C++

11. HÉT – HEADER FÁJLOK MÚLTJA, JELENE ÉS JÖVŐJE

A JEGYZETET KÉSZÍTETTE:

CSAPÓ ÁDÁM BALÁZS ÉS ŐSZ OLIVÉR

Múlt: A header fájlok megjelenése

A C nyelvből erednek, ahol minden forrásfájl egy önálló fordítási egység
- Csak azt kell újrafordítani, ami változott

A függvényhívás helyén csak a függvény deklarációjára van szükség fordításkor
- Külön fájlokba került a deklaráció és implementáció (definíció)
- Kevesebb kód kerül include-olásra, így gyorsabb a fordítás
- Lehetővé teszi a library-k linkelését
- De:
  - A template függvényeket a headerben kell definiálni
  - Az inline definíció több optimalizálási lehetőséget nyújt a fordítónak

Ha a header változik, akkor újra kell fordítani mindent, ami include-olja
- Az interfész ritkán változik
- Csak azt és ott include-oljuk, amit muszáj
  - Amit lehet, a cpp-ben, ne a headerben include-oljunk

Precompiled header

A header fájlok használata segít, hogy ne kelljen minden újrafordítani, de amit kell, annál redundanciát jelenthet

- Sok, nagyméretű header fájlok, amik sok helyen vannak include-olva
- Minden fordítási egységnél újra feldolgozza őket a fordító

A megoldás az „előfordítás”

- Előre lefordítjuk a headert, így azt már nem kell minden fordítási egységnél külön megtenni

Minden fordítási egységben csak 1 precompiled headert lehet felhasználni, és ennek kell lennie az első include-nak

- De ebben lehetnek további include-ok, így oda lehet gyűjteni a sok helyen használt, de ritkán változó header fájlokat

Precompiled header használata

GCC-ben:

- Ha a fordítónak egy header fájlt adunk át (.h, .hh vagy .hpp), akkor egy .gch kiterjesztésű precompiled headert készít belőle
- Minden fordítási egység fordításakor az első include direktívánál megnézi, hogy van-e az include-olt headerből precompiled változat, és ha van, akkor azt használja a sima header helyett

Visual Studio-ban:

- Project Properties > C/C++ > Precompiled Headers
- Megköveteli, hogy minden forrásfájl elején be legyen include-olva
  (de ez fájlonként kikapcsolható a File Properties alatt)
- Az előfeldolgozandó header fájl alapértelmezett neve pch.h vagy stdafx.h, a generált precompiled header pedig azonos nevű .pch fájl

Privát tagok

A header tartalmazza az osztály publikus interfészét

Viszont a privát tagfüggvényeit és adattagjait is

- Ezek is include-olásra kerülnek, feleslegesen
- További include-okra lehet szükség miattuk a headerben
- Ha csak a privát tagok változnak, akkor is újra kell fordítani a headert használó forrásokat

Hogy lehetne megoldani, hogy csak az implementációs forrásfájlban szerepeljenek a privát tagok?

PImpl idióma

A megoldás, hogy a privát tagokat becsomagoljuk egy struktúrába, amit csak a forrásfájlban definiálunk

- Az egyetlen privát tag egy objektum lesz ebből a struktúrából

Viszont a headerben nem lehet definiálatlan struktúra

- Az osztály méretének ismertnek kell lennie

Mutatót viszont tárolhatunk definiálatlan típusra (elég deklarálni)

PImpl példa header

```
#pragma once

class ShyClass {
private:
  struct Private;  // forward-declaration
  Private* p;
public:
  ShyClass();
  ~ShyClass();
  int getSum() const;
  // more public methods ...
};
```

PImpl példa .cpp

```cpp
#include "ShyClass.h"
#include <vector>  // only used for private member

struct ShyClass::Private {
  std::vector<int> myNumbers;
  Private() : myNumbers({1, 2})  // initialize private data
  {}
  void addNumber(int n) {
    myNumbers.push_back(n);
  }
  int privGetSum() {
    int sum = 0;
    for (int n : myNumbers) sum += n;
    return sum;
  }
};
```

```cpp
ShyClass::ShyClass()
  : p(new Private())
{
}

ShyClass::~ShyClass() {
  delete p;
}

int ShyClass::getSum() const {
  int n = p->privGetSum();  // call private function
  p->addNumber(n);  // modifies state in const method!
  return n;
}
```

PImpl példa unique_ptr-rel

HEADER FILE

```
#pragma once

#include <memory>

class ShyClass {
private:
  struct Private;  // forward-declaration
  std::unique_ptr<Private> p;
public:
  ShyClass();
  ~ShyClass();
  int getSum() const;
  // more public methods ...
};
```

SOURCE FILE

```
// struct ShyClass::Private { remains the same ... };

ShyClass::ShyClass()
  : p(std::make_unique<Private>())
{}

ShyClass::~ShyClass() {
  // p is deleted automatically
}

int ShyClass::getSum() const {
  int n = p->privGetSum();  // call private function
  p->addNumber(n);  // modifies state in const method!
  return n;
}
```

std::experimental::propagate_const

Hiába vagyunk const metódusban, a mutatón keresztül módosítható a Private struktúra

A megoldás a propagate_const wrapper

- A benne lévő pointer const kontextusban úgy viselkedik, mintha const-ra mutató pointer lenne
- Fordítási hibát kapunk, ha const függvényből a mutatott objektum non-const metódusát hívjuk

```
std::experimental::propagate_const<std::unique_ptr<Private>> p;
```

Beterjesztésre került, de még nem került be a szabványba

- MSVC nem is támogatja
- GCC viszont igen

Publikus tagok elérése a privátból

Gyakran a privát függvényekből is el kell érni a publikus tagokat, nem csak fordítva

Az előző példában erre nincs mód

El kell tárolni a külső osztály címét a privát struktúrában, azon keresztül hozzá lehet férni a külső osztály publikus tagjaihoz

```
struct ShyClass::Private {
  ShyClass* self;
  std::vector<int> myNumbers;
  Private(ShyClass* parent)
    : self(parent)
    , myNumbers({1, 2})
  {}
  void addNumber(int n);
  int privGetSum();
};
```

Jövő: C++20 modules

A C++20 szabvány vezette be, a legújabb fordítok már támogatják, de az IDE támogatottság és a használatának elterjedése még a jövő zenéje

A modulok önálló fordítási egységek (szemben a header fájlokkal, amik csak bemásolódnak más fordítási egységekbe), így csak egyszer kell őket lefordítani

- Nincs szükség precompiled header fájlra
- Míg pch-ból csak egyet lehet include-olni, modulból többet is, így nem kell egy nagy csomagba tenni őket

A modulokból az export kulcsszóval lehet elérhetővé tenni a deklarációkat

Az import kulcsszóval lehet betölteni egy modul deklarációit

Modulok Visual Studioban

Project Properties > C/C++ > Language > C++ Language Standard > Preview (/std:c++latest)

Project Properties > C/C++ > Language > Enable Experimental C++ Standard Library Modules

A standard library is elérhető modulokként, ehhez a Visual Studio telepítőjében ki kell választani a C++ Modules csomagot

- **std.regex** provides the content of header <regex>
- **std.filesystem** provides the content of header <filesystem>
- **std.memory** provides the content of header <memory>
- **std.threading** provides the contents of headers <atomic>, <condition_variable>, <future>, <mutex>, <shared_mutex>, and <thread>
- **std.core** provides everything else in the C++ Standard Library
- Az IntelliSense jelenleg nem tudja kezelni a standard library modulokat, így a kódkiegészítés és syntax highlighting nem működik

Példa

MyModule.ixx

```
export module MyModule;

import std.core;

void PrivateFunc() {
  std::cout << "PrivateFunc() called\n";
}

export void MyFunc() {
  std::cout << "MyFunc() called\n";
  PrivateFunc();
}
```

main.cpp

```
import std.core;
import MyModule;


int main() {
  std::string mod = "Hello modules!\n";
  std::cout << mod;
  MyFunc();
}
```

Modul több fájlban implementálva

Ugyan a moduloknál a fordítást nem befolyásolja, de az áttekinthetőség érdekében érdemes külön fájlba tenni a publikus interfész deklarációit és az implementációt

Az exportálandó deklarációkat tartalmazó fájl a **modul interfész fájl**

- Ebből pontosan 1 van modulonként
- Az `export module MyModule;` deklarációval kezdődik
  - Ha muszáj, előtte lehet include, de ha lehet, inkább modulokként importáljuk a szükséges deklarációkat

A definíciókat és privát deklarációkat tartalmazó fájl a **modul implementációs fájl**

- Ebből 0, 1, vagy több is lehet egy modulhoz
- A `module MyModule;` deklarációval kezdődik