Laborator 1 - Introducere in C++

Responsabili

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

În cadrul acestui laborator ne propunem să ilustrăm conceptele din C++ cu care veți lucra pe parcursul acestui semestru.

Într-un mod extrem de simplist spus C++ este un superset al limbajului C, iar tot ceea ce ați învățat în C la PC [https://ocw.cs.pub.ro/courses/programare] se poate compila cu un compilator pentru limbajul C++, funcționalitatea rămânând aceeași.

Obiective

Ne dorim să:

- Realizăm tranziția de la C la C++
- Înțelegem ce presupune definirea unei clase
- Învățăm ce înseamnă constructor / destructor

De ce C++?

Pentru că C++ permite implementarea structurilor de date cu tipuri de date generice, prin intermediul template-urilor, într-un mod care nu presupune trecerea la programarea orientată pe obiecte. În cadrul acestui laborator nu ne așteptăm să dobândiți cunoștințe (elementare sau avansate) legate de programarea obiectuală, întrucât în anul II există un curs [http://elf.cs.pub.ro/poo] dedicat acestui lucru.

Vă încurajăm însă să citiți cât mai multe despre C++ pe parcurs și să cereți lămuriri suplimentare din partea asistenților de la laborator.

Sintaxa C++

De la structuri C la clase C++

Definirea structurii

În cadrul laboratorului de Programarea Calculatoarelor am învățat să declarăm și să folosim tipuri de date complexe, structuri [https://ocw.cs.pub.ro/courses/programare/laboratoare/lab11] în limbajul C. Pentru a recapitula, iată mai jos un exemplu simplu de astfel de structură, pentru a reprezenta un număr complex.

complex.h

```
#ifndef __COMPLEX_H__
#define __COMPLEX_H__

struct complex {
    double re;
    double im;
};

#endif // __COMPLEX_H__
```

Accesarea membrilor

Tipul de date definit mai sus, ca orice alt tip de date, poate fi folosit drept:

- o variabilă locală, alocată pe stivă [http://en.wikipedia.org/wiki/Stack-based_memory_allocation]
  - accesarea membrilor se face cu operatorul **.** (referențierea structurii)
  - number.re (membrul *re* al obiectul *number*)
- un pointer către o zonă alocată dinamic, pe heap [http://en.wikipedia.org/wiki/Heap-based_memory_allocation#Dynamic_memory_allocation].
  - accesarea membrilor se face cu operatorul → (dereferențierea structurii)
  - pNumber→re (membrul *re* al obiectul indicat de *pNumber*)

Mai jos puteți urmări un exemplu în acest sens:

main.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include "complex.h"

int main() {
    // Variabila locala de tip struct complex
    struct complex number;
    number.re = 0;
    number.im = 1;

    // Variabila de tip pointer, catre o zona de memorie alocata dinamic
    struct complex *pNumber = (struct complex *)malloc(sizeof(struct complex));
    pNumber->re = 1;
    pNumber->im = 0;

    free(pNumber);

    return 0;
}
```

Metode

Metodele sunt funcții specifice unui anumit tip de date. O metodă poate fi apelată prin operatorul de referențiere a structurii.

complex.h

```
struct complex {
    double re;
    double im;

    void show();
    complex conjugate();
};
```

complex.cpp

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

void complex::show() {
    if (im > 0) {
        std::cout << re << "+" << im << "i" << std::endl;
    } else {
        std::cout << re << im << "i" << std::endl;
    }
}

complex complex::conjugate() {
    complex conj;
    conj.re = re;
    conj.im = -im;
    return conj;
}
```

main.cpp

```
#include "complex.h"
int main() {
    complex number;
    number.re = 2;
    number.im = 5;
    number.show();

    complex conj = number.conjugate();
    conj.show();
    return 0;
}
```

Se va afișa:

```
2+5i
2-5i
```

Clase

Formal am făcut deja primii pași mai sus pentru a implementa o clasă în C++, utilizând keyword-ul *struct*.

Totuși, ce înseamnă o clasă? Nu trebuie decât să ne gândim la ce am făcut mai sus:

- am definit un tip de date
- i-am adăugat atribute (am definit ce proprietăți îl caracterizează: partea reală și partea imaginară)
- i-am adăugat metode (am definit cum se comportă: afișare și conjugare)

Cu această adăugare menționată, putem să ne referim la ceea ce înseamnă o **clasă**, respectiv un **obiect**.

Ne referim la o **clasă** ca fiind o amprentă (blueprint) sau descriere generală. Un **obiect** sau o **instanță a clasei** este o variabilă concretă ce se conformează descrierii clasei.

