pentru fluxuri de ieșire), fie din clasa **istream** (pentru fluxuri de intrare).
- Este posibil lucrul cu fișiere instanțiind obiecte fie din clasa **ofstream** (pentru fișiere de ieșire), fie din clasa **ifstream** (pentru fișiere de intrare).
- Deși operațiile din C++ pot arunca excepții, veți observa că spre deosebire de Java, C++ nu vă obligă să tratați excepțiile. Acest lucru poate fi și bun și rău. Dacă alegeți să nu folosiți excepțiile de la streamuri, ele vor fi ignorate și codul va continua să se execute (eventual) defectuos. Pentru compatibilitate, C++ setează și o serie de flag-uri care sunt moștenite pentru toate stream-urile din clasa părinte comună **ios**. Cele mai folosite flag-uri sunt: eof, fail și bad. Semnificațiile lor variază în funcție de tipul de stream folosit și este indicat să verificați documentația, mai ales la început.

Pentru cele trei fișiere clasice care sunt deschise automat pentru orice program care se execută, C++ definește trei stream-uri cu nume consacrate:

- **std::cin** - stream-ul definit peste **stdin**
- **std::cout** - stream-ul definit peste **stdout**
- **std::cerr** - stream-ul definit peste **stderr**

Implicit, toate operațiile cu stream-uri în C++ sunt buffer-ate (acest lucru se poate dezactiva cu ajutorul unei funcții handler). Din acest motiv, este foarte greșit să folosiți simultan pentru afișare sau citire și stream-uri și fișiere, deoarece vi se vor încăleca rezultatele și efectul final nu va fi cel dorit.

Afișarea și citirea în/din stream-uri se face în mod convențional cu operatorii << pentru scriere și >> pentru citire.

Nu în ultimul rând, trebuie reținut că obiectele de tip "stream" sunt făcute în mod intenționat neasignabile în C++ (a fost omisă intenționat definirea operatorului de atribuire). Motivul pentru care nu se pot atribui stream-uri este riscul duplicării bufferului. Există metode prin care puteți manipula bufferele, dar este improbabil să aveți cu adevărat de lucrat cu așa ceva (v-ar putea folosi doar la redirecționări forțate, dar puteți rezolva altfel, folosind pointeri).

În următorul exemplu de cod, deschidem două fișiere, unul de intrare și unul de ieșire, și efectuăm câteva operații elementare de I/O.

```
1  #include <iostream>
2  #include <fstream>
3
4  int main()
5  {
6      // Instantiem si deschidem simultan doua stream-uri:
7      // * in, pentru citire
8      // * out, pentru scriere
9      std::ifstream in("input.txt");
10     std::ofstream out("output.txt");
11
12     // Verificam daca s-a deschis "out" cu succes
13     if (out.fail()) {
14         std::cerr << "ERROR: Nu s-a putut deschide fisierul \"output.txt\"\n";
15         return 0;
16     }
17
18     int a, b;
19     // Citim doi intregi din "in"
20     in >> a >> b;
21
22     // Inchidem "in"
23     in.close();
24
25     // Atasam lui "in" alt fisier de intrare
26     // (pentru fisiere binare, folosim ifstream::bin)
27     in.open("other_input.txt", std::ifstream::in);
28
29     // Scriem a si b in "out", cu niste text suplimentar
30     out << "A este: " << a << " iar B este: " << b << "\n";
31
32     // Inchidem toate fisierele
33     in.close();
34     out.close();
35
36     return 0;
37 }
```

```
38
39
```

Un ultim aspect care ar trebui tratat aici și de care s-ar putea să aveți nevoie la proiect este serializarea. În C++, mecanismul de serializare/deserializare nu se face automat ca în Java, deci va trebui să folosiți stream-uri binare și să supraîncărcați voi operatorii <<, respectiv >> conform cu protocolul pe care vreți să îl implementați.

4 Șiruri de caractere

4.1 Clasele std::string și std::stringstream

C++ pune la dispoziție o clasă nativă pentru lucrul cu șiruri de caractere, și anume clasa **string**. Această clasă există, desigur și în Java, unde funcționează aproximativ similar. Pentru clasa **string** sunt supraîncărcați câtiva operatori mai intuitivi:

- Indexare: [] - permite accesul la caracterul de pe o anumită poziție
- Adunare: += - permite adăugarea unui șir/caracter la sfârșitul unui string
- Egalitate: == - funcționează similar cu funcția strcmp()
- Atribuire: = - permite copierea string-urilor

Avantajul evident la string-uri este faptul că acestea pot fi folosite mult mai intuitiv decât șirurile de caractere, și nu mai este nevoie să ținem cont de terminatorii de șir sau de problemele de realocare de memorie, deoarece acestea sunt tratate automat în interiorul claselor. De asemenea, folosind operatorii supraîncărcați, codul arată mai aerisit și mai intuitiv, ceea ce face implementarea mai rapidă și mai elegantă.

Spre deosebire de Java însă, C++ nu are suport nativ pentru caracterele Unicode sau UTF-8. Toate caracterele sunt reținut în mod ASCII. Există separat tipul de date **w_char** pentru tipuri de caractere wide (pe doi Bytes), iar biblioteca standard pune la dispoziție inclusiv macro-uri pentru cast-ul implicit, dar greșeli se pot face frecvent. Mai mult, o serie de clase template acceptă alocatori ca parametru în constructori, ceea ce înseamnă că de fapt ele pot funcționa cu orice tip de caractere am vrea noi să definim dacă scriem codul necesar manipulării tipului de date. Ca și în alte situații, C++ are un preț de plătit pentru libertatea de alegere pe care o oferă programatorului: mai multă bătaie de cap!

Cu toate acestea, este extrem de improbabil să aveți nevoie de caractere Unicode în aplicațiile de algoritmică, deci nu trebuie să vă faceți griji cu privire la encoding, ci puteți să contați pe faptul că este mereu ASCII.

Clasa **stringstream** este folosită în C++ pentru a stoca un buffer de caractere. De obicei ea este utilă atunci când vrem să generăm stringuri prin "afișări" repetate din diferite obiecte. În Java, puteam face conversia automat folosind o construcție de tipul (**""+object**), dar în C++ acest cod nu are efectul așteptat. Pentru serializare trebuie să folosim operatorul << pentru a printa informațiile într-un stringstream, și apoi putem folosi funcția membru **str()** fără parametrii pentru a recupera șirul generat din stream. Dacă trimitem un parametru în membrul **str()**, acesta are efectul de a înlocui stringul din stream.

