Bevezetés a programozásba 2

5. Előadás: Fordítási egység

ISMÉTLÉS

Tagfüggvény kiemelése

```
struct Particle {
    int x,y;
    unsigned char r,g,b;
    void rajzol();
};

void Particle::rajzol() {
    gout << move_to(x, y)
    << color (r, g, b)
    << dot;
}
```

ISMÉTLÉS

Interface - Implementation

```
struct Particle {
    int x,y;
    unsigned char r,g,b;
    void rajzol();
};
```

```
void Particle::rajzol() {
    gout << move_to(x, y)
    << color (r, g, b)
    << dot;
}
```

ISMÉTLÉS

Láthatóság

- Miért jó elrejteni a mezőket?
  - Mert akkor a program többi része nem fér hozzá, csak a tagfüggvényeken keresztül
  - Ez biztonságos, és karbantartható kódot fog adni
- Láthatósági szintek:
  - public: mindenki látja
  - protected: örökösök látják csak
  - private: senki más nem látja
- C++-ban a láthatóság típusszintű

ISMÉTLÉS

Láthatóság

```
struct Particle {
    Particle(int X, int Y);
    virtual void mozog( … );
    virtual void rajzol( … );
protected:
    double x,y;
    unsigned char r,g,b;
};
```

ISMÉTLÉS

Osztály

```
class Particle {
public:
    Particle(int X, int Y);
    virtual void mozog( … );
    virtual void rajzol( … );
protected:
    double x,y;
    unsigned char r,g,b;
};
```

Implementáció elrejtése

- Külön fájlba tesszük a felületet és a megvalósítást
- Az interface helye a fejlécfájl (header, .h, .hpp) fájl lesz
- Az implementáció helye egy új .cpp fájl
- Így előre lefordítható lesz a típusunk (.o fájl), ez a tárgykód
- A program (“project”) pedig több tárgykódból készül el

Implementáció elrejtése

particle.hpp

```
#ifndef PARTICLE_HPP
#define PARTICLE_HPP

struct Particle {
    int x,y;
    unsigned char
r,g,b;
    void rajzol();
};

#endif
```

particle.cpp

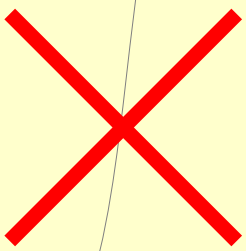
```
#include "particle.hpp"

void Particle::rajzol() {
    gout << move_to(x, y)
    << color (r, g, b)
    << dot;
}
```

main.cpp

```
#include "particle.hpp"

int main() {
 …
    Particle p;
    p.rajzol();
}
```



A projekt fogalma

- Forráskód fájlok (.cpp fájlok)
  - A .h/.hpp fájlokra a .cpp fájlokból úgyis hivatkozunk
- Fordítási beállítások
  - lib hozzáadása, pl. SDL
  - Warning level
  - Optimalizáció
- Target (grafikus/konzol, release/debug, application/lib)
- „Makefile”, „Solution”

Példa projekt (Code::Blocks)

```
 …
<Option title="feladat" />
 …
<Add option="-O2" />
 …
   <Linker>
      <Add option="-s" />
      <Add library="graphics" />
   </Linker>
 …
<Add library="SDLmain" />
 …
<Unit filename="main.cpp" />
```

A fordítás menete

- Preprocesszor
- Forráskódok értelmezése, típusok feltérképezése (pl. méret miatt).
- Forrásfájlonként egy tárgykód (object, .o) fájl létrehozása. Ebben gépi kódú részletek vannak, előkészítve a csatlakozási pontokat
- A linker összeköti a tárgykódokat
  - “undefined reference”: egyik tárgykódban sincs feloldva egy csatlakozási pont, pl függvényhíváshoz nem található függvény implementáció

Preprocesszor direktívák

- #ifdef / #ifndef AZONOSITO, #endif
  - A fordításba a következő részletet akkor tegye/ne tegye bele, ha definiálva van az AZONOSITO
- #define AZONOSITO [ÉRTÉK]
  - Egy azonosító definiálása, lehet az értéke konstans vagy képlet, fordítási időben behelyettesítődik
- #include <X> vagy “X”
  - Az adott fájl tartalmát ide illessze be. A <> azt jelenti, hogy szabványos elérési úton keresse, a “” pedig hogy az adott .cpp könyvtárában.

