Laborator 2 - Notiuni de C++

Responsabili

- Isabella Mincă [mailto:mailto:isabella.minca@gmail.com]
- Adina Budrigă [mailto:mailto:adina.budriga@gmail.com]

În cadrul acestui laborator ne propunem să continuăm ilustrarea conceptelor din C++ cu care veți lucra pe parcursul acestui semestru.

Obiective

Ne dorim să:

- Să continuăm tranziția de la C la C++
- Învățăm alocarea memoriei in C++
- Învățăm ce reprezintă rule of three
- Învățăm supraincarcarea operatorilor.

Referințe

In C++ există două modalități de a lucra cu adrese de memorie:

- pointeri (la fel ca cei din C)
- referințe.

Referinţa poate fi privită ca un pointer constant inteligent, a cărui iniţializare este forţată de către compilator (la definire) şi care este dereferenţiat automat.

Semantic, referințele reprezintă aliasuri ale unor variabile existente. La crearea unei referinţe, aceasta trebuie iniţializată cu adresa unui obiect (nu cu o valoare constantă).

Sintaxa pentru declararea unei referințe este:

```
tip& referinta = valoare;
```

Exemplu:

```
    int x = 1, y = 2;
    int& rx = x; // Referinta
    rx = 4;      // Modificarea variabilei prin referinta
    rx = 15;     // Modificarea variabilei prin referinta
    rx = y;      // Atribuirea are ca efect copierea continutului
                 // Din y in x si nu modificarea adresei referintei
```

Spre deosebire de pointeri:

- referinţele sunt iniţializate la creare (pointerii se pot iniţializa oricând)
- referinţa este legată de un singur obiect şi această legătură nu poate fi modificată pentru un alt obiect
- referințele nu au operații speciale, toți operatorii aplicați asupra referințelor sunt de fapt aplicați asupra variabilei referite(de exemplu extragerea adresei unei referințe va returna adresa variabilei referite)
- nu există referinţe nule – ele sunt întotdeauna legate de locaţii de memorie

Referinţele se folosesc:

- în listele de parametri ale funcţiilor
- ca valori de întoarcere ale funcţiilor

Motivul pentru aceste tipuri de utilizări este unul destul de simplu: când se transmit parametrii funcțiilor, se copiază conținutul variabilelor transmise pe stivă, lucru destul de costisitor. Prin transmiterea de referințe, nu se mai copiază nimic, așadar intrarea sau ieșirea dintr-o funcție sunt mult mai putin costisitoare.

Smart pointers

Sunt clase wrapper peste pointeri folosite pentru manipularea memoriei alocate dinamic.

Avantajul utilizării smart pointer-ilor față de cei obișnuiți (C-like) este acela că nu trebuie să ne mai facem griji pentru eliberarea memoriei, acest lucru făcându-se automat și cu o politică stabilită de la început, în funcție de tipul pointer-ului ales (poate fi *shared_ptr* - mai mulți pointeri dețin o anumită resursă, iar când ultimul dintre aceștia iese din scope-ul programului, memoria este eliberată automat; *unique_ptr* - ajută la manipularea unei singure resurse și nu pot fi copiați, iar obiectul deținut este șters atunci când fie unique_ptr-ul ce îl referențiază este șters sau resetat).

Smart pointer-ii respectă modelul RAII (Resource Acquisition Is Initialization - achiziţia resurselor prin iniţializare): înglobează un obiect și eliberează resursa folosită de el prin destructorul său, iar în sine, sunt folosiți ca și pointerii clasici, învățați până acum.

Mai multe despre aceștia puteți afla aici [https://www.codeproject.com/Articles/541067/Cplusplus-Smart-Pointers].

Keyword const

În C++, există mai multe întrebuințări ale cuvântului cheie **const**:

- specifică un obiect a cărui valoare nu poate fi modificată
- specifică metodele unui obiect read-only care pot fi apelate

Pentru a specifica, un obiect a cărui valoare nu poate fi modificată, **const** se poate folosi în următoarele feluri:

- `const tip variabila` ⇒ specifică o variabilă constantă
- `tip const& referinta_ct = variabilă;` ⇒ specifică o referință constantă la un obiect, obiectul neputând fi modificat
- `const int *p_int` ⇒ specifică un pointer la int modificabil, dar conținutul locației de memorie către care `p_int` arată nu se poate modifica.
- `int * const p_int` ⇒ specifică un pointer la int care nu poate fi modificat (Variabilei `p_int` nu i se poate asigna nici o valoare, dar conținutul locației de memorie către care `p_int` arată se poate modifica)

Orice obiect constant poate apela doar funcții declarate constante. O funcție constantă se declară folosind sintaxa:

```
void fct_nu_modifica_obiect() const; // Am utilizat cuvântul cheie const
                                     // Dupa declarația funcției fct_nu_modifica_obiect
```

Această declaratie a functiei garantează faptul că obiectul pentru care va fi apelată nu se va modifica.

Regula de bază a apelării membrilor de tip funcție ai claselor este:

- funcțiile `const` pot fi apelate pe toate obiectele

- funcțiile non-const pot fi apelate doar pe obiectele non-const.

Exemple:

complex.h

```
// Declarație
class Complex {
private:
    int re;
    int im;

public:
    Complex();

    int GetRe() const;
    int GetIm() const;

    void SetRe(int re);
    void SetIm(int im);
};
```

main.cpp

```
int main() {
    // Apelare
    Complex c1;
    const Complex c2;
    c1.GetRe();        // Corect
    c1.SetRe(5);       // Corect
    c2.GetRe();        // Corect
    c2.SetRe(5);       // Incorect
}
```

Funcții care returnează referințe

Pentru clasa Complex, definim funcţiile care asigură accesul la partea reală, respectiv imaginară a unui număr complex:

```
double getRe(){ return re; }
double getIm(){ return im; }
```

Dacă am dori modificarea părţii reale a unui număr complex printr-o atribuire de forma:

```
z.getRe()=2.;
```

constatăm că funcţia astfel definită nu poate apărea în partea stângă a unei atribuiri.

Acest neajuns se remediază impunând funcţiei să returneze o referinţă la obiect, adică:

```
double& getRe(){ return re; }
```

Codul de mai sus returnează o referință către membrul `re` al obiectului `Complex z`, așadar orice atribuire efectuată asupra acestui câmp va fi vizibilă și în obiect.

Clase/metode prietene

Așa cum am văzut în primul articol, fiecare membru al clasei poate avea 3 specificatori de acces:

- public
- private
- protected

Alegerea specificatorilor se face în special în funcție de ce funcționalitate vrem să exportăm din clasa respectivă.

Dacă vrem să accesăm datele private/protejate din afara clasei, avem următoarele opțiuni:

- Funcții care ne întorc/setează valorile membre
- Funcții/Clase prietene (friend) cu clasa curentă.

O funcție prieten are următoarele proprietăți:

- O funcţie este considerată prietenă al unei clase, dacă în declararea clasei, este declarată funcţia respectivă precedată de specificatorul **friend**
- Declararea unei funcţii prieten poate fi făcută în orice parte a clasei(publică, privată sau protejată).
- Definiţia funcţiei prieten se face global, în afara clasei.
- Funcția declarată ca **friend** are acces liber la orice membru din interiorul clasei.

O clasă prieten are următoarele proprietăți:

- O clasă B este considerată prieten al unei clase A, dacă în declararea clasei A s-a întâlnit expresia: `friend class B`
- Clasa B poate accesa orice membru din clasa A, fără nici o restricție.

De asemenea, dacă clasa A este considerată prieten cu clasa B, nu înseamnă că si clasa B este considerată prieten cu clasa A. Nici tranzitivitatea nu este valabilă în relaţia de prietenie dintre clase.

Exemplu:

complex.h

```
#include <cmath>
#include "polinom.h"

class Complex {
private:
    int re;
    int im;

public:
    int GetRe();
    int GetIm();

    friend double ComplexModul(Complex c);   // Am declarat fct ComplexModul ca prieten
    friend class Polinom;                    // Acum clasa Polinom care acces deplin la
                                             // membrii **re** și **im**
};

double ComplexModul(Complex c) {
    return sqrt(c.re * c.re + c.im * c.im);  // Are voie, intrucat e prietena
}
```

Supraîncarcarea operatorilor

Un mecanism specific C++ este supraîncarcarea operatorilor, prin care programatorul poate asocia noi semnificaţii operatorilor deja existenţi. De exemplu, dacă dorim ca două numere complexe să fie adunate, în C trebuie să scriem funcții specifice, nenaturale. În C++ putem scrie foarte ușor:

```
Complex a(2, 3);
Complex b(4, 5);
Complex c = a + b; // Operatorul + a fost supraîncarcat pentru a aduna două numere complexe
```

Acest lucru este posibil, întrucât un operator este văzut ca o funcție, cu declarația:

```
    tip_rezultat operator#(listă_argumente);
```

Așadar pentru a supraîncărca un operator pentru o anumită clasă, este necesar să declarăm funcția următoare în corpul acesteia:

```
    tip_rezultat operator#(listă_argumente);
```

Există câteva restricții cu privire la supraîncarcare:

- Nu pot fi supraîncărcaţi operatorii: ::, ., .*, ?:, sizeof.
- Setul de operatori ai limbajul C++ nu poate fi extins prin asocierea de semnificaţii noi unor caractere, care nu sunt operatori, de exemplu nu putem defini operatorul === .
- Prin supraîncărcarea unui operator nu i se poate modifica aritatea (astfel operatorul ! este unar şi poate fi redefinit numai ca operator unar).
- Asociativitatea şi precedenţa operatorului se menţin.
- La supraîncărcarea unui operator nu se pot specifica argumente cu valori implicite.

Operatori supraîncărcaţi ca funcţii membre

Funcţiilor membru li se transmite un argument implicit **this** (adresa obiectului curent), motiv pentru care un operator binar poate fi implementat printr-o funcţie membru nestatică cu un singur argument.

Operatorii sunt interpretați în modul următor:

- Operatorul binar **a#b** este interpretat ca **a.operator#(b)**
- Operatorul unar prefixat **#a** este interpretat ca **a.operator#()**
- Operatorul unar postfixat **a#** este interpretat ca **a.operator#(int)**

complex.h

```
#include <iostream>

class Complex {
public:
    double re;
    double im;

    Complex(double real, double imag): re(real), im(imag) {};

    // operatori supraîncărcați ca funcții membre
    Complex operator+(const Complex& d);
    Complex operator-(const Complex& d);
    Complex& operator+=(const Complex& d);
};
```

complex.cpp

```
#include "complex.h"

Complex Complex::operator+(const Complex& d) {
    return Complex(re + d.re, im + d.im);
}

Complex Complex::operator-(const Complex& d) {
    return Complex(re - d.re, im - d.im);
}

Complex& Complex::operator+=(const Complex& d) {
    re += d.re;
    im += d.im;
    return *this;
}
```

Supraîncărcarea operatorilor I/O

În C++, orice dispozitiv de I/O este văzut drept un stream, așadar operațiile de I/O sunt operații cu stream-uri, care se definesc în felul următor:

- **Citire**: se execută cu operatorul de extracție », membru al clasei istream
- **Scriere**: se execută cu operatorul de inserție «, membru al clasei ostream