În continuare avem un exemplu orientativ de folosire al acestor clase:

```
1 #include <iostream>
2 #include <string>  // Clasa string
3 #include <cstring> // Functiile pe stringuri din C
4 #include <sstream>
5
6 int main()
7 {
8     // Instantiem un string:
9     std::string s("Ana are mere");
10
11    // Adaugam la sfarsit un sufix:
12    s += ". Eric Cartman ";
13
14    // Daca caracterul de pe pozitia 3 este spatiu, afisam lungimea sirului si sirul
15    if (s[3] == ' ') {
16        std::cout << "Lungimea " << s.length() << ": " << s << "\n";
17    }
18
19    // Obtinem continutul din "s" sub forma de string în stil C
20    // c_str() intoarce un pointer la o zona statica de memorie, deci daca vrem sa
21    // pastram string-ul, trebuie sa folosim strdup() ca sa il clonam
22    char * ps = strdup(s.c_str());
23    char * toFree = ps;
24
25    // Cream un stream dintr-un string
26    std::stringstream ss;
27
28    // Impartim ps in cuvinte si le scriem in ss, separate prin underscore
29    ps = strtok(ps," ");
30    while (ps){
31        ss << ps << "_";
32        ps = strtok(NULL, " ");
33    }
34
35    // Afisam continutul din ss pe ecran
36    std::string rep = ss.str();
37    std::cout << "Lungimea " << rep.length() << ": " << rep << "\n";
38
39    // Distrugem ps;
40    delete[] toFree;
41
42    return 0;
43 }
```

5 Gestiunea memoriei

5.1 Alocarea memoriei: new vs. malloc

În C, alocarea memoriei se făcea în mod obișnuit prin intermediul funcției **malloc** (sau alternativ se puteau folosi și **calloc** sau pentru șiruri de caractere, **strdup**). Deși funcțiile din bibliotecile de C sunt păstrate aproape integral în C++, folosirea lui **malloc** nu este suficientă pentru alocarea de noi obiecte, în acest scop definindu-se operatorul **new** (care a fost implementat și în Java mai apoi).

Atunci când folosim operatorul **new**, se execută în spate două lucruri distincte:

- Se alocă memorie suficientă pentru a stoca obiectele cerute, conform cu spațiul ocupat de clasă
- Se invocă automat constructorii pentru obiectele nou create

Valoarea returnată de operator este un pointer către zona de memorie care conține obiectul sau obiectele nou alocate.

Deja ar trebui să fie evident motivul pentru care un simplu apel la **malloc** nu este suficient pentru a crea un obiect dintr-o clasă: chiar dacă se alocă memorie suficientă, problema este că nu este invocat constructorul pentru obiectele respective. Dacă vom lua drept exemplu un obiect de tipul std::vector pe care încercăm să îl alocăm dinamic cu **malloc**, în realitate noi nu facem decât să rezervăm memorie pentru un obiect de tipul std::vector, dar în acel spațiu nu se va construi efectiv un astfel de obiect, ci vom avea valori neinițializate. Încercarea ulterioară de a insera un element în acea colecție va eșua aproape sigur, din diferite motive posibile:

- pointerii interni clasei nu au la rândul lor memorie alocată, deoarece rezervarea de memorie s-ar fi făcut în constructor
- variabilele care țin de starea internă a obiectului nu au valori consistente (de exemplu, capacitatea maximă rezervată, numărul de elemente din colecție, etc.)

Singura metodă corectă de a aloca dinamic un obiect sau un array de tip structural este prin intermediul operatorului **new**. Se pune atunci, evident, întrebarea de ce au mai fost păstrate funcțiile de alocare de memorie din C. Răspunsul nu ține doar de compatibilitate. Funcțiile din C de alocare de memorie se pot folosi în continuare pentru alocări de buffere binare sau de obiecte care nu au constructori expliciți (acolo unde nu există oricum nevoie de a executa cod în momentul instanțierii). Așa cum v-ați obișnuit deja, limbajul vă dă

libertatea să folosiți ce vreți și nu vă constrânge în nici un fel, dar scrieți totul pe propria răspundere.

În continuare am dat ca exemplu câteva exemple de folosire a operatorului **new**.

```
#include <iostream>

class Complex{
public:
    double real, imag;
    Complex(double real, double imag) : real(real), imag(imag) { }
    Complex() : real(0), imag(0) { }
};

std::ostream& operator<< (std::ostream& out, Complex& right){
    return out << "(" << right.real << "," << right.imag << ")\n";
}

int main()
{
    // Alocam un intreg dinamic si citim o valoare de la tastatură
    int * n = new int;
    std::cin >> (*n);

    // Cream un vector de n numere complexe
    // Pentru toate elementele din array vom invoca constructorul
    // fara parametri!
    Complex* v = new Complex[(*n)];

    // Citim n numere Complexe si le afisam apoi
    int real, imag;
    for (int i = 0; i < (*n); i++){
        std::cin >> real >> imag;
        v[i] = Complex(real,imag);
    }

    for (int i = 0; i < (*n); i++){
        std::cout << v[i];
    }

    // Nu dezalocam memoria inca, vom trata acest lucru in capitolul urmator.

    return 0;
}
```

5.2 Dezalocarea memoriei: free vs. delete

Dezalocarea memoriei alocate dinamic în C se făcea prin intermediul funcției **free**. Așa cum deja intuiți, simplul apel către **free** nu este suficient pentru a dezaloca memoria în C++, din motive similare cu motivele pe care le-am citat la alocarea de memorie. Pentru a dezaloca memorie, în C++ există operatorul **delete**, care primește ca operand un pointer catre o zonă de memorie alocată dinamic.

Diferența dintre **free** și **delete** este aceea că dezalocarea unui obiect cu **free** nu face decât să elibereze spațiul ocupat de obiectul în sine și atât, în timp ce dezalocarea cu **delete** invocă în mod automat și destructorul obiectului. Acest lucru este relevant, de exemplu, dacă în interiorul clasei există pointeri la memorie alocată dinamic care trebuie de asemenea distrusă și care ar fi leak-uit dacă foloseam **free** și nu se apela destructorul.

