C++ DevCamp Organizatoric

Desfasurarea Cursului

Cursul se va desfășura în decursul a 6 săptămâni, în sesiuni a cate 3h.

- Vineri, 26 martie - orele 10^{00} - 13^{00}
- Joi, 1 aprilie - orele 10^{00} - 13^{00}
- Vineri, 9 aprilie - orele 10^{00} - 13^{00}
- Vineri, 16 aprilie - orele 10^{00} - 13^{00}
- Vineri, 23 aprilie - orele 10^{00} - 13^{00}
- Vineri, 7 mai - orele 10^{00} - 14^{00}

March

Sun	Mon	Tue	Wed	Thu	Fri	Sat
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31			

April

Sun	Mon	Tue	Wed	Thu	Fri	Sat
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	

May

Sun	Mon	Tue	Wed	Thu	Fri	Sat
						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31					

Structura Cursului

Fiecare modul consta in o parte teoretica (~2h) si una practica (~1h).

Modulul 1

- Clase, obiecte, constructori, destructori, etc

Modulul 2

- Mostenire, polimorfism, virtualizare.

Modulul 3

- Memorie, pointeri, copy constructor, move constructor, etc.

Modulul 4

- Templateuri.

Modulul 5

- Smart pointeri.

Modulul 6

- Functii, functii anonime, etc.

C++ DevCamp
Module 1 | Notiuni Introductive

De ce sa înveți C++?

- Limbaj de programare popular
  - In baze de date
  - In sisteme de operare
  - In compilatoare
  - In grafica
  - In sisteme embeded
- Portabil, eficient.