Acești operatori pot fi supraîncărcați pentru o clasă pentru a defini operații de I/O direct pe obiectele clasei. În general, pentru clase care au foarte mulți membri, afișarea individuală a acestora poate deveni ușor obositoare și deloc estetică. Spre exemplu:

student.h

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

class Student {
private:
    int grade, age, friends;
    std::string name, school;

public:
    Student(int newGrade=0, int newAge=0, int newFriends=0,
            std::string newName="", std::string newSchool=""):
            grade(newGrade), age(newAge), friends(newFriends),
            name(newName), school(newSchool) {}

    int getGrade() { return this->grade; }
    int getAge() { return this->age; }
    int getFriends() { return this->friends; }

    std::string getName() { return this->name; }
    std::string getSchool() { return this->school; }
};

int main() {
    Student student(10, 18, 100, "MrPerfect", "CTI");

    std::cout << "Studentul " << student.getName() << " are " <<
                                 student.getAge() << " ani, " <<
                                 student.getFriends() << " de prieteni si invata la " <<
                                 student.getSchool() << "!\n";

    return 0;
}
```

Să ne imaginăm că am avea de printat informația pentru mai mulți studenți - în cel mai bun caz, cu ceea ce știm până acum, am putea să creăm o nouă metodă (să îi spunem printInfo) în cadrul clasei Student, folosită pentru a afișa informația cerută.

```
class Student {
…
void printInfo() {
    std::cout << "Studentul " << student.getName() << " are " <<
                                 student.getAge() << " ani, " <<
                                 student.getFriends() << " de prieteni si invata la " <<
                                 student.getSchool() << "!\n";

}
...
};
```

Chiar și așa, nu am putea apela metoda respectivă în mijlocul unui output statement, al unui stream, ci astfel:

```
std::cout << “Afisez informatia pentru: \n”;
student.printInfo();
std::cout << “Done\n”;
```

Soluția recomandată este aceea a supraîncărcării operatorului binar «, având ca operand stâng std::cout, iar cel drept un obiect de tipul Student, în cazul de față.

student.h

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

class Student {
private:
    int grade, age, friends;
    std::string name, school;

public:
    Student(int newGrade=0, int newAge=0, int newFriends=0,
            std::string newName="", std::string newSchool=""):
            grade(newGrade), age(newAge), friends(newFriends),
            name(newName), school(newSchool) {}

    int getGrade() { return this->grade; }
    int getAge() { return this->age; }
    int getFriends() { return this->friends; }

    std::string getName() { return this->name; }
    std::string getSchool() { return this->school; }

    friend std::ostream& operator<< (std::ostream &out, const Student &student);
};

std::ostream& operator<< (std::ostream &out, const Student &student) {
    /* Operatorul << este prieten cu clasa Student, deci ii poate
       accesa membrii, chiar daca acestia sunt private. */
    out << "Studentul " << student.name << " are " <<
                           student.age << " ani, " <<
                           student.friends << " de prieteni si invata la " <<
                           student.school << "!\n";

    return out;
}
```

main.cpp

```
#include "student.h"

int main() {
        Student student(10, 18, 100, "MrPerfect", "CTI");

        std::cout << student;

        return 0;
}
```

Similar se procedează pentru încărcarea operatorului », cu observația că std::cin este, de această dată, un obiect de tipul std::istream.

Supraîncărcarea operatorului de atribuire

Așa cum am amintit mai sus, majoritatea operatorilor pot fi supraîncărcați. O atenție importantă trebuie acordată operatorului de atribuire, dacă nu este supraîncărcat, realizează o copiere membru cu membru.

Pentru obiectele care nu conţin date alocate dinamic la iniţializare, atribuirea prin copiere membru cu membru funcţionează corect, motiv pentru care nu se supraîncarcă operatorul de atribuire.

Pentru clasele ce conţin date alocate dinamic, copierea membru cu membru, executată în mod implicit la atribuire conduce la copierea pointerilor la datele alocate dinamic, în loc de a copia datele.