Din aceleași rațiuni ca și în secțiunea precedentă, **free** nu este nicidecum o funcție obsolete, ci are aplicativitate în alte aspecte ale limbajului. Ca regulă generală:

- tot ceea ce este alocat cu **new** trebuie dezalocat cu **delete**
- tot ceea ce este alocat cu **malloc** (și alte funcții similare din C) trebuie dezalocat cu **free**

De remarcat aici că atunci când vrem să dezalocăm un array și nu un singur obiect, sintaxa corectă este **delete**[]. În continuare, adăugând codul pentru dezalocarea de memorie la exemplul de la punctul precedent, se obține:

```
1  #include <iostream>
2
3  class Complex{
4  public:
5      double real, imag;
6      Complex(double real, double imag) : real(real), imag(imag) { }
7      Complex() : real(0), imag(0) { }
8  };
9
10 std::ostream& operator<< (std::ostream& out, Complex& right){
11     return out << "(" << right.real << "," << right.imag << ")\n";
12 }
13
14 int main()
15 {
16     int * n = new int;
17     std::cin >> (*n);
18     Complex* v = new Complex[(*n)];
19
```

```
20      int real, imag;
21      for (int i = 0; i < (*n); i++){
22          std::cin >> real >> imag;
23          v[i] = Complex(real,imag);
24      }
25
26      for (int i = 0; i < (*n); i++){
27          std::cout << v[i];
28      }
29
30      // Asa se dezaloca un singur obiect (avem nevoie de pointerul sau)
31      delete n;
32      // Asa se dezaloca un array (avem nevoie de pointerul primului element)
33      delete[] v;
34
35      return 0;
36  }
```

6 Standard Template Library

6.1 Introducere: de ce STL?

Standard Template Library (sau pe scurt, **STL**) este o bibliotecă generică de C++, parțial inclusă în C++ Standard Library. Ea reprezintă una din modalitățile de a folosi cod gata implementat, și conține implementări de template-uri pentru o serie de structuri de date și algoritmi mai uzuali. Buna stăpânire a noțiunilor legate de STL vă va ajuta să dezvoltați rapid implementări la fel de scurte și mult mai eficiente decât cele din Java.

Asta face efortul învățării folosirii STL să pară mic în comparație cu avantajele obținute. Pentru început, veți observa că puteți citi valori, sorta vectori, și alte operații de rutină într-un mod mult mai sigur și mai eficient, scriind un singur rând de cod, iar în timp veți descoperi cum operații mai complexe precum folosirea de cozi ordonate sau căutarea în arbori binari pot fi făcute la fel de ușor și sigur.

Pentru a sprijini afirmațiile făcute mai sus, vom porni de la un exemplu simplu: un program care citește valorile unui vector, le sortează și le afișează pe ecran. Prima variantă este o implmentare clasică de C.

```
1  #include <stdlib.h>
2  #include <iostream.h>
3
4  // a and b point to integers. cmp returns -1 if a is less than b,
5  // 0 if they are equal, and 1 if a is greater than b.
6  inline int cmp (const void *a, const void *b)
7  {
8          int aa = *(int *)a;
9          int bb = *(int *)b;
10         return (aa < bb) ? -1 : (aa > bb) ? 1 : 0;
11 }
12
13 // Read a list of integers from stdin
14 // Sort (c library qsort)
15 // Print the list
16
17 main (int argc, char *argv[])
18 {
19         const int size = 1000; // array of 1000 integers
20         int array [size];
21         int n = 0;
22         // read an integer into the n+1 th element of array
23         while (cin >> array[n++]);
24         n--; // it got incremented once too many times
25
```

```
26          qsort (array, n, sizeof(int), cmp);
27
28          for (int i = 0; i < n; i++)
29                  cout << array[i] << "\n";
30
31  }
```

A doua variantă folosește STL. Nu vă faceți griji dacă nu înțelegeți încă cum funcționează, mai multe detalii se vor regăsi în secțiunile care urmează:

```
1  #include <algorithm>
2  #include <vector>
3  #include <iostream>
4  #include <iterator>
5
6  using namespace std;
7
8  int main ()
9  {
10     vector<int> v;
11     istream_iterator<int> start (cin);
12     istream_iterator<int> end;
13     back_insert_iterator< vector<int> > dest(v);
14
15     copy (start, end, dest);
16     sort (v.begin(), v.end());
17     copy (v.begin(), v.end(), ostream_iterator<int>(cout, "\n"));
18     return 0;
19 }
```

6.2 Template-uri

Înainte de a continua, aș dori să precizez că tot codul din STL se compune din clase template. Din acest motiv, pare firesc să ne punem întrebarea "**Ce este un template?**", chiar dacă o parte dintre programatorii începători le folosesc corect fără să conștientizeze cu adevărat ce sunt și cum funcționează.

Un **template** este în esență soluția pe care o aduce C++ pentru programarea generică: este o porțiune de cod în care anumite tipuri de date sunt lăsate "necompletate" în mod intenționat. Tipurile de date abstractizate se notează cu niște nume simbolice (asemănătoare cu niște variabile) care vor fi înlocuite la compilare conform cu specializările și instanțierile din restul codului.

Pentru a da un exemplu scurt, să ne imaginăm că ne-am dori o clasă care să rețină o pereche de obiecte. Fără template-uri, am fi avut doar două soluții, destul de incomode amândoua:

- Să punem în clasă doi pointeri generici (de tip (**void***)) și să lăsăm tratarea tipurilor în întregime în seama programatorului, dar astfel nu am mai fi putut face verificarea tipurilor la compilare
- Să scriem câte o clasă separată pentru fiecare combinație de tipuri de date. Această soluție nu merge în mod evident, deoarece mereu se scriu clase noi, și ar fi un de neconceput să întreținem câteva mii de versiuni diferite de clasă care diferă doar prin tipurile de dată ale unor membrii.