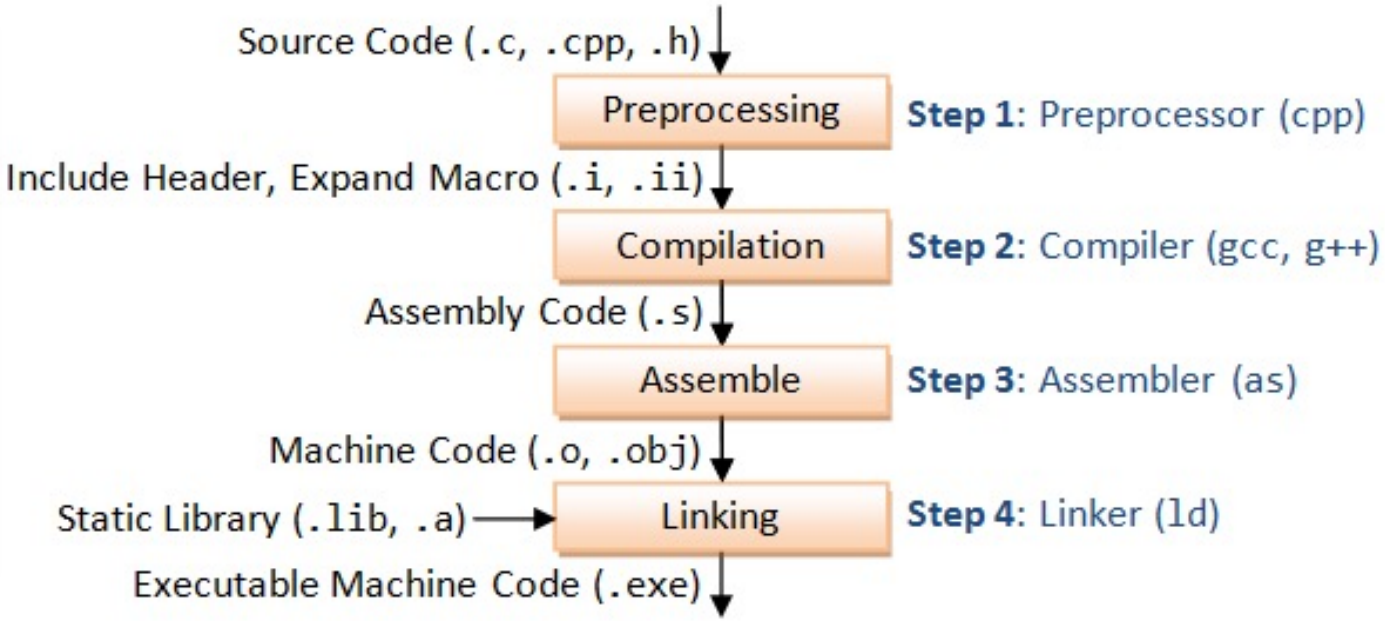
Fazele procesului de compilare

Fazele procesului de compilare

Procesul de compilare (sau "**Compilarea**"[1] mai scurt) se referă la obținerea unui fișier executabil dintr-un fișier sursă.

4 faze, evidențiate în diagrama alaturata:

1. Preprocesarea
2. Compilarea
3. Asamblarea
4. Editarea de legături



1) A nu se confunda cu etapa 2 cu acelasi nume.

Fazele procesului de compilare | 1. Preprocesarea

Preprocesarea presupune înlocuirea directivelor de preprocesare din fişierul sursă C.

Directivele de preprocesare încep cu #.
Printre cele mai folosite sunt:

- **#include** – pentru includerea fişierelor header într-un alt fișier.
- **#define** și **#undef** – pentru definirea, respectiv anularea definirii de macrouri.
- **#if**, **#ifdef**, **#ifndef**, **#else**, **#elif**, **#endif** – pentru compilarea condiţionată.

Utilizare:

- pentru comentarea bucăților mari de cod.
- pentru evitarea includerii de mai multe ori a unui fişier header, tehnică numită include guard.
- **__FILE__**, **__LINE__**, **__func__** sunt înlocuite cu numele fişierului, linia curentă în fișier şi numele funcției
- **operatorul #** este folosit pentru a înlocui o variabilă transmisă unui macro cu numele acesteia.
- debugging

Fazele procesului de compilare | Compilarea

Compilarea este faza în care din fişierul preprocesat se obţine un fişier în limbaj de asamblare.

Fazele procesului de compilare | Asamblarea

Asamblarea este faza în care codul scris în limbaj de asamblare este tradus în cod mașină reprezentând codificarea binară a instrucțiunilor programului inițial.

Fişierul obţinut poartă numele de fişier cod obiect.

Fazele procesului de compilare | Editarea de legături

Pentru obținerea unui fişier executabil este necesară rezolvarea diverselor simboluri prezente în fişierul obiect.

Această operaţie poartă denumirea de editare de legături, link-editare, linking sau legare.

Structuri si clase C++

Structurii

Utilizând keyword-ul ***struct***.

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

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

Clase

Utilizând keyword-ul ***class***.

```
class complex {
        double re;
        double im;

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

"class" vs. "struct"

Singura diferență in 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**

Pointerul this

- Este un cuvânt cheie (rezervat) în C++.
- "this" este un pointer catre obiectul curent.
- Nu face parte din semnătura metodei.

Constructori și destructori

Constructorul

- Este apelat în momentul alocării unui obiect.
- Nu are tip returnat.
- Operații sunt uzuale în constructor:
  - inițializarea membrilor clasei cu valori predefinite sau date ca parametru
  - alocarea memoriei pentru anumiți membri

Destructorul

- 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.
- **Constructorul** este apelat în mod **explicit** de către noi.
- **Destructorul** însă este apelat **implicit** la terminarea blocului care realizează dealocărea automată a obiectului.

Constructor/destructor implicit

- În cazul în care utilizatorul nu definește niciun constructor, va exista un constructor default (fara parametrii) 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).
- Dacă exista un constructor sau destructor definit de utilizator, atunci compilatorul nu va mai genera unul implicit.

Defaulted and Deleted members (C++11)

- Incepand cu C++11, putem forta compilatorul sa defineasca sau sa stearga o functie speciala (constructor/destructor/etc).
- MyClass() = default;

  ⇒ am adaugat "= default;" la finalul functiei ca sa fortam compilator sa defineasca functia in mod explicit ca pe o functie default;
- MyClass(const MyClass&) = deleted;

  ⇒ am adaugat "= deleted;" la finalul functiei ca sa fortam compilator sa dezactiveze functia

Defaulted and Deleted members (C++11) (2)