Vom numi **clasă** tipul de date definit de *struct complex* sau *class complex* și **obiect** o instanțiere (o alocare dinamică sau locală) a tipului de date.

Când discutăm despre tipul de date *complex* ne referim la clasă. Când discutăm despre variabila *number* ne referim la un obiect, o instanță a clasei.

Keyword-ul "class" vs. "struct"

Și totuși, C++ adăugă keyword-ul *class*. Care este diferența între *class* și *struct*? Iată cum definim complet clasa de mai sus, separând antetul de implementare și de programul principal.

complex.h

```
class Complex {
    double re;
    double im;

    Complex conjugate();
    void show();
};
```

complex.cpp

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

Complex Complex::conjugate() {
    Complex conj;
    conj.re = re;
    conj.im = -im;

    return conj;
}

void Complex::show() {
    if (im > 0) {
```

```
        std::cout << re << "+" << im << "i" << std::endl;
    } else {
        std::cout << re << im << "i" << std::endl;
    }
}
```

main.cpp

```
#include "complex.h"

int main() {
    Complex number;
    number.re = 2;
    number.im = 4;
    number.show();

    Complex conj = number.conjugate();
    conj.show();

    return 0;
}
```

Compilare

Sursele C++ se compilează folosind compilatorul **g++**. Acesta permite exact aceleași opțiuni de bază ca și **gcc**, compilatorul utilizat pentru sursele de C.

- Încercați să compilați și să rulați codul din cele 3 fișiere de mai sus.

```
g++ complex.cpp main.cpp -o exemplu
```

Ce observați?

Înlocuiți acum keyword-ul *class* cu keyword-ul *struct* și compilați din nou.

Specificatori de acces

Am observat mesajul de eroare în urma compilării fișierelor de mai sus.

Astfel, **singura diferență** folosirea celor două keyword-uri este nivelul implicit de vizibilitate a metodelor și atributelor.

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

Membrii și metodele precedați de label-ul **private** pot fi folosite numai în interiorul clasei, în cadrul metodelor acesteia (o metodă privată poate fi apelată doar de o alta metodă din cadrul clasei).

Membrii nu pot fi citiți sau modificați în mod default din afara clasei.

Iată cum puteam remedia soluția:

complex.h

```
class Complex {
    double re;
    double im;

public:
    Complex conjugate();
    void show();
};
```

This

C++ se ocupă (în spate, fără intervenția noastră) să paseze un parametru extra oricărei metode apelate (dacă aceasta nu este statică [https://www.tutorialspoint.com/cplusplus/cpp_static_members.htm]).

El este un pointer catre obiectul curent și se numește this. Nu face parte efectiv din semnătura metodei și este un cuvânt cheie (rezervat) în C++, deci aveți grijă cum vă numiți variabilele.

complex.h

```
#include <iostream>

class Complex {
    double re;
    double im;

public:
    void initialize(double re, double im) {
        this->re = re;
        this->im = im;
    }

    Complex conjugate() {
        Complex conj;
        conj.initialize(re, -im);

        return conj;
    }

    void show() {
        if (im > 0) {
            std::cout << re << "+" << im << "i" << std::endl;
        } else {
            std::cout << re << im << "i" << std::endl;
        }
    }
};
```

main.cpp

```
#include "complex.h"

int main() {
    Complex number;
    number.initialize(2, 3);
    Complex conj;
    conj = number.conjugate();
    conj.show();
    return 0;
}
```

Se va afișa:

```
2+3i
2-3i
```

Constructori și destructori

Studiați codul de mai jos.

complex.h

```
class Complex {
public:
    // Constructor
```

complex.cpp

```
#include "complex.h"
Complex::Complex(double re, double im) {
    this->re = re;
```

```
    Complex(double re, double im);

    // Destructor
    ~Complex();

    double getRe();
    double getIm();

    Complex conjugate();

private:
    double re;
    double im;
};
```

```
    this->im = im;
}

Complex::~Complex() {
}

Complex Complex::conjugate() {
    Complex conjugat(re, -im);
    return conjugat;
}

double Complex::getRe() {
    return re;
}

double Complex::getIm() {
    return im;
}
```

main.cpp

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

int main() {
    Complex number(2, 3);
    std::cout << number.getRe() << " " << number.getIm()) << std::endl;

    return 0;
}
```

Constructor

Observăm două bucăți din cod în mod special:

```
Complex::Complex(double re, double im);
```

Linia de mai sus **nu are tip** returnat, spre deosebire de celelalte linii. Acesta este **constructorul** clasei, care este apelat în momentul alocării unui obiect.

Ce operații sunt uzuale în constructor?