Soluția pe care o oferă C++ prin template-uri este să scriem codul din metodele clasei folosind niște tipuri parametrizate, urmând ca acestea să fie înlocuite după nevoie de compilator. Un template care implementează clasa din aceste exemplu poate fi următorul (chiar dacă există deja clasa std::pair<T,V>)

```
1  #include <iostream>
2
3  // Consideram doua tipuri distincte, la care le spunem generic T si V
4  template<class T, class V>
5  class Pair{
6  public:
7      T tData;
8      V vData;
9      Pair<T,V>(T tData, V vData);
10 };
11
12 // Pentru a exemplifica sintaxa, am ales sa nu definesc
13 // constructorul inline
14 template<class T, class V>
15 Pair<T,V>::Pair(T tData, V vData) : tData(tData), vData(vData)
16 {
17 }
18
19 template<class T, class V>
20 std::ostream& operator<< (std::ostream& out, Pair<T,V>& pair){
21     return out << "(" << pair.tData << "," << pair.vData << ")\n";
22 }
23
24 int main()
25 {
26     // Instantiem un obiect de tip Pair<int,int>
27     Pair<int,int> pair1(1,3);
28     std::cout << pair1;
29
30     // Instantiem un obiect de tip Pair<std::string,double>
31     Pair<std::string,double> pair2(std::string("Eric Cartman"), 10.0f);
32     std::cout << pair2;
33
34     return 0;
```

```
35 }
```

Primul lucru pe care trebuie să îl precizăm aici este că atât **Pair<int,int>** cât și **Pair<std::string,double>** sunt clase distincte! Ele sunt generate de către compilator folosind substituție de tipuri dacă se depistează în cod o nouă combinație de tipuri utilizată. După substituția tipurilor, codul template-ului se compilează separat pentru fiecare specializare în parte (o specializare este o particularizare a template-ului pentru anumite tipuri de date concrete) ca și cum am fi scris clase distincte. Acest lucru încetinește mult compilarea și poate genera erori la compilare neașteptate dacă tipurile de date nu implementează toate metodele invocate în template (în C++ nu există interfețe în mod explcit, este **responsabilitatea programatorului** să se asigure că o anumită clasă implementează toate metodele pe care trebuie să le pună la dispoziție). De asemenea, copierea repetată a codului de numește **code bloating** și în proiectele mari încetinește compilarea foarte mult și duce la binare foarte voluminoase.

Al doilea lucru care trebuie remarcat este acela că template-ul de **operator<<** presupune că tipurile **T** și **V** au, la rândul lor, definit operatorul << pentru ostream-uri. în caz constrar, nu veți putea compila codul. Nu am fi putut scrie ceva similar în C, deoarece **printf** trebuie să țină cont de tipurile de date și formatele pe care le afișează.

O facilitate folosită destul de mult în STL este aceea de supraîncărcare a specializărilor. Asta înseamnă că pentru anumite cazuri particulare de tipuri de date, se pot specifica implementări mai eficiente. Cel mai clar exemplu este template-ul **std::vector<bool>**, care în loc să fie administrat sub forma unui array ca toate celelalte specializari de std::vector, este comprimat pe biți.

Comparativ, în Java nu există efectiv template-uri, chiar dacă există ceva similar: Generics. Mecanismul de clase generice permite specificarea de tipuri de date concrete în cod, de exmeplu ne așteptăm ca un obiect care este o instanță de **java.util.Vector<String>** să conțină doar obiecte de tip String. În realitate, însă, Java face aceste verificări dinamic la runtime, din cauza polimorfismului implicit. Din această cauză, codul nu se multiplică în realitate, ci doar apar niște restricții care să vă scape de cast-uri inutile. Puteți trișa acest mecanism foarte ușor, dacă vă puneți mintea.

O observație undeva la limita dintre comic și serios se referă la cei care declară în Java obiecte de tipul **java.util.Vector<Object>**. Chiar dacă nu este nimic greșit la această declarație, ea nici nu face nimic special, pentru ca în Java **orice** obiect este implicit derivat din tipul Object :).

6.3 Componentele STL

În principiu, STL pune la dispoziție trei tipuri de implementări pe care să le puteți folosi de-a gata în cod:

- Containere
- Adaptori
- Algoritmi

În cele ce urmează vom detalia cu exemple practice fiecare categorie.

6.4 Containere: vector, list, slist, deque, set, map

Clasa template **std::vector<T>** nu este cu nimic mai mult decât un wrapper peste un array alocat dinamic, și funcționează similar cu echivalentul din Java pe care l-am dat exemplu în mod repetat. Cu toate acestea, în C++ există o serie de diferențe specifice limbajului:

- Inserția la sfârșit se face cu ajutorul metodei **push_back()**. Încercarea de a insera elemente pe poziții nealocate poate duce la SegFault, pentru că nu vi se garantează nicăieri capacitatea array-ului din interior (folosiți **resize()** pentru asta).
- Accesul la elemente se poate face fie prin operatorul [], fără verificarea limitelor, fie prin funcția membru **at()**, cu verificarea limitelor. Avantajul funcției **at()** este că nu primiți SegFault dacă accesați indecși invalizi (atunci când mașina virtuală Java ar fi aruncat dulcele mesaj de excepție "Array Index Out of Bounds"). Dezavantajul este că timpul de acces este mai mare (trebuie executat un "if" suplimentar). Pentru că politica C++ este să vă lase să vă alegeți singuri ce vreți să folosiți, operatorul [] vă dă acces direct și rapid la array-ul din interiorul clasei, dar pe propria voastră răspundere!
- Puteți manipula memoria alocata unui vector folosind una din metodele **resize()** sau **reserve()**. Din păcate, există multă confuzie cu privire la folosirea lor, care se traduce adesea în SegFault și vandalizarea tastaturii.
  - Metoda **resize()** redimensionează capacitatea array-ului la capacitatea dată ca parametru. Dacă noua dimensiune este mai mare decât vechea dimensiune, se invocă constructorul tipului pentru noile obiecte de la sfârșit, astfel încât acestea pot fi folosite direct!
  - Metoda **reserve()** face cu totul altceva. Ea nu redimensionează neapărat array-ul alocat, ci mai degrabă se asigură că acesta are o capacitate cel puțin egală cu parametrul. Dacă este nevoie de o realocare pentru a spori capacitatea vectorului, **NU** se invocă constructorul în spațiul de memorie de la sfârșit! Dacă veți încerca să folosiți acele obiecte, veți avea tot felul de surprize neplăcute.

- Atribuirea dintre doi vectori se trauce prin dezalocarea memoriei primului vector și clonarea tuturor obiectelor din vectorul atribuit în vectorul la care se atribuite. Chiar dacă STL face aceste operații în batch, interschimbarea a doi vectori prin trei atribuiri durează mai mult sau mai puțin o eternitate în termeni de timp de execuție. Soluția inteligentă este să folosim metoda **swap()**, care interschimbă în mod inteligent doar pointerii către zonele de memorie din interior și variabilele de statistică.

