Programozási nyelvek I. C++.

2. előadás

A jegyzetet *Pataki Norbert* előadásán *Lanka Máté* készítette.
Előadás ideje: 2017.02.22. 12:00-14:00.

Preprocesszálás:

Gyakorlatokon volt szó, hogy a C++-ban a fordítás milyen lépésekből áll. Az első a mai téma, azaz a preprocesszálás, a második a nyelvi fordítás, erről fog majd a félév lényegi része szólni, míg a harmadik lépés az összeszerkesztés, ez a C++ programok alapvető mechanizmusai.

Vannak fordítási egységek (pl. a.cpp, b.cpp). A preprocesszálás ezek kódok szövegszerű átalakítása. Ez azt jelenti, hogy ténylegesen nem alacsonyabb szintűre fordul a kód, hanem marad magas szinten, csak bizonyos részleteket átalakít, fájlokat kimásol, kitöröl, bemásol. A legelső lépés hatására az eredmény tehát még szintén egy magas szintű kód, tartalmazza az általunk leírt részleteket, fordítás itt még nem zajlik le. A konkrét példában így a'.cpp és b'.cpp lesz. Ebben a lépésben a header-öket is bekopipészteljük a kódba, ez még szintén nem fordítási egