C++

10. HÉT – PROGRAMKÖNYVTÁRAK (LIBRARIES)

A JEGYZETET KÉSZÍTETTE:

CSAPÓ ÁDÁM BALÁZS ÉS ŐSZ OLIVÉR

Mire jó egy C++ library?

Ha általánosan használható függvényeket, osztályokat írunk, akkor felmerül az igény, hogy ezeket több programhoz is felhasználhassuk (vagy mások).

Kód újra-felhasználási módszerek:

- Felhasznált források belefordítása a projektbe
  - Speciális eset: header-only library (pl. a Boost nagy része)
- Lefordított források linkelése a projektbe (ez a statikus library)
- Dinamikus linkelés futtatáskor (shared library és DLL)
  - A library nem kerül bele a futtatható binárisba, de jelen kell lennie futáskor

Statikus library használata

1. A szokásos módon elvégezzük a fordítást (compile) a library kódra
2. A kapott object fájlokat becsomagoljuk egy archívumba
   - Unix alapú rendszeren: `ar rcs libsomething.a object1.o object2.o object3.o`
   - Windows-on: `lib /out:libsomething.lib object1.obj object2.obj object3.obj` vagy Visual Studio-ban állítsuk a projekt kimenetét Library-re
3. A library használatához fordításkor szükség van a deklarációkat tartalmazó header-ökre
4. Linkelésnél megadjuk a felhasználandó library-t
   - Unix alapú rendszeren: `g++ main.cpp -Lpath/to/mylib -lsomething`
     - A /lib és /usr/lib az alapértelmezett keresési hely
     - `main.cpp` helyett lehet `main.o` is, de mindenképp meg kell előznie a library argumentumot
   - Windows-on: `cl something.lib main.obj/main.cpp` vagy Visual Studióban:
     Project Properties > Linker > General > Additional Library Directories
     Project Properties > Linker > Input > Additional Dependencies

Shared library

A leghasznosabb library-ket sok program felhasználja. Viszont pazarlás mindbe belemásolni ugyanazt a kódot.

A shared library dinamikusan, futtatáskor kerül linkelésre/betöltésre (loading)

- A shared library egy példányban kerül a memóriába, amit több program is használhat
- A futtatható bináris kisebb lesz, de a futtatáshoz mellékelni kell a shared library-t
- A shared library betöltése, linkelése extra munkát jelent programindításkor
- Ha módosul a library implementáció (de az interfész nem), nem kell újrabuildelni a programot
- A standard library is ilyen shared library
- Nem csak a program indításakor, hanem futás közben is tölthetők be újabb modulok, így ha valamire nincs szükség, az nem kerül feleslegesen betöltésre

Shared object használata Linuxon

1. Fordításkor bekapcsoljuk a PIC (Position-Independent Code) flaget:
   `g++ -c -fpic code.cpp`
2. A kapott object fájlokból shared objectet készítünk:
   `g++ -shared -o libsomething.so code.o`
3. Ugyanúgy linkeljük a programunkhoz, mint a statikus library-t:
   `g++ main.cpp -Lpath/to/mylib -lsomething`
4. Hogy a loader megtalálja, a `/usr/lib` alá tesszük a library-t, vagy környezeti változóban megadjuk az elérési útvonalat futtatás előtt:
   `export LD_LIBRARY_PATH=/path/to/lib`

DLL (Dynamic-link library) használata Windows-on

1. Visual Studióban DLL projektet hozunk létre
2. A publikus interfészt deklaráló header fájlokban exportáljuk az osztályokat és függvényeket:

```
#ifdef MYLIBRARY_EXPORTS                        // DLL projektben definiálva van a PROJECT_EXPORTS
#define MYLIBRARY_API __declspec(dllexport)     // a DLL buildje ezt használja
#else
#define MYLIBRARY_API __declspec(dllimport)     // a DLL-t felhasználó projekt meg ezt
#endif

MYLIBRARY_API void function();
class MYLIBRARY_API MyClass {...};
```

3. A projekt buildelésével kapunk egy .dll és egy .lib fájlt
4. A .lib nem tartalmazza az implementációt, csak egy import lib, amit linkelnünk kell a használat helyén
5. A DLL-t az .exe fájl mellé rakjuk, hogy futtatáskor megtalálja (vagy telepítjük a rendszer DLL-ek közé)