- Există specializări de vector în STL care sunt reimplementate mai eficient. Mai exact, este vorba de specializarea **std::vector<bool>** care ține elementele pe biți, nu pe char-uri. Teoretic, acest lucru duce la o economie de memorie de 1/8, dar practic alocarea se face în pași incrementali de **sizeof(int)** din motive de performanță de viteză. Dacă acest lucru vi se pare deranjant, aveți libertatea să instanțiați vectorii cu alocatori scriși de voi (vă asigur eu, NU se merită efortul).

Acestea fiind zise, lipsește doar un exemplu concret de folosire. Nu am precizat în mod deosebit numele altor metode din vector, deoarece le puteți găsi în documentați oficială foarte rapid. Am ales în schimb să pun accentul pe înțelegerea funcționării.

```cpp
#include <iostream>
#include <string>

// Headerul care contine template-ul pentru vector
#include <vector>

int main()
{
    int arrayInt[4] = { 0, 1, 2, 3};
    // Cream un vector din intregii intre doua adrese dintr-un array
    std::vector<int> vInt(arrayInt, arrayInt+4);

    // Cream un vector de stringuri
    std::vector<std::string> v;

    char cifre[2] = { 0, 0 };
    for (int i = 0; i < 10; i++){
        cifre[0] = i + '0';
        // Inseram la sfarsit instante de stringuri create din "cifre"
        v.push_back(std::string(cifre));
    }

    // Afisam "0", in doua moduri: cu si fara bounds checking
    std::cout << v[0] << " " << v.at(0) << "\n";

    // Redimensionam vectorul v la 3 elemente
    // (celelalte 7 se distrug automat)
    v.resize(3);

    // Il reextindem la 8 elemente cu reserve()
    // (cele 5 elemente in plus nu se instantiaza automat)
    // Reserve se foloseste pentru a evita in avans realocari in cascada
    v.reserve(8);

    // Cream un vector nou, gol, si il interschimbam cu cel vechi
    std::vector<std::string> w;
    w.swap(v);

    // O sa cauzeze exceptie pentru ca elementul de pe pozitia 6 nu este instantiat
    std::cout << w.at(6) << "\n";

    return 0;
}
```

Clasele template **std::list<T>** și **std::slist<T>** implementează liste dublue, respectiv simplu înlănțuite de obiecte generice. Metodele de inserție sunt **push_back()** și **push_front()** în funcție de capătul la care dorim să inserăm, iar cele de eliminare de elemente sunt **pop_back()** și **pop_front()**.

Aici trebuie de remarcat de asemenea că nu mai este permis accesul prin subscripting, ca la vector (complexitatea ar fi **O(N)**). În schimb, accesul se face mai degrabă la elementele de la captele listelor prin metodele **front()** și **back()**.

În rest, modul de folosire este foarte intuitiv. Un exemplu similar cu cel de la vectori regăsim mai jos:

```
1  #include <iostream>
2  #include <string>
3
4  // Headerul care contine template-urile pentru liste
5  #include <list>
6
7  int main()
8  {
9      std::list<std::string> l;
10
11     char cifre[2] = { 0, 0 };
12     for (int i = 0; i < 10; i++){
13         cifre[0] = i + '0';
14         l.push_back(std::string(cifre));
15     }
16
17     // Afisam elementele de la inceputul si sfarsitul listei
18     std::cout << l.front() << " ... " << l.back() << "\n";
19
20     // Sortam lista
21     l.sort();
22
23     // Golim lista de la coada la cap si afisam elementele
24     // din ea pe masura ce le scoatem
25     while (!l.empty()){
26         std::cout << l.back() << " ";
27         l.pop_back();
28     }
29     std::cout << "\n";
30
31     return 0;
32 }
```

Un **std::deque<T>** funcționează ca o listă dublu înlănțuită, dar permite niște operații suplimentare datorită modului de reprezentare internă. Cea mai importantă operație suplimentară este accesul la elemente prin subscripting (adică prin operatorul []). Din acest punct de vedere seamănă mai mult cu un vector. Această structură de date este alocatorul implicit pentru o serie de adaptori de containere, dar în rest nu are facilități deosebite așa că nu voi insista asupra ei.

Un **std::set<T>** este o implementare a unei multimi în sens matematic. Tipul de date cu care se specializează un template trebuie să suporte comparație (operatorul < să fie bine definit între două obiecte de tip **T**). Inserarea într-un set se realizează cu ajutorul metodei **insert()**, iar căutarea se face cu ajutorul metodei **find()** care întoarce un iterator (mai multe despre iteratori în secțiunea corespunzătoare). Ambele operații se fac în timp amortizat **O(logN)**. Trebuie de asemenea de reținut că seturile sunt implementate cu ajutorul arborilor bicolori (numiți și red-black).

```
1 #include <iostream>
2
3 // Headerul care contine template-ul pentru set
4 #include <set>
5
6 int main()
7 {
8     // Cream un set de intregi. Comparatorul este implicit "<"
9     std::set<int> s;
10
11     // Inseram un element
12     s.insert(1);
13
14     // Verificam existenta lui. find() intoarce un iterator
15     if (s.find(1) != s.end()){
16         std::cout << "Da!\n";
17     } else {
18         std::cout << "Nu!\n";
19     }
20
21     return 0;
22 }
```

Un **std::map<T,V>** este un template cu doi parametrii de instantiere: tipul cheilor și tipul valorilor asociate. Funcționarea este bine cunoscută din Java, dar sunt câteva detalii de implementare unice pentru C++. Ca și **std::vector**, un **map** implementează operatorul de indexare [], dar atenție: folosirea opeartorului conduce automat la inserarea în map a cheii! Dacă nu se atribuie o valoare în cadrul instrucțiunii, cheii îi va corespunde ca valoarea un obiect construit cu constructorul fără parametri. Funcția alternativă la inserare se numește chiar **insert()**, iar funcția de ștergere se numește **erase()**, ca la seturi. De fapt, un map este în realitate doar un set de obiecte de tipul **std::pair<T,V>** sortate după elementul **T**. Cheile sunt, din nou, ținute sub formă de arbore bicolor.

```
1  #include <iostream>
2  #include <string>
3
4  // In acest header se afla template-ul pentru map
5  #include<map>
6
7  int main()
8  {
9      std::map<std::string, int> numere;
10
11     // Inseram implicit un numar prin indexare
12     numere[std::string("unu")] = 1;
13
14     // Inseram prin apel de functie
15     numere.insert(std::pair<std::string, int>(std::string("doi"), 2));
16
17     // Stergem un numar din map
18     numere.erase(std::string("unu"));
19
20     // Afisam un numar din map
21     std::cout << numere[std::string("doi")] << "\n";
22
23     return 0;
24 }
```

Ce este interesant și util la un map este că funcționează aproape ca un vector cu indici de orice tip posibil de dată, însă cu niște penalizări de complexitate.

