Laborator 01 - Introducere in C++

Responsabili

- Octavian Rînciog [mailto:mailto:octavian.rinciog@cs.pub.ro] (2013)
- Victor Cărbune [mailto:mailto:victor.carbune@cti.pub.ro] (2012)

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 [http://ocw.cs.pub.ro/courses/programare-ca] 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
- Folosim template-uri

De ce C++?

Pentru că C++ permite implementarea structurilor de date cu tipuri de date generice, prin intermediul template-urilor.

În cadrul acestui laborator nu ne așteptăm să dobândiți cunoștințe (elementare sau avansate) legate de programarea pe obiecte, întrucât în anul II există un curs 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 [http://ocw.cs.pub.ro/courses/programare-ca/laboratoare/lab10] 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 (automată [http://en.wikipedia.org/wiki/Automatic_variable]), 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.cc

```
#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;
}
```

Funcții specifice structurii

Dându-se o variabilă de tip *struct complex* vom dori să efectuăm diferite operații asupra acesteia.

complex.h

```
#ifndef __COMPLEX_H
#define __COMPLEX_H

struct complex {
    double re;
    double im;
};

// Initializeaza campurile unei structuri date.
void complex_initialize(struct complex *number, double re, double im);

// Intoarce o structura ce contine numarul complex conjugat.
struct complex complex_conjugate(struct complex *number);

#endif // __COMPLEX_H
```

Ce observăm că au în comun cele două funcții? Hint: primul parametru, care reprezintă un pointer către o zonă de memorie care reține tipul *struct complex*

Metode

Acest pattern de a defini funcții specifice unui anumit tip de date este extrem de întâlnit. De asemenea, se observă că aceste funcții nu ar putea fi folosite în combinație cu alte structuri de date, ele fiind specifice *struct complex*.

Așadar, iată cum o structură poate să capete metode specifice (codul de mai jos este specific C++):

complex.h

```
#ifndef __COMPLEX_H
#define __COMPLEX_H

struct complex {
    double re;
    double im;

    void complex_initialize(double re, double im);
    struct complex complex_conjugate();
};
```

```
#endif // __COMPLEX_H
```

Astfel, avem definite metode, care operează pe tipul de date respectiv, și care pot fi apelate întocmai cum se realiza accesarea membrilor de date. Observăm însă că a dispărut primul parametru! De ce?

Întrucât metodele definite mai sus pot fi apelate numai pe o variabilă de tip *struct complex*, C++ transmite ascuns în spate un pointer **struct complex *this** care poate fi folosit pentru a accesa câmpurile variabilei date.

Iată implementarea metodelor în antetul definit mai sus.

Compilați și rulați codul de mai jos cu **g++**.

complex.h

```
#ifndef __COMPLEX_H
#define __COMPLEX_H

struct complex {
    double re;
    double im;

    void complex_initialize(double re, double im) {
        this->re = re;
        this->im = im;
    }

    struct complex complex_conjugate() {
        struct complex conjugate;
        conjugate.complex_initialize(this->re, -(this->im));

        return conjugate;
    }
};
#endif // __COMPLEX_H
```

main.cc

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

int main() {
    struct complex number;

    number.complex_initialize(2, 3);
    printf("%.2lf %.2lf\n", number.re, number.im);

    number.complex_initialize(5, 6);
    printf("%.2lf %.2lf\n", number.re, number.im);

    return 0;
}
```

Clase

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

Dar totuși, ce înseamnă o clasă? Și aici nu trebuie decât să ne gândim la ce am făcut mai sus pentru a implementa o clasă:

- am adăugat atribute (am definit ce proprietăți caracterizează clasa: partea reală și partea imaginară)
- am adăugat metode (am definit cum se comportă clasă: inițializarea și conjugarea)

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 sau descriere generală. Un **obiect** este o instanță a clasei respective - o variabilă concretă ce este de tipul 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();
};
```

complex.cc

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

    return conjugate;
}
```

main.cc

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

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

    printf("%.2lf %.2lf\n", number.re, number.im);

    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.cc main.cc -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-ul este nivelul implicit de vizibilitate a metodelor și atributelor.

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

Membri precedați de label-ul **private** pot fi folosiți numai în interiorul clasei, în cadrul metodelor acesteia. Ei nu pot fi citiți sau modificați din afara clasei.

Iată cum puteam remedia soluția:

complex.h

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

    Complex conjugate();
};
```

Constructori și destructori

Studiați codul de mai jos.

complex.h

```
class Complex {
public:
    // Constructor
    Complex(double re, double im);

    // Destructor
    ~Complex();

    double getRe();
```

complex.cc

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

Complex::~Complex() {
}
```

```
    double getIm();

    Complex conjugate();

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

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

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

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

main.cc

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

int main() {
    Complex number(2, 3);
    printf("%lf %lf\n", number.getRe(), number.getIm());

    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) {
}
```

Cei doi constructori sunt 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 free pe membrul respectiv.

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;
```

Templates

Motivul principal pentru care folosim C++ în cadrul SD este datorită funcționalității oferite de template-uri.

Acestea permit generalizarea tipurilor de date folosite în interiorul funcțiilor și claselor.