```cpp
// C++ code to demonstrate the
// use of defaulted functions
#include <iostream>
using namespace std;

class A {
public:

    // A user-defined
    // parameterized constructor
    A(int x)
    {
        cout << "This is a parameterized constructor";
    }

    // Using the default specifier to instruct
    // the compiler to create the default
    // implementation of the constructor.
    A() = default;
};

int main()
{
    // executes using defaulted constructor
    A a;

    // uses parametrized constructor
    A x(1);
    return 0;
}
```

```cpp
// C++ program to disable the usage of
// copy-constructor using delete operator
#include <iostream>
using namespace std;

class A {
public:
    A(int x): m(x)
    {
    }

    // Delete the copy constructor
    A(const A&) = delete;

    // Delete the copy assignment operator
    A& operator=(const A&) = delete;
    int m;
};

int main()
{
    A a1(1), a2(2), a3(3);

    // Error, the usage of the copy
    // assignment operator is disabled
    a1 = a2;

    // Error, the usage of the
    // copy constructor is disabled
    a3 = A(a2);
    return 0;
}
```

Copy constructorul

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

Copy constructorul

- Când se apelează:
  - Apel explicit
    ```
    MyClass m;
    MyClass x = MyClass(m);
    ```
  - Transfer prin valoare ca argument într-o funcție
    ```
    void foo(MyClass obj);
    MyClass o; f(o);
    ```
  - Transfer prin valoare ca return al unei funcții
  - La inițializarea unei variabile declarate pe aceeași linie
    ```
    MyClass m;
    MyClass x = m;
    ```

Rule of Three

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

Keywords

const

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

https://en.cppreference.com/w/cpp/keyword/const

const (3)

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

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.

https://en.cppreference.com/w/cpp/keyword/const

static

In interiorul unei clase, specifică un obiect a cărui valoare nu este legata de instantierea clasei

⇒ membrii statici

static (2)

- Când creăm o instanță a unei clase, valorile câmpurilor din cadrul instanței sunt unice pentru aceasta și pot fi utilizate fără pericolul ca instanţierile următoare să le modifice în mod implicit.
- Există însă posibilitatea ca uneori, anumite câmpuri din cadrul unei clase să aibă valori independente de instanțele acelei clase, astfel că acestea nu trebuie memorate separat pentru fiecare instanță.
- Aceste câmpuri se declară cu atributul **static** și au o locație unică în memorie, care nu depinde de obiectele create din clasa respectivă.
- Pentru a accesa un câmp static al unei clase (presupunând că acesta nu are specificatorul private), se face referire la clasa din care provine, nu la vreo instanță.
- Același mecanism este disponibil și în cazul metodelor.

static (3) – Singleton design pattern

- Pattern-ul Singleton este utilizat pentru a restricționa numărul de instanțieri ale unei clase la un singur obiect, deci reprezintă o metodă de a folosi o singură instanță a unui obiect în aplicație.
- Aplicarea pattern-ului Singleton constă în implementarea unei metode ce permite crearea unei noi instanțe a clasei dacă aceasta nu există, și întoarcerea unei referințe către aceasta dacă există deja.
- În C++, pentru a asigura o singură instanțiere a clasei, constructorul trebuie să fie *private*, iar instanța să fie oferită printr-o metodă statică, publică.

```
class MySingleton{

  private:
    static std::once_flag initInstanceFlag;
    static MySingleton* instance;
    MySingleton()= default;
    ~MySingleton()= default;

  public:
    MySingleton(const MySingleton&)= delete;
    MySingleton& operator=(const MySingleton&)= delete;

    static MySingleton* getInstance(){
      std::call_once(initInstanceFlag,MySingleton::initSingleton);
      return instance;
    }

    static void initSingleton(){
      instance= new MySingleton();
    }
};

MySingleton* MySingleton::instance= nullptr;
std::once_flag MySingleton::initInstanceFlag;
```

static (4) – Meyer's Singleton

- Exista mai multe implementari ale acestui design pattern.
- Meyers Singleton (numit dupa Scott Meyers) este cea mai eleganta implementare thread safe a pattern-ului singleton. Ea se bazeaza pe faptul ca o variabila statica definita in interiorul unui context va fi creata exact o singura data.