Dinamikus library betöltés C programból Linuxon

A `<dlfcn.h>` függvényeinek segítségével:

- `void* dlopen(const char *filename, int flag)`
  - Betölti a megadott library-t a memóriába, és visszaadja a címét (egy handle-t), hiba esetén NULL-t
  - Ha már be van töltve valahova, akkor ugyanazt a címet adja vissza
  - A flag-ek segítségével lehet szabályozni a betöltés módját
- `void* dlsym(void *handle, char *symbol)`
  - Visszaad egy függvény pointert a library adott nevű (symbol) függvényére
- `int dlclose()`
  - Számon tartja, hány helyről próbálták betölteni, és csak akkor szabadítja fel, ha annyiszor hívták meg, ahányszor a dlopen()-t
- `char* dlerror()`
  - Visszaadja a legutóbbi hibás dl... hívás hibaüzenetét C stringként

Dinamikus library betöltés C++ programból Linuxon

C-ben a függvény neve egyértelműen azonosítja azt (symbol = function name), de C++-ban nem (function name -> name mangling -> symbol)

- Function overloading
- Template-ből generált függvények
- Osztályok tagfüggvényei

Megoldás: `extern "C"` kulcsszóval kell deklarálni a függvényeket

- C függvényt deklarál C++-ban (egyedi függvénynév szükséges)

A betöltött osztályok példányosítására egy függvény pointer nem alkalmas

- Hogy kérjük le a konstruktor pointerét? Hogy hívjuk meg? Visszatérési értéke nincs, a new meg nem működik, ha nincs definiálva az osztály.
- Megoldás: polimorfizmus (absztrakt ősosztály virtuális interfésszel), factory függvények (létrehozásra és megsemmisítésre is)

Dinamikus library betöltés C++ programból Windows-on

A működési elv Windows-on is ugyanaz, mint Linuxon, csak a szintaktika más

- `dlfcn.h` helyett `windows.h`
- `dlopen(filename, flags)` helyett `LoadLibrary(filename)`
  - vagy `LoadLibraryExA(filename, NULL, flags)`
- `dlsym(handle, symbol)` helyett `GetProcAddress(handle, symbol)`
- `dlclose(handle)` helyett `FreeLibrary(handle)`

Ha dinamikusan kerül betöltésre, akkor nem kell linkelni az import library-t

Library verziókövetés

Library fejlesztése során különösen figyelni kell rá, hogy az interfész ritkán változzon

- A visszafelé kompatibilitás gyakran nem kivitelezhető, vagy nem szempont
- A library felhasználóinak kell követniük a library fejlődését, változásait

Windows-ban korábban sok problémát okoztak az inkompatibilis DLL-ek (DLL Hell)

- Nem a megfelelő verzió került betöltésre
- A programok felülírták egymás DLL-jeit

Megoldás: a verziószám kerüljön bele a fájlnévbe

Dependency kezelés

Ha sok library-vel dolgozunk, azok beszerzése, konfigurálása és verziókövetése nehézkessé válhat

Sok más nyelvhez van standard csomagkezelő, ami ezt automatizálja (pl. Node Package Manager)

C++ projektekben sajátos nehézségek:

- Platformfüggő kódrészletek
- Külön debug és release build kell library-ből is
- Többféle fordító, eltérő verziók, beállítások
- Különböző build toolchainek
- Nincs egy standard repository

A build automatizáló eszközök csak részben nyújtanak megoldást:

- Egymással nem tudnak együttműködni
- A library-k beszerzését, frissítését, és a build konfigurálását manuálisan kell megoldani

Ismertebb C++ package managerek

Több próbálkozás is indult a probléma megoldására, de még nincs győztes

- CPM
  - CMake alapú
  - Fejlesztése leállt 2015-ben
- vcpkg
  - A Microsoft saját fejlesztésű, open source megoldása, nem csak Windows-ra
  - Jelenleg ez a legelterjedtebb, de nem tekinthető sztenderdnek
- Apache Maven és Gradle
  - Java központú, de egy ideje már C++ is támogatott
- Conan
  - Decentralizált, open source, bármilyen build toollal kompatibilis
  - Pythonban fejlesztik 2015. óta, még nem annyira elterjedt, de már elég sok package elérhető