6.5 Adaptori: queue, stack, priority_queue

Adaptorii sunt niște clase care au rolul de a încapsula în interior alte containere mai generalizate, cu scopul de a restricționa accesul la ele. Voi lua drept exemplu adaptorul pentru stivă deoarece este una dintre cele mai folosite structuri de date. În principiu, în STL nu există o clasă specială care să implementeze doar stive. Cei care au proiectat limbajul și-au dat seama că acest efort ar fi inutil, dar totuși cum ar fi putut prevenit programatorii să apeleze metode de acces ale elementelor de la baza stivei?

Soluția oferită de adaptori este foarte simplă: creăm o clasă nouă care să conțină în interior un membru privat de alt tip (implicit este vorba de un **std::deque**, dar putem da și alți alocatori), iar apelurile la obiectul exterior să fie direcționate către apeluri de la obiectul intern astfel încât să rămână vizibilă doar funcționalitatea care ne interesează (de exemplu, la o stivă ne interesează doar **push()**, **pop()** și **top()**. Principalii adaptori din C++ sunt:

- queue - reprezintă o structură de tip FIFO (first in, first out)
- stack - reprezintă o structură de tip LIFO (last in, first out)
- priority_queue - reprezintă o structură de tip HEAP (cel mai mic element este primul care este scos)

6.6 Iteratori

Având în vedere multitudinea de containere și adaptori pe care C++ o pune la dispoziție în biblioteca standard, a fost nevoie de un alt concept pentru a abstractiza traversarea colecțiilor: **iteratorii**. Un **iterator** în C++ se deosebește destul de mult conceptual față de un iterator din Java. În esență, în C++ concepția de la care s-a pornit a fost că un **iterator** este orice clasă care se comportă ca un pointer:

- Suportă operatorul "++" (eventual și "–") pentru a trece la următorul (eventual precedentul) element din colecție, respectiv adunarea cu întregi
- Atunci când îl referențiem (supraîncărcăm operatorul "*"), returnează un obiect de tipul celor din clasa pe care o deservește.

Conform cu această definiție, un pointer la int este un iterator perfect legal pentru un array (se pot aplica pe el operatorii "++", "+" și "*"). Cu toate astea însă, din cauză că toate colecțiile au de-a face cu iteratori, standardul de facto este o structură de date să aibă în interior un tip de dată care să implementeze un iterator.

Deoarece tipul iterator este definit intern în colecții, acestea pun de obicei la dispoziție două metode prin care se pot obține iteratori:

- **begin()** - această metodă ne oferă primul element din colecție
- **end()** - această metodă ne oferă elementul din colecție imediat după ultimul element. De remarcat că **end()** nu întoarce un iterator la ultimul element din colecție, ci primul element din afara colecției, cu alte cuvinte nu poate fi niciodata referențiat, ci folosește la comparații ca să știm unde să ne oprim.
- **rbegin(), rend()** - aceste metode sunt oglinditele celor de mai sus, doar că dinspre coadă spre cap. Anumite structuri de date pun la dispoziție traversări în ambele direcții, doar că în cazul acesta, rend() o sa fie prima locație dinainte primul element din colecție și din nou, nu poate fi referențiat (pentru că oricum nu există).

Justificarea foarte clară pentru folosirea operatorilor este aceea că **poți parcurge orice colecție dacă știi**

- **de unde să începi**
- **unde să te oprești**
- **cum să treci de la un element la altul**

De asemenea, mai există și iteratori de tip const_iterator, care garantează că nu modifică elementele structurii pe măsură ce se deplasează prin ea.

```
1  #include <iostream>
2  #include <vector>
3  #include <set>
4  #include <map>
5
6  int main()
7  {
8      // Cream un vector, un map si o multime de intregi
9      std::vector<int> v;
10     std::set<int> s;
11     std::map<int,int> m;
12
13     // Inseram in amandoua numerele de la 0 la 9
14     for (int i = 0; i < 10; i++){
15         v.push_back(i);
16         s.insert(i);
17         m[i] = -i; // Cheia e numarul, valoarea e -numarul
18     }
19
20     // Iteram tot setul, in ordine, si afisam
21     for (std::set<int>::iterator it = s.begin(); it != s.end(); it++){
22         // it este iterator de set<int>, deci functioneaza identic cu un pointer
```

```
23          // de int (chiar daca nu este un pointer de int, are aceeasi operatori)
24          std::cout << (*it) << " ";
25      }
26      std::cout << "\n";
27
28      // Stergem din vector primul si al doilea element. Obsevatie:
29      // v.begin()+2 este, ca si v.end(), primul element care NU participa la stergere
30      v.erase(v.begin(), v.begin()+2);
31
32      // Afisam ce a ramas din vector in ordine inversa
33      std::vector<int>::reverse_iterator rit;
34      for (rit = v.rbegin(); rit != v.rend(); rit++){
35          std::cout << (*rit) << " ";
36      }
37      std::cout << "\n";
38
39      // Iteram prin map si afisam. Observatie: un map este un set de pair<Key,Value>
40      // deci un iterator de map functioneaza ca un pointer la un pair<Key,Value>
41      for (std::map<int,int>::iterator it = m.begin(); it != m.end(); it++){
42          std::cout << "(" << it->first << "," << it->second << ")\n";
43      }
44
45      return 0;
46  }
47
```

6.7 Algoritmi

Probabil cel mai mare avantaj din folosirea STL îl reprezintă existența algoritmilor gata implementați pentru conainerele definite. Lista acestor algoritmi este foarte variată. Astfel, STL pune la dispoziție algoritmi pentru:

- **Căutare și statistici** : find, count, mismatch, equal, search
- **Modificarea ordinii elementelor** : swap, copy, reverse, replace, rotate, partition
- **Sortare** : sort, partial_sort, nth_element
- **Căutare binară** : binary_search, lower_bound, upper_bound
- **Interclasare** : merge, set_intersection, set_difference
- **Gestiunea datelor** : make_heap, sort_heap, push_heap, pop_heap
- **Minim și maxim** : min, min_element, lexicographical_compare

Din motive de păstrare a generalității, algoritmii din STL se folosesc de iteratori. Deși inițial folosirea acestor implementări ar putea părea greoaie, odată stăpânite conceptele care stau la baza lor, veți reuși să realizați implementări foarte rapide.