Operatorul de atribuire poate fi redefinit numai ca funcţie membră, el fiind legat de obiectul din stânga operatorului =, motiv pentru care va întoarce o referinţă la obiect.

mystring.h

```
class MyString {
    char* s;
    int n; // Lungimea sirului

public:
    MyString();
    MyString(const char* p);
    MyString(const String& r);

    ~MyString();

    MyString& operator=(const String& d);
    MyString& operator=(const char* p);
};
```

mytring.cpp

```
#include "mystring.h"
#include <string.h>

MyString& MyString::operator=(const MyString& d) {
    if (this != &d) {        // Evitare auto-atribuire
        if(s) {              // Curatare
            delete [] s;
        }

        n = d.n;             // Copiere
        s = new char[n + 1];
        strcpy(s, d.s);
    }

    return *this;            // Intoarce referinta la obiectul modificat
}

MyString& MyString::operator=(const char* p) {
    if (s) {
        delete [] s;
    }

    n = strlen(p);
    s = new char[n+1];
    strcpy(s, p);

    return *this;
}
```

Copy-constructor

Reprezintă un tip de constructor special care se folosește când se dorește/este necesară o copie a unui obiect existent. Dacă nu este declarat, se va genera unul default de către compilator.

Poate avea unul din următoarele prototipuri

- MyClass(const MyClass& obj);
- MyClass(MyClass& obj);

Când se apelează?

1) Apel explicit

explicit_copy_constructor_call.cpp

```
MyClass m;
MyClass x = MyClass(m); /* apel explicit al copy-constructor-ului */
```

2) Transfer prin valoare ca argument într-o funcție

call_by_value.cpp

```
void f(MyClass obj);
...
MyClass o;
f(o); /* se apelează copy-constructor */
```

3) Transfer prin valoare ca return al unei funcții

return_by_value.cpp

```
MyClass f() {
    MyClass a;
    return a; /* se apelează copy-constructor */
}
```

4) La inițializarea unei variabile declarate pe aceeași linie

init.cpp

```
MyClass m;
MyClass x = m; /* se apelează copy-constructor */
```

Tipul parametrului pentru copy-constructor trebuie să fie identic cu cel al parametrului pentru operatorul de assignment.

Rule of Three

Reprezintă un concept de **must do** pentru C++. Astfel:

Dacă programatorul și-a declarat/definit unul dintre **destructor**, **operator de assignment** sau **copy-constructor**, trebuie să îi declare/definească și pe ceilalți 2.

De ce poate crea probleme nerespectarea regulii?

Copy constructorul implicit face atribuire în cazul membrilor cu tip primitiv și apelează recursiv copy constructorul membrilor săi cu tip complex (class, struct, union).

De exemplu pentru clasa:

base.h

```
#include "foo.h"
#include "bar.h"

class Base {
    int x;
    int *v;
    Foo f;
    Bar b;
};
```

Copy constructorul implicit va arăta astfel:

```
Base(const Base& other) : x(other.x), v(other.v), f(other.f), (other.b) {}
```

În mod similar, operatorul de assignment implicit face atribuire memberwise:

```
void operator=(const Base& other) {
    x = other.x;
    v = other.v;
    f = other.f;
    b = other.b;
}
```

Se observă că în cazul unui membru de tip pointer (v in acest caz) se va face shallow copy (v = other.v) atât in copy constructorul implicit, cât și în cadrul operatorului de assignment implicit. De aceea, dacă este nevoie de deep copy (copiere element cu element), aceasta ar trebui sa fie oferită atât de copy constructor, cât și de operatorul de assignment.

De asemenea, în acest caz cel mai probabil va fi nevoie si de eliberarea resurselor, deci destructorul nu va avea comportamentul default.

Pe de altă parte, dacă se definește destructorul, probabil acesta elberează resurse alocate dinamic. Dacă s-a realizat doar shallow copy în cadrul unuia dintre copy constructor și operatorul de assignment, atunci resursa va fi eliberată de 2 ori (un exemplu se poate urmări aici [http://www.geeksforgeeks.org/rule-of-three-in-cpp/]).

Schelet

Schelet

Exercitii

Fiecare laborator va avea unul sau doua exerciții publice si un pool de subiecte ascunse, din care asistentul poate alege cum se formeaza celelalte puncte ale laboratorului.

1) [**3p**] Clasa Complex

- [**0.5p**] Adăugați clasei Complex metode pentru adunare, scădere și înmulțire cu un alt număr complex utilizand supraincarcarea operatorilor.
- [**2p**] Implementati copy-constructorul, assignment operator si destructorul.
- [**0.5p**] Supraîncărcați operatorul ”«” pentru a afișa mai ușor un obiect de tip Complex.

2) [**3p**] Clasa Fraction

- [**2p**] Adăugați clasei Fraction metode pentru înmulțire/împărțire cu o altă fracție, respectiv pentru determinarea numărului zecimal echivalent, utilizand supraincarcarea operatorilor.
- [**1p**] Testați (și verificați corectitudinea) prin intermediul exemplului pus la dispoziție în Fraction.cpp.

3) [**3p**] Clasa MappingEntry - conține 2 membri de tipuri potențial diferite (aici vom folosi `int` și `char*`) și realizează, din punct de vedere conceptual, asocierea între două valori (una se numește *cheie*, iar cealaltă *valoare*).

- [**1p**] Implementați și folosiți clasa MappingEntry de mai sus adăugând constructor, destructor.
- [**1p**] Alocați o instanță de tip MappingEntry local și dinamic (utilizând new / delete).
- [**1p**] Verificați cu Valgrind că nu aveți memory leaks.

4) [**4p**] Fie clasa Person, care incapsuleaza numele(char *) si varsta(int) a unei persoane.

- [**2p**] Implementati copy-constructorul, assignment operator si destructorul pentru clasa Person
- [**1p**] Arătați funcționalitatea prin adăugarea unui cod simplist în fișierul Person.cpp.
- [**1p**] Verificați cu Valgrind că nu aveți memory leaks.

5) [**4p**] Fie clasa Student, ce conține informații despre numele unui elev(char * sau string), clasa în care este(int) și media obținută la matematică (float).

- [**1p**] Implementati copy-constructorul, assignment operator si destructorul pentru clasa Student
- [**1p**] Creați o funcție prietenă care să primească un obiect de tip Student și să afișeze informațiile cunoscute despre acesta (numele, clasa, media la matematică).

- [**2p**] Creați clasa Professor care să fie prietenă cu clasa Student. Clasa Professor va conține un vector de elevi și va avea o metodă prin care un profesor să poată modifica nota la matematică a unui elev dat.

Interviu

Această secțiune nu este punctată și încearcă să vă facă o oarecare idee a tipurilor de întrebări pe care le puteți întâlni la un job interview (internship, part-time, full-time, etc.) din materia prezentată în cadrul laboratorului.

- Care este diferența dintre referințe și pointeri?
- Ce este o clasă prietenă? Ce caracteristici și avantaje au acestea?
- Poate un copy constructor să fie private?

Și multe altele…

Bibliografie obligatorie

1. Cum scriem cod C++ corect? [https://google-styleguide.googlecode.com/svn/trunk/cppguide.xml]
2. Tutorial basic C++ [http://swarm.cs.pub.ro/~adrian.sc/PA/TutorialC++.pdf]

Bibliografie recomandată

1. C++ Reference [http://www.cplusplus.com]
2. Standard C++98 [http://sites.cs.queensu.ca/gradresources/stuff/cpp98.pdf]
3. Standard C++11 [https://github.com/cplusplus/draft/blob/master/papers/N3485.pdf]
4. Valgrind quick start [http://valgrind.org/docs/manual/quick-start.html]

sd-ca/2018/laboratoare/lab-02.txt · Last modified: 2019/02/01 13:22 by teodora.serbanescu