FEJLETT PROGRAMOZÁSI NYELVEK, 2009
2. GYAKORLAT - Linux alatti C/C++ programozás

Cél:

- Dinamikus csatolású függvénykönyvtár készítése és használata
- Plugin-szerű betöltés
- Egyszeű C++ osztályok készítése

I. Dinamikus csatolású függvénykönyvtárak

1. Jellemzők

Shared library, Dynamically Linked Library, Shared Object

- A dinamikus csatolású függvénykönyvtár a statikushoz hasonlóan tárgykód állományok csoportjából képződik.
- Amíg a statikus csatolású függvénykönyvtár használata esetén a fordító beleszerkeszti a végrehajtható állományba azon függvények kódját, amelyeket ténylegesen meghív a program, addig a dinamikus csatolású könyvtárak esetén a végrehajtható állomány csak az osztott függvénykönyvtárra való hivatkozást fog tartalmazni.
- Ha több végrehajtható állomány van, amely össze van kapcsolva az osztott könyvtárral, akkor mindenikük csak hivatkozást fog tartalmazni erre, magát a függvénykönyvtárat egyikük sem fogja tartalmazni. Ez nagyon előnyös a végrehajtható állományok helyigénye szempontjából.
- Amíg a statikus csatolású függvénykönyvtár több tárgykód állomány archívuma, addig az osztott függvénykönyvtár egyetlen tárgykód állomány. Így a dinamikus csatolású függvénykönyvtár használatkor az egész csatolódik a végrehajthatóhoz. A statikus csatolás esetében csak azokat a tárgykód állományokat csatolja a fordító (szerkesztőlinker), amelyekre tényleges hivatkozás történt.

2. Készítés

1. lépés

Azon forrásállományokat, amelyekből készíteni szeretnénk a függvénykönyvtárat lefordítjuk (csak compile) az fPIC kapcsoló segítségével.

```
gcc -c -fPIC test1.c test2.c test3.c
```

PIC Position-Independent Code

Az osztott könyvtárbeli függvények különböző memória címekre fognak betöltődni. Ezért a kódjuk nem függhet a betöltési címtől. Erre kell emlékeztetni a fordítót az fPIC kapcsoló segítségével.

2. lépés

Összeszerkesztjük a megfelelően lefordított tárgykód állományokat.

```
gcc -shared -fPIC -o libtest.so test1.o test2.o test3.o
```

3. Használat

Osztott függvénykönyvtárak használata a statikusokéhoz hasonló. Ugyanazokat a kapcsolókat használjuk az útvonal (-L), illetve a könyvtárnév megadására (-l).

```
gcc -o main main.o -L . -ltest
```

Ha ugyanazon függvényekből készítünk statikus és dinamikus csatolásút is, akkor a fordító azt fogja csatolni amelyiket hamarabb megtalálja, az útvonal beállításoknak megfelelően. Ha viszont ugyanabban a könyvtárban van a statikus illetve a dinamikus csatolású is, akkor a fordító a dinamikus csatolásút részesíti előnyben. A statikus csatolású iránti igényünket a `static` kapcsolóval nyilváníthatjuk ki.

```
gcc -static -o main main.o -L . -ltest
```

Az előző parancs alapján elkészül a `main` nevű végrehajtható állomány. Ez az állomány a `main.o` tárgykódfájlból készül a `libtest.a` függvénykönyvtár használatával. A `libtest.a` függvénykönyvtárat az aktuális munkakatalógusban keresi a fordító a `-L .` kapcsoló hatására. Ha a könyvtár fordításával minden rendben volt, a végrehajtható állományunkat is sikeresen elkészítettük, még mindig lehet egy probléma. Indításkor kaphatjuk a következő üzenetet:

Cannot open shared library...

Ez azt jelenti, hogy futásidőben az operációs rendszer csak a szabványos katalógusokban keresi az osztott könyvtárat. Erre a problémára a barbár megoldás az lenne, hogy bemásoljuk a könyvtárat a szabványos katalógusba (ha van írási jogosultságunk). Az elegáns megoldás pedig az, hogy kibővítjük a függvénykönyvtárak keresési útvonalát. Ez a `-Wl,-rpath <utvonal>` kapcsolóval adható meg.

```
gcc -o main main.o -L . -ltest -Wl,-rpath .
```

Feladat:

Az előző órán készített stack modulból készítsen dinamikusan csatolható függvénykönyvtárat és próbálja ki a múlt órán elkészített példaprogramra.