Fejlécfájl kerete

particle.hpp

```
#ifndef PARTICLE_HPP
#define PARTICLE_HPP

struct Particle {
    int x,y;
    unsigned char
r,g,b;
    void rajzol();
};

#endif
```

Csak egyszer szerkeszthető a kódba: a következő #include alkalmával ez már definiálva van

Ökölszabályok

- .cpp fájlt nem inkludálunk
- Egy fájlba egy téma kerül, sokszor ez egyetlen típus
- Nincs körkörös hivatkozás
  - Pl.: struct A {B b;}; struct B {A a;}; nincs, de mutatót és referenciát szabad tenni (mert ismert a mérete)
  - Az ebből felmerülő problémák feloldhatóak előre deklarált típusokkal. (pl. #include “particle.hpp” helyett csak annyi, hogy struct Particle;) Majd a linker szól, ha nincs meg az implementáció. Ilyenkor csak mutató és referencia használható

Fordítási egység

- Fordítási egység az, ami önállóan fordítható
- A C++ nyelvben minden .cpp fájl önálló fordítási egység
- Moduláris programozás
  - Felgyorsítja a fordítást: csak a megváltozott forráskódokhoz tartozó egységek fordulnak újra
  - A személyi felelősség értelmezhető a rendszer alkotóelemeire: önállóan tesztelhető modulok
  - Strukturális / moduláris programozás

Linker

- A fordítási egységekből közös binárist állít elő
- Néhány hibalehetőség:
  - Nincs meg egy függvény, vagy egy változó: undefined reference
  - Ugyanaz a változó a fejlécfájlban deklarálva mindegyik azt beszerkesztő fordítási egységben globális, egymással ütköznek linkeléskor: multiple definition
    - extern : valahol majd deklarálva lesz, a többiek tudjanak róla, hogy van. Pontosan egy helyen deklarálod is.
    - **`extern groutput& gout;`**
    - esetleg static, ha mindenhol külön-külön akarod

A könyvtár

- Egy vagy több fordítási egység, amik jól körülhatárolható célra együtt használhatóak
- Előre lefordítható (pl. libgraphics.a)
- Nehezen túlbecsülhető szerepe van egy programnyelv tekintetében, hogy képes-e könyvtárakat kezelni
  - Pl. PLanG nem, ezért mindent neked kell csinálni
- Újrafelhasználható eredmények
  - http://gnuwin32.sourceforge.net/
  - http://devpaks.org/

Könyvtárak

- Példák: SDL, STL, stb..
- Nagyon sok van, a legtöbb feladatot valaki más már megcsinálta, neked elég felhasználni
  - Természetesen meg kell tanulni használni
- A saját widgetkészlet jó példa könyvtárra:
  - Jól körülírható célra készült
  - Újrafelhasználható
  - Mások is használhatják, ha van dokumentáció (lesz)
  - Könyvtáron belül következetes stílus

Könyvtár létrehozása

- A project ne tartalmazzon main() függvényt
  - IDE: A target legyen “library”
  - parancssor: g++ -c forras1.cpp forras2.cpp …
- Fordítás után indítható bináris helyett libX.a fájl jön létre (.lib is lehet más fordítóknál)
- A libX.a és az X.hpp együtt használható, csak a projectet kell beállítani
- Második beadandóban a maximális pontszámért így kell majd fordítani

Könyvtár fordítása : előtte

particle.hpp

```
#ifndef PARTICLE_HPP
#define PARTICLE_HPP

struct Particle {
    int x,y;
    unsigned char
r,g,b;
    void rajzol();
};

#endif
```

particle.cpp

```
#include "particle.hpp"

void Particle::rajzol() {
    gout << move_to(x, y)
    << color (r, g, b)
    << dot;
}
```

main.cpp

```
#include "particle.hpp"

int main() {
 …
   Particle p;
   p.rajzol();
}
```

Könyvtár fordítása : utána

particle.hpp

```
#ifndef PARTICLE_HPP
#define PARTICLE_HPP

struct Particle {
    int x,y;
    unsigned char
r,g,b;
    void rajzol();
};

#endif
```

libparticle.a

gépi kódú tartalom

main.cpp

```
#include "particle.hpp"

int main() {
 …
   Particle p;
   p.rajzol();
}
```

Könyvtár tartozékai

- Fejlécfájl
- Előrefordított kód és/vagy forráskód
- Dokumentáció
  - Lehetőleg több, mint kommentek a fejlécfájlban
  - html, pdf, ...
- Esetleg dinamikus könyvtár fájlok (.dll)
- Esetleg példaprogramok és tutorial
- Esetleg néhány IDE számára projectfájl

Összefoglalás

- Az implementáció elrejtése a konzisztencia megőrzésének legbiztosabb eszköze
- Az implementáció külön fájlba, külön fordítási egységbe helyezhető
- A fordítási egységek önállóan fejleszthetőek, tesztelhetőek
- Fordítási egységekből, azok halmazaiból könyvtár készíthető
- A moduláris programozás segít a többfős csapatok munkájában