```
#include <thread>

class MySingleton{
public:
  static MySingleton& getInstance(){
    static MySingleton instance;
    return instance;
  }
private:
  MySingleton();
  ~MySingleton();
  MySingleton(const MySingleton&)= delete;
  MySingleton& operator=(const MySingleton&)= delete;

};

MySingleton::MySingleton()= default;
MySingleton::~MySingleton()= default;

int main(){

  MySingleton::getInstance();

}
```

Keyword-ul auto (C++11)

Introdus incepand 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).

```
double d = 5.0;
```

⇒ Menționăm explicit tipul de date al lui d

```
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:

https://en.cppreference.com/w/cpp/keyword/auto

Conversii de tip (cast-uri)

static_cast

- Folosit pentru a realiza conversia de la un tip de date la altul.
- Realizat la compilare. Fara verificari la runtime.

```
new_type value = static_cast <new_type> (expression);
```

- Folosit pentru conversia intre tipuri primitive (de la int la float, sau invers) sau pentru a face conversia de la clasa de baza la o clasa derivata (downcast).

```cpp
#include <iostream>
using namespace std;
int main()
{
    float f = 3.5;
    int a = f; // this is how you do in C
    int b = static_cast<int>(f);
    cout << b;
}
```

```cpp
#include <iostream>
int main()
{
    int i = 10;
    void* v = static_cast<void*>(&i);
    int* ip = static_cast<int*>(v);
    return 0;
}
```

https://en.cppreference.com/w/cpp/language/static_cast

dynamic_cast

- Folosit pentru a realiza conversia unui pointer (sau referinta) la altul, pe baza relatiei de mostenire.
- Util cand nu stim tipul dinamic al obiectului. Returneaza null pointer daca castul nu a reusit.
- Realizat la runtime

```
new_type value = dynamic_cast <new_type> (expression);
```

reinterpret_cast

- Folosit pentru a realiza conversia unui pointer la altul, indiferent de relatia dintre ei.
- Realizat la runtime.

```
new_type *var_name = reinterpret_cast < new_type *>(pointer_variable);
```

```cpp
// CPP program to demonstrate working of
// reinterpret_cast
#include <iostream>
using namespace std;

int main()
{
    int* p = new int(65);
    char* ch = reinterpret_cast<char*>(p);
    cout << *p << endl;
    cout << *ch << endl;
    cout << p << endl;
    cout << ch << endl;
    return 0;
}
```

https://www.geeksforgeeks.org/reinterpret_cast-in-c-type-casting-operators/

const_cast

- Folosit pentru a inlatura atributul const.
  - Modifica membrii care nu sunt const in functii const.
  - Pentru a trimite argumente constante unor functii care nu primesc argumente c

```c
#include <iostream>
using namespace std;

int fun(int* ptr)
{
    return (*ptr + 10);
}

int main(void)
{
    const int val = 10;
    const int *ptr = &val;
    int *ptr1 = const_cast <int *>(ptr);
    cout << fun(ptr1);
    return 0;
}
```

https://en.cppref

```
#include <iostream>
using namespace std;

class student
{
private:
    int roll;
public:
    // constructor
    student(int r):roll(r) {}

    // A const function that changes roll with the help of const_cast
    void fun() const
    {
        ( const_cast <student*> (this) )->roll = 5;
    }

    int getRoll()  { return roll; }
};

int main(void)
{
    student s(3);
    cout << "Old roll number: " << s.getRoll() << endl;

    s.fun();

    cout << "New roll number: " << s.getRoll() << endl;

    return 0;
}
```


Adobe