- inițializarea membrilor clasei cu valori predefinite sau date ca parametru
- alocarea memoriei pentru anumiți membri

A doua bucată observată este:

```
Complex numar(2, 3);
```

Până acum nu ați mai alocat astfel structurile. Ce se întâmplă în spate este exact ceea ce intuiți: este apelat constructorul obiectului și se execută instrucțiunile acestuia pentru variabila numar (reprezentată ca pointer prin this, direct în interiorul constructorului).

În constructorul definit mai sus, tot ceea ce se întâmplă este să se inițializeze membri. Pentru asta, C++ vă pune la dispoziție o sintaxă simplă:

```
Complex::Complex(double real, double imaginar) :
    re(real),
    im(imaginar) {
}

// SAU

Complex::Complex(double real, double imaginar) {
```

```
    this->re = real;
    this->im = imaginar;
}
```

Cei doi constructori sunt (aproape) identici ca funcționalitate.

Destructor

Așa cum probabil ați observat, **constructorul** este apelat în mod **explicit** de către voi. **Destructorul** însă, în cazul de mai sus, este apelat **implicit** la terminarea blocului care realizează dealocărea automată a obiectului.

Un destructor nu are parametri și se declară în interiorul clasei astfel:

```
~Complex();
```

Dacă în constructor sau în interiorul clasei ați fi alocat memorie, cel mai probabil în destructor ați fi făcut curat și ați fi apelat delete pe membrul respectiv.

Constructor/destructor implicit

În cazul în care utilizatorul nu definește niciun constructor, va exista un constructor default creat de compilator. Acesta are doar comportamentul implicit al unui constructor : apelează constructorul default (fără parametri) al membrilor clasei de tip complex (class, struct, union).

Asemanător, în cazul în care nu este definit niciun destructor, se va crea un destructor implicit care aplează destructorul default al membrilor clasei (al caror tip nu este primitv).

Dacă exista un constructor sau destructor definit de utilizator, atunci compilatorul nu va mai genera unul implicit.

objects.h

```
#include <iostream>
class Foo{
public:
    Foo() {
        std::cout << "Default constructor of Foo" << std::endl;
    }

    ~Foo() {
        std::cout << "Default destructor of Foo" << std::endl;
    }
};

class Base {
    int i;
    Foo f;
public:
    void sayHello() {
         std::cout << "Hello from Base" << std::endl;
    }
};
```

main.cpp

```
#include "objects.h"
int main() {
    Base b; // Se va apela constructorul default al clasei Base;
    b.sayHello();

    // Se va apela destructorul default al clasei Base;

    return 0;
}
```

Se va afișa:

```
Default constructor of Foo
Hello from Base
Default destructor of Foo
```

Alocarea / Dealocarea dinamică

C++ introduce perechea de keyword-uri *new* și *delete*, care se folosesc pentru a aloca dinamic instanțe ale claselor.

```
Complex *numar = new Complex(2, 3);
delete numar;
```

Keyword-ul *new* apelează constructorul clasei, iar keyword-ul *delete* apelează destructorul clasei.

Observație

- Un obiect se alocă/dezalocă cu combinația *new/delete*
- Un vector de obiecte se alocă/dezalocă cu *combinația new[]/delete[]*

```
Complex *numere = new Complex[10];
delete[] numere;
```

Dacă dorim să alocăm memorie dinamică în mai multe dimensiuni, vom folosi o procedură asemănătoare cu cea folosită în C. Presupunând că vrem să alocăm o matrice de dimensiune NxM atunci declarăm un dublu pointer, alocăm N pointeri pentru linii și apoi M elemente pentru fiecare linie în parte.

```
int **mat;
mat = new int*[N];

for (int i = 0; i < N; i++) {
    mat[i] = new int[M];
}

// ..
// Folosire matrice
// ..

// Dealocare
for (int i = 0; i < N; i++) {
    delete[] mat[i];
}
delete[] mat;
```

În cazul de mai sus, N și M nu trebuie sa fie constante, pot fi (de exemplu) variabile citite de la tastatură.
Atenție Memoria alocată nu va fi toată într-o zonă continuă (doar elementele unei linii alocate cu `new int[M]` vor fi continue, dar între două linii consecutive pot exista "spații" în memorie).

Dacă toate dimensiunile (mai puțin prima) sunt constante (cunoscute la compile time) se poate aloca dinamic memoria în felul următor:

```
int (*mat)[100] = new int[N][100];
// mat va fi alocata în zona de heap (nu pe stivă), într-o zonă continuă de memorie

// ..
// Folosire matrice
// ..

// Dealocare
delete[] mat;
```

Atenție Nu convertiți matricea de mai sus la un pointer dublu, deoarece cele două nu sunt același lucru (prima este un vector de N pointeri care pointează către liniile matricei, pe când a doua este o zonă continuă de memorie în care compilatorul accesează elementele la fel ca într-o matrice clasică).

