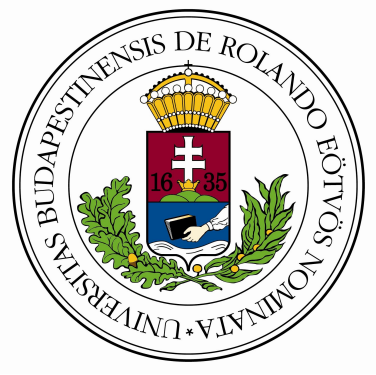
Imperatív programozás

Statikus programszerkezet



Kozsik Tamás és mások

ELTE Eötvös Loránd Tudományegyetem

Statikus programszerkezet

- kifejezés
- utasítás
- alprogram
- modul

Modul

- Nagyobb egység
- Nagy belső kohézió
- Szűk interfész
  - Gyenge kapcsolat modulok között
  - Jellemzően egyirányú

Fordítás

- forráskód (source code) forrásfájlban (source file)
  - `factorial.c`
- fordítóprogram (compiler)
  - `gcc -c factorial.c`
- tárgykód (target code, object code)
  - `factorial.o`

Fordítási egység

(compilation unit)

- a forráskód egy része (pl. egy modul)
- egyben odaadjuk a fordítónak
- tárgykód keletkezik belőle

Egy program több fordítási egységből szokott állni!

C-ben

Egy forrásfájl tartalma

Szerkesztés, végrehajtható kód

- tárgykódok (target code, object code)
  - `factorial.o` stb.
- szerkesztőprogram (linker)
  - `gcc -o factorial factorial.o`
- végrehajtható kód (executable)
  - `factorial`
  - alapértelmezett név: `a.out`

Sok tárgykódból lesz egy végrehajtható kód!

Végrehajtás

```
./factorial
```

Több fordítási egység

factorial.c

```
int factorial( int n )
{
    int result = 1, i;
    for(i=2; i<=n; ++i)
    {
        result *= i;
    }
    return result;
}
```

tenfactorial.c

```
#include <stdio.h>

int factorial( int n );

int main()
{
    printf("%d\n", factorial(10));
    return 0;
}
```

Fordítás, szerkesztés, futtatás

```
gcc -c factorial.c tenfactorial.c
gcc -o factorial factorial.o tenfactorial.o
./factorial
```

A két lépés összevonható egy parancsba

- forráskód forrásfájl(ok)ban
  - `factorial.c` és `tenfactorial.c`
- fordítóprogram és szerkesztőprogram végrehajtása
  - `gcc -o factorial factorial.c tenfactorial.c`
- végrehajtható kód (executable)
  - `factorial`

Fordítási hibák

- Nyelv szabályainak megsértése
- Fordítóprogram detektálja

factorial.c

```
int factorial( int n )
{
    int result = 1;
    for(i=2; i<=n; ++i)
    {
        result *= i;
    }
    return result;
}
```

gcc -c factorial.c

```
factorial.c: In function ‘factorial’:
factorial.c:6:9: error: i undeclared (first use in this function)
     for(i=2; i<=n; ++i)
         ^
```

Szerkesztési hibák

factorial.c

```c
int factorial( int n )
{
    int result = 1, i;
    for(i=2; i<=n; ++i)
    {
        result *= i;
    }
    return result;
}
```

tenfactorial.c

```c
#include <stdio.h>

int faktorial( int n );

int main()
{
    printf("%d\n", faktorial(10));
    return 0;
}
```

Fordítás, szerkesztés, hiba

```
$ gcc -c factorial.c tenfactorial.c
$ gcc -o factorial factorial.o tenfactorial.o
tenfactorial.o: In function `main':
tenfactorial.c:(.text+0xa): undefined reference to `faktorial'
collect2: error: ld returned 1 exit status
```

Fordítási és futási idejű szerkesztés

Statikus szerkesztés

- Még a program futtatása előtt
- A tárgykódok előállítása után “egyből”
- Előnye: szerkesztési hibák fordítási időben

Dinamikus szerkesztés

- A program futtatásakor
- Dinamikusan szerkeszthető tárgykód
  - Linux *shared object*: `.so`
  - Windows *dynamic-link library*: `.dll`
- Előnyei
  - kisebb végrehajtható állomány
  - kevesebb memóriafogyasztás

Programok C-ben

Fordítási idő

- Forrásfájlok (.c és .h)
- Előfeldolgozás
- Fordítási egységek
- Fordítás
- Tárgykódok
- Statikus szerkesztés
- Futtatható állomány

Futási idő

- Futtatható állomány, tárgykódok
- Dinamikus szerkesztés
- Futó program

Program Pythonban

Egy csomó utasítás egymás után

factorial.py

```
def factorial(n):
    result = 1
    for i in range(2,n+1):
        result *= i
    return result

print(factorial(10))
```

Végrehajtás

```
python3 factorial.py
```

Végrehajtható shellscript

factorial.py

```
#!/usr/bin/python3

def factorial(n):
    result = 1
    for i in range(2,n+1):
        result *= i
    return result

print(factorial(10))
```

Végrehajtás

```
./factorial.py
```

Több modulból álló Python program

```
python3 main.py
```

signaling.py

```
counter = 0

def signal():
    global counter
    counter += 1
```

main.py

```
import signaling

signaling.signal()
signaling.signal()
print(signaling.counter);
```

Minősítés nélküli függvénynév

```
python3 main.py
```

signaling.py

```
counter = 0

def signal():
    global counter
    counter += 1
```

main.py

```
import signaling
from signaling import signal

signal()
print(signaling.counter);
```

Trükk: főprogram is, könyvtár is

factorial.py

```
def factorial(n):
    result = 1
    for i in range(2,n+1):
        result *= i
    return result

if __name__ == "__main__":
    print(factorial(10))
```

Végrehajtás

```
python3 factorial.py
```

Modulok Pythonban

- Fájlokba szervezett kód
  - Definíciók (lásd: `dir(modulename)`)
  - Utasítások (inicializáció)
- `import` és `from…import`
- Modul vagy főprogram

```
if __name__ == "__main__":
```

- Keresési útvonal: `sys.path` és `PYTHONPATH`
- Fordítás: `.pyc` fájlok a `__pycache__` könyvtárban
- Csomagok: hierarchikus névtér

Modulok C-ben

- Fordítási egységek
- Forráskód: `.c` és `.h`
- `#include`
- Szerkesztés: statikus vagy dinamikus

C - statikus globális deklarációk

- Más fordítási egységben nem érjük el
- “Belső szerkesztésű” (internal linkage)
- Az implementációhoz tartozik
- Nem része a modul interfészének
- Információ elrejtés elve

```
int positive = 1;
static int negative = -1;
extern int increment;

static void compensate(void){
    negative -= increment;
}

void signal(void){
    positive += increment;
    compensate();
}
```

Több fordítási egységből álló C program

```
gcc -c -W -Wall -pedantic -ansi main.c
gcc -c -W -Wall -pedantic -ansi positive.c
gcc -o main -W -Wall -pedantic -ansi positive.o main.o
```

positive.c

```c
int positive = 1;
static int negative = -1;
extern int increment;

static void compensate(void){
    negative -= increment;
}
void signal(void){
    positive += increment;
    compensate();
}
```

main.c

```c
#include <stdio.h>

int increment = 3;
extern int positive;
extern void signal(void);

int main(){
    signal();
    printf("%d\n", positive);
    return 0;
}
```

Fejállományok

positive.h

```
extern int positive;
extern void signal(void);      /* itt az extern elhagyható */
```

positive.c

```
#include "positive.h"
extern int increment;

int positive = 1;
static int negative = -1;
static void compensate(void){
    negative -= increment;
}
void signal(void){
    positive += increment;
    compensate();
}
```

main.c

```
#include <stdio.h>
#include "positive.h"

int increment = 3;

int main(){
    signal();
    printf("%d\n", positive);
    return 0;
}
```

Fejállományok

"header files": `.h`

- Modulok közötti interfész
  - `extern`
  - nem `static`
- Modulban és kliensében `#include`
  - típusegyeztetés
  - szerkesztési hibák megelőzése
- Előfeldolgozó (preprocessor)

Include guard

positive.h

```c
#ifndef POSITIVE_H
#define POSITIVE_H

extern int positive;
extern void signal(void);

#endif
```

main.c

```c
#include <stdio.h>
#include "positive.h"

int increment = 3;

int main(){
    signal();
    printf("%d\n", positive);
```

Include guard: többszörös beillesztés elkerülésére

low_level_module.h

```
#ifndef LOW_LEVEL_MODULE_H
#define LOW_LEVEL_MODULE_H
...
#endif
```

middle_module.h

```
#ifndef MIDDLE_MODULE_H
#define MIDDLE_MODULE_H
#include "low_level_module.h"
...
#endif
```

main.c

```
#include "low_level_module.h"
#include "middle_module.h"
...
```

Modul interfésze

vector.h (részlet)

```
#ifndef VECTOR_H
#define VECTOR_H

#define VEC_EOK      0
#define VEC_ENOMEM   1

struct VECTOR_S;
typedef struct VECTOR_S *vector_t;

extern int vectorErrno;

extern void *vectorAt( vector_t v, size_t idx);
extern void  vectorPushBack( vector_t v, void *src);

#endif
```

Fordítási egységek közötti függőségek

- Independent compilation
- Szerkesztés feladata
- Fordítási folyamat: `make` (és `cmake`), `bazel` stb.