Sintaxa pentru acestea este:

```
template <class identifier> declaratie;
template <typename identifier> declaratie;
```

declaratie poate fi fie o funcție, fie o clasă. Nu există nicio diferență între keyword-ul *class* și *typename* - important este că ceea ce urmează după ele este un placeholder pentru un tip de date.

Function Template

În primul rând template-urile pot fi aplicate funcțiilor.

Un exemplu comun și simplu este următorul:

```
template<typename T>
T getMax(T a, T b) {
    return a > b ? a : b;
}
```

Funcția poate fi apelată astfel:

```
getMax<int>(2, 3);
getMax<double>(3.2, 4.6);
```

Class Template

Concret, să presupunem că avem o clasă numită KeyStorage care are:

- o cheie (de tip int)
- un membru de date generic (al cărui tip de date nu îl știm la momentul scrierii clasei).

Vrem să putem folosi codul clasei indiferent de tipul de date al membrului.

Iată cum putem face acest lucru:

KeyStorage.h

```
template<typename T>
class KeyStorage {
public:
    int key;
    T member;
};
```

În funcția main, să presupunem că vrem să folosim clasa cu membrul de tip long.

main.cc

```
#include "KeyStorage.h"

int main() {
    KeyStorage<long> keyElement;
    return 0;
}
```

Practic, oriunde folosim tipul de date T în clasă, este înlocuit cu tipul pe care îl specificăm.

Where's the magic happening?

Sunt destul de multe lucruri de spus despre template-uri, dar ne vom concentra pe lucrurile care schimbă modul în care ați implementat până acum.

Template-urile sunt de fapt indicii pentru compilator pentru a genera cod la rândul lui! Practic, voi îi spuneți compilatorului un șablon generic pe care ați vrea să-l folosiți și el trebuie să fie pregătit să îl pună la dispoziția voastră când aveți nevoie.

Ce trebuie să rețineți din asta? Totul se întâmplă la **compile time**, nu la run time.

Compilatorul practic analizează modul în care voi folosiți clasa respectivă și generează pentru fiecare mod în care o folosiți șablonul corespunzător. Folosirea KeyStorage<int> și KeyStorage<float> determină compilatorul să genereze cod pentru ambele clase (înlocuind o dată T cu int și altă cu long).

Guideline-uri implementare

Pentru că totul se întâmplă la compile time, înseamnă că în momentul în care compilatorul întâlnește secvența de cod ce folosește template-uri trebuie să știe *toate* modurile în care aceasta este folosita.

Asta înseamnă că:

- Trebuie să scrieți întreaga implementare în header! *sau*
- Scrieți descrierea clasei generic în header, în fișierul de implementare fiecare metodă declarată este de fapt o funcție cu template și la sfârșitul implementării adăugat *template class numeclasa<numetip>*;

Ultimul rând de fapt forțează folosirea template-ului cu un anumit tip de date și deci compilatorul generează cod corespunzător (trebuie să scrieți asta pentru toate tipurile.

Clasa KeyStorage

Iată mai jos o structură mai dezvoltată pentru clasa KeyStorage, în care cheia este setată în constructor. .

KeyStorage.h

```
template<typename T>
class KeyStorage {
public:
    KeyStorage(int k);
    ~KeyStorage();

    T getMember();
    T setMember(T element);

private:
    T member;
    int key;
};
```

Implementarea completa a ei poate fi realizată:

- în header (în cazul template-urilor, acest mod este cel mai indicat).
- în fișierul de implementare .cc / .cpp (al cărui schelet parțial îl găsiți mai jos).

KeyStorage.cpp

```
#include "KeyStorage.h"

template<typename T>
KeyStorage<T>::KeyStorage(int k) {
//TODO
}
```

```
template<typename T>
KeyStorage<T>::~KeyStorage() {
}

//TODO: restul metodelor.

// La sfarsit, cu tipurile de date pe care le veti folosi.
template class KeyStorage<int>;
template class KeyStorage<long>;
```

Exercitii

- [**2p**] Parcurgeți laboratorul în întregime și discutați cu asistentul vostru eventualele neclarități.
- [**3p**] Clasa Complex
  - [**2p**] Adăugați clasei Complex metode pentru adunare, scădere și înmulțire cu un alt număr complex.
  - [**1p**] Arătați funcționalitatea prin adăugarea unui cod simplist în fișierul main.cc
- [**5p**] Clasa KeyStorage
  - [**2p**] Implementați și folosiți utilizând template-uri clasa KeyStorage de mai sus adăugând constructor, destructor.
  - [**2p**] Alocați o instanță de tip KeyStorage local și dinamic (utilizând new / delete).
  - [**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 înseamnă specializarea unui template? Cum se realizează aceasta?
- Ce este o clasă abstractă?
- Când este indicat să folosim template-uri versus clase abstracte?
- 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. Templates [http://www.cplusplus.com/doc/tutorial/templates]
3. Standard C++98 [http://sites.cs.queensu.ca/gradresources/stuff/cpp98.pdf]
4. Standard C++11 [https://github.com/cplusplus/draft/blob/master/papers/N3485.pdf]

sd-ca/laboratoare/laborator-01.txt · Last modified: 2013/02/19 10:31 by andrei.parvu