În continuare, vom da două exemple de folosire ale sortării, respectiv căutării binare din STL

```
1 #include <algorithm>
2 #include <vector>
3 #include <iostream>
4
5 int main()
6 {
7     // Cream un vector cu niste intregi oarecare
8     int myInts[] = { 32, 71, 12, 45, 26, 80, 53, 33 };
9     std::vector<int> v(myInts, myInts+8);
10
11    // Sortam intre ele elementele de pe pozitiile 0, 1, 2, 3 si 4
12    // Vom specifica capetele intre care sortam prin doi iteratori
13    std::sort(v.begin(), v.begin()+5);
14
15    // Ca sa gasim rapid un element in vector, vom folosi cautarea binara
16    // Intai si intai, terbuie sa sortam tot vectorul pentru a cauta binar
17    std::sort(v.begin(), v.end());
18
19    // Verificam daca elementul "15" se afla in vector
20    if (binary_search(v.begin(), v.end(), 15)){
21        std::cout << "15 exista in vector!\n";
```

```
22      } else {
23          std::cout << "15 nu exista in vector!\n";
24      }
25
26      return 0;
27  }
```

6.8 Greșeli frecvente

Programatorii de Java tind să facă la început o serie de greșeli în folosirea containerelor din STL, datorate asemănării înșelătoare dintre sintaxa celor două limbaje.

Derivarea containerelor. În Java, o practică uzuală este derivarea colecțiilor de date pentru a le modifica funcționalitatea. Această practică este foarte greșită în C++. Limbajul nu vă interzice să faceți asta, dar derivarea claselor rezultate din specializarea template-urilor de STL vă poate cauza memory leak-uri și comportamente neașteptate, deoarece destructorii containerelor din C++ NU sunt **virtuali**. Ei au fost lăsați intenționat nevirtuali în limbaj din motive de performanță. Chiar dacă sunt coincidențe în care teoretic ar funcționa corect și derivarea, singura practică corectă de a extinde un container de C++ este scrierea unui **wrapper** (poartă și numele de adaptor, dar nu trebuie confundați cu adaptorii din Java). Un **wrapper** este o clasă care conține în interior un obiect pe care îl manipulează într-un anumit mod.

Inserarea prin copiere. În Java, atunci când inseram un obiect într-o colecție, de fapt ceea ce inseram noi era o referință la un obiect, și nu obiectul în sine! În C++, ambele comportamente sunt posibile. Putem avea:

- containere de instanțe de obiecte (trebuie să fie atribuibile). Ex: **std::vector<MyClass>**
- containere de pointeri la obiecte (se comportă cam ca în Java). Ex: **std::vector<MyClass*>**

ATENȚIE!. Atunci când avem containere de obiecte, inserția se face prin copiere! Cu alte cuvinte, dacă scriem **v.push_back(someObject);**, în realitate se va instanția un obiect nou care se va depune în container, iar someObject va rămâne neatins. Obiectele din containere trebuie să poată fi asignabile (dacă au constante în interior, va trebui să redefiniți operatorul de atribuire)!

Iteratorii inconsisteți și SegFault-urile. Ar trebui ca de fiecare dată când folosiți iteratori, să considerați că aceștia sunt un tip de pointei, și deci vă pot cauza SegFault. În principiu, un iterator rămâne valid, sau "safe", atâta timp cât colecția din care a fost creat nu se modifică. Ca să înțelegem mai bine această problemă, să presupunem următorul scenariu: Avem un vector<int> în care simulăm o coadă (de exemplu, la parcurgerea în lățime a unui graf). În cursul simulării, parcurgem vectorul cu un iterator și pentru fiecare element vizitat adăugăm, eventual, alte elemente la finalul vectorului. Aparent, am putea spune că nimic rău nu se poate întâmpla, dar trebuie să ne gândim la ce se întâmplă în interior. De fiecare dată când adăugăm la finalul vectorului un nou element, există posibilitatea să fie nevoie de o realocare în array-ul din interior. Dacă realocarea nu se poate face in-place din diverse motive, atunci se va aloca memorie în altă parte și se va muta conținutul în noua locație. Vechea locație de memorie devine invalidă (cu alte cuvinte, a "fugit" memoria de sub iterator). La următoarea referențiere de operator, vom obține SegFault. Un alt caz

des întâlnit de SegFault este încercarea de a șterge elemente din containere pe măsură ce acestea sunt parcurse de iteratori. În C++, think pointer-style!

Containerele sortate. O greșeală destul de frecventă este și presupunerea că un container sortat (de exemplu, un set sau un map) se reorganizează automat atunci când modificăm conținutul unui obiect din interior prin intermediul unui pointer. Acest lucru **nu este adevărat**! Dacă veți modifca prin pointeri (sau iteratori) obiectele din interior, veți face containerul inconsistent. O abordare corectă de a păstra conținutul sortat este să ștergeți din container obiectul pe care vreți să îl modificați și să reintroduceți versiunea modificată cu **insert()**.

Sincronizarea. Chiar dacă nu veți scrie în anul 2 programe paralele, puteți reține despre containerele din C++ că **NU** sunt thread-safe! Dacă veți vrea să le partajați între thread-uri, trebuie să le protejați explicit cu mutex-uri sau semafoare.

7 Namespace-uri

Un aspect al cărei lipsă se simțea în C și pe care C++ l-a introdus este posibilitatea separării spațiilor de nume în limbaj.

Un **spațiu de nume** poate fi comparat cu un context în care declarăm nume (de clase, de tipuri, de variabile, de funcții, etc.). În interiorul unui spațiu de nume, nu putem avea două nume identice (de exemplu, nu putem avea două variabile sau două clase cu același nume), dar în spații de nume diferite nu avem acest tip de restricții.

Cel mai bun exemplu de spațiu de nume îl reprezintă o clasă (deși o clasă nu este efectiv un spațiu de nume, doar se comportă asemănător). Astfel, în interiorul clasei std::vector<int>, numele "iterator" specifică în mod unic un singur tip de dată, în timp ce în interiorul clasei std::map<int,int>, numele "iterator" specifică tot în mod unic, un alt tip de dată. Chiar dacă cele două tipuri de dată se numesc la fel, ele fac parte din spații de nume diferite și nu se confundă între ele.