II. Dinamikus betöltés és felszabadítás (Loading and Unloading)

FAKULTATÍV FELADAT

Bizonyos helyzetekben futás közben szeretnénk kódot betölteni, csatolás nélkül. A betöltött kódot pedig használat után el szeretnénk távolítani. Ez a plugin modulokhoz hasonlóan működik. Linux alatt a `dlopen, dlsym és dlclose` függvények segítségével oldhatjuk meg a problémát. Például a `libtest.so` függvénykönyvtárat a következő függvénnyel tölthetjük be:

```
dlopen("libtest.so", RTLD_LAZY);
```

Az RTLD_LAZY a csatolt függvénykönyvtár szimbólumaira vonatkozik és lusta betöltést jelent. Betöltés után a betöltött függvények címét a `dlsym` függvénnyel kaphatjuk meg.

main.c

```
#include <dlfcn.h>
...
void * handle = dlopen( "libtest.so",RTLD_LAZY);
void (*fptr)() = dlsym(handle, "my_function");
(*fptr)();
dlclose(handle);
...
```

FONTOS!

- ne feledjük a programot a `-ldl` kapcsolóval fordítani a `libdl` függvénykönyvtár csatolása érdekében.
- fordításkor használjuk a -Wl,-rpath <katalógus> kapcsolót azért, hogy futásidőben is megtalálja a függvénykönyvtárat

```
gcc -o main main.c -ldl -Wl,-rpath .
```

III. Egyszerű C++ osztályok készítése

Használja a Netbeans IDE-t a feladat elkészítésére!

1. Adott egy képernyőpontot ábrázoló `Pont` osztály. Kiemelve vannak azok a részek, amelyeket másképpen végzünk, mint Javaban:

- a példány adattagjaira: `this` mutató
  `this->x`
- a láthatóságokat blokkokra adjuk meg:
  `public:`
  alapértelmezett a `private` (pl. int x, y)
- ha egy metódus nem változtatja meg az objektum állapotát- `const` módosító

Pont.h

```
#ifndef _PONT_H
#define _PONT_H

class Pont{
    int x, y;
public:
    Pont( int x=0, int y=0){
        if( x>=0 && x<=2000 ) this->x = x;
        else
            this->x = 0;
        if( y>=0 && y<=2000 ) this->y = y;
        else
```

```
            this->y = 0;
    }

    int getX() const{
        return x;
    }

    int getY() const{
        return y;
    }

    void move( int nx, int ny){
        if( nx>=0 && nx<=2000 && y>=0 && y<=2000){
            this->x = nx;
            this->y = ny;
        }
    }
};

#endif     /* _PONT_H */
```

Tanulmányozza a `Pont` osztály alábbi használatát és válaszoljon a következő kérdésekre:

- Mi a különbség a `p1,p2,` illetve a `pp1, pp2` között?
- Mi a szerepe a `delete` operátornak?
- Miért nem használható a delete a `p1, p2` esetében?
- Mi a `cout`?

```
#include <cstdlib>
#include <iostream>
#include "Pont.h"
using namespace std;

int main(int argc, char** argv) {
    Pont p1(2,3);
    cout<<"p1( "<<p1.getX()<<","<<p1.getY()<<")"<<endl;
    Pont p2(100, 200);
    cout<<"p2( "<<p2.getX()<<","<<p2.getY()<<")"<<endl;


    Pont * pp1 = new Pont(300, 400);
    Pont * pp2 = new Pont(500, 1000);

    cout<<"pp1( "<<pp1->getX()<<","<<pp1->getY()<<")"<<endl;
    cout<<"pp2( "<<pp2->getX()<<","<<pp2->getY()<<")"<<endl;

    delete pp1;
    delete pp2;

    return (EXIT_SUCCESS);
}
```

2. Adott a következő C++ program. Futtassa és értelmezze a programot!

```
#include <iostream>
#include <iterator>
#include <algorithm>
#include <string>
#include <vector>

using namespace std;

int main(int argc, char** argv) {
    vector<string> v;
    cout<<"Irjon be karakterlancokat ^D vegjelig: "<<endl;
    copy(istream_iterator<string>(cin),istream_iterator<string>(),
         back_inserter(v));
    sort(v.begin(), v.end());
    cout<<"A kimenet:"<<endl;
    copy( v.begin(), v.end(), ostream_iterator<string>(cout,"\n"));

    return 0;
}
```