Keyword auto

Odată cu C++11, keyword-ul *auto* poate fi folosit pentru deducerea implicită a tipului unei variabile încă de la inițializarea acesteia, nemaifiind nevoie să specificăm tipul unor date pe care compilatorul deja le cunoaște. Acest lucru ne poate salva puțin timp atunci când avem de-a face cu typename-uri foarte lungi sau greu de urmărit și ne obligă să inițializăm variabilele de acel tip (fără inițializare în momentul declarării unei variabile, compilatorul nu ar avea de unde să știe ce tip să îi atribuie).

Spre exemplu, în cadrul unei funcții putem avea:

```
double d = 5.0; // Menționăm explicit tipul de date al lui d
```

Sau, folosind *auto*:

```
auto d = 5.0; // 5.0 este un literal de tip double, deci și d va fi tot double
```

În C++14 s-a extins funcționalitatea lui *auto*, putând fi utilizat și pentru return type-ul unor funcții:

```
auto computeSum(int a, int b) {
    return a + b;
}
```

Observații:

- *auto* se folosește cel mai frecvent în cadrul for-urilor, așa cum vom vedea la partea de probleme
- nu putem folosi *auto* în cadrul parametrilor unor funcții (la compile-time nu s-ar putea deduce tipul acestora)
- În C++11 se poate folosi *auto* ca și return type doar în semnăturile funcțiilor, compilatorul nemaifăcând efectiv deducerea tipului de date returnat (se specifică la final, ca în exemplul de aici [https://cppstyle.wordpress.com/trailing-return-type/]).

Cum compilăm folosind C++11? Prin plasarea flag-ului corespunzător astfel:

```
g++ -std=c++11 main.cpp -o exemplu
```

Schelet

Schelet

Exerciții

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.

Tipul pereche

- În biblioteca standard a limbajului este definit tipul pereche.
- O variabilă de tip pair se declară astfel:

```
// std::pair<tip_element1, tip_element2> p;
```

```
// Exemple:
std::pair<int, float> p1;
std::pair<float, Complex> p2;
```

- Mai multe despre template-uri veți afla în laboratorul 3
- Pentru a crea o pereche:

```
std::pair<int, char> p = std::make_pair(1, 'x');
```

- Accesarea elementelor perechii:

```
std::pair<int, float> p;
int first = p.first;
float second = p.second;
```

1) [**2p**] În fișierul Auto.cpp, creați un vector de perechi de 10 elemente alocat static. Primul câmp va fi int și va avea valoarea poziției în vector, iar al doilea va fi float și va avea valoarea poziției în vector inmulțită cu 2.5. Parcurgeți vectorul utilizând auto și afisați pentru fiecare pereche primul câmp (acestea vor fi separate prin spațiu). La final, afișați suma obținută prin adunarea celui de-al doilea câmp din fiecare pereche, separată de asemenea prin spațiu.

2) [**4p**] Clasa Complex

- [**1p**] Adăugați clasei Complex din fișierul Complex.h constructorul și getterii indicați.
- [**3p**] Adăugați metode pentru modulul numărului, adunare și înmulțire cu un alt număr complex.

3) [**4p**] Clasa Fracție

- [**1p**] Adăugați clasei Fracție din fișierul Fraction.h constructorul și getterii indicați.
- [**3p**] Adăugați clasei Fracție metode pentru înmulțire și împărțire cu o altă fracție, respectiv pentru determinarea numărului zecimal echivalent.

4) [**4p**] Clasa Gradebook conține un vector de valori double reprezentând notele din catalog, capacity reprezentând capacitatea maximă a catalogului și count reprezentând numărul de note adăugate in catalog.

- [**1p**] Adăugați un constructor care primește ca parametru capacitatea dorită, alocând memorie pentru vectorul grades și inițializând valorile capacity și count. Adăugați un destructor care eliberează memoria.
- [**1p**] Adăugați o metodă care inserează o notă în catalog (dacă se dorește adaugarea unei note după ca s-a atins capacitatea maximă, puteți să afișați un mesaj de eroare și să ieșiți din metodă)
- [**1p**] Adaugați o metodă care calculează media aritmetică a notelor prezente in catalog.
- [**1p**] Verificați cu Valgrind că nu aveți memory leaks.

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 între struct și class în C++?
- Ce face keyword-ul static în fața metodei unei clase în C++?
- Ce este diferit între o metodă statică și o metodă normală a unei clase? (Hint: explicați cum e cu pointer-ul *this*)

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-01.txt · Last modified: 2019/02/01 13:19 by teodora.serbanescu