Spațiile de nume au structură ierarhică (de arbore) și sunt specificate sub formă de prefixe, urmate de operatorul de rezoluție de scop (::). Ați observat că în cadrul exemplelor din acest tutorial am scris aproape de fiecare dată **std::** în fața claselor și constantelor din STL. **std::** este spațiul de nume standard în care sunt depuse toate numele care fac parte din standardul limbajului. Un alt spațiu de nume este spațiul global (sau implicit) de nume. Din acest spațiu fac parte clasele definite de utilizator sau variabilele globale care nu sunt incluse în nici un alt spațiu de nume.

Un spațiu de nume se poate specifica înbrăcând codul într-un bloc de forma **namespace spațiul_meu_de_nume { ...cod... }** .

Uneori devine obositor, sau este inestetic să prefixăm toate clasele folosite cu namespace-ul din care acestea fac parte (mai ales dacă este vorba de un namespace lung). De exmeplu, dacă avem o funcție statică numită "makeCoffee" în clasa "BestClassEver" din namespace-ul "best_project_ever" care face parte din namespace-ul "best_company_ever", apelarea ei ar trebui să fie de forma best_company_ever::best_project_ever::BestClassEver::makeCoffee(); Cu siguranță nu ne dorim să scriem așa ceva prea des.

Namespace-ul implicit în cod este cel curent (prefixarea cu el este redundantă). El poate fi prescurtat și prin **::** la nevoie, pentru a distinge între variabile globale și locale. Pentru accesul la numele din alte spații, trebuie să precizăm explicit acest lucru. Totuși, putem evita scrierea repetitivă a altor spații de nume dacă instruim compilatorul să facă "merge" între două spații de nume. Acest lucru se poate realiza cu ajutorul directivei **using** ca în exemplul de mai jos:

```cpp
#include <vector>

// Din acest moment, facem reuniunea spatiilor :: si std::
// Pentru toate clasele folosite, compilatorul va incerca sa caute singur
// spatiul de nume corect (:: sau std::, dupa caz). In cazul in care exista
// mai mult de o posibilitate, va trebuie sa scrieti namespace-ul explicit!
using namespace std;

// Se poate specifica si reuniunea cu o singura clasa astfel:
// using spatiu_de_nume::Clasa;

int main()
{
    int numere[5] = { 4, 3, 2, 1, 0 };
    vector<int> v(numere, numere+5);

    sort(v.begin(), v.end());

    for (vector<int>::iterator it = v.begin(); it != v.end(); ++it){
        cout << (*it) << " ";
    }

    cout << "\n";

    return 0;
}

```

Chiar dacă folosirea lui **using** scurtează codul, este greșit să folosim **using** în headere! Headerele sunt fișiere care pot fi incluse unele în altele și în fișiere sursă în orice combinații. Existența unui **using** în ele poate duce accidental la coliziuni de nume din două namespace-uri diferite, coliziuni care nu ar fi fost posibile dacă nu foloseam **using**.

În definitiv, folosirea lui **using** pentru spațiul **std::** în fișierele de implementare ține de gusturile fiecăruia. Chiar dacă produce cod mai scurt și mai clar, anumiți programatori preferă să scrie mereu explicit spațiile de nume.

8 Tratarea Excepțiilor

C++ a introdus la apariția sa mecanismul de tratare al excepțiilor (**try-catch**). Această funcționalitate a fost transpusă în Java, unde a devenit foarte importantă (știm deja că Java vă obligă să tratați toate excepțiile). În C++, însă, lucrurile stau mult diferit.

Limbajul nu obligă tratarea excepțiilor. Dacă o funcție aruncă o excepție care nu poate fi tratată de nici o funcție de deasupra ei din lanțul de apeluri, atunci se încheie forțat execuția programului.

Mai mult, excepțiile în C++, spre deosebire de Java, pot fi obiecte de orice tip de dată! Acest lucru ar fi făcut îngrozitoare tratarea tuturor tipurilor de excepție de fiecare dată, motiv pentru care s-a introdus operatorul **ellipsis**, notat prin trei puncte **(...)**. Semnificația acestui operator este orice excepție, și funcționează similar cu superclasa **Exception** din Java.

În continuare se exemplifică sintaxa de excepții din C++:

```
#include <iostream>

int main()
{
    try{
        // Asa se arunca o exceptie de tip int
        throw 5;
    } catch (int errCode) {
        // Care este prinsa aici
        std::cout << "Exceptie de tip int.\n";
    } catch (...) {
        // Just to make sure, lasam si posibilitatea altor exceptii
        std::cout << "Orice alta exceptie.\n";
    }

    return 0;
}
```

Observație: Majoritatea coding guide-urilor renumite condamnă folosirea excepțiilor în cod deoarece duc la cod greu de întreținut. Acest lucru contrastează puternic cu mentalitatea din Java!

O altă observație este aceea că stream-urile din STL nu aruncă în mod obligatoriu excepții. Dacă dorim ca ele să arunce excepții, acest lucru trebuie setat manual din cod.

9 Resurse

În încheierea acestui tutorial, sper că ați reușit să vă formați o părere asupra limbajului. În decursul secțiunilor, am încercat să ating toate chestiunile importante, fără a merge mai în detaliu dincolo de strictul necesar atât scrierii de cod corect cât și înțelegerii documentației și surselor din biblioteci.

După cum am spus și în introducere, acest tutorial nu ține nicidecum locul unui manual de C++. Celor care le place limbajul și care sunt interesați să îl înțeleagă în detaliu, le recomand în special cartea "The C++ Programming Language", scrisă de Bjarne Stroustrup, creatorul limbajului. Totuși, cartea este extrem de voluminoasă și foarte detaliată, și s-ar putea să descurajeze programatorii care sunt la primele contacte cu limbajul.

În continuare, am ales câteva surse foarte bune de documentație care vă vor fi de real ajutor de fiecare dată când vă loviți de probleme în C++:

Resurse online:

- Sursa oficiala de documentatie pt C++ (prima și cea mai accesibilă sursă de informare despre limbaj!)
- Un tutorial de bază pentru STL
- Un tutorial ceva mai serios despre STL
- Introducere în STL
- Tutorialul de STL al www.decompile.com
- Site-ul oficial al bibliotecii BOOST

Cărti:

- "The C++ Programming Language - Special Edition", Bjarne Stroustrup
- "C++", Danny Kalev, Michael J. Tobler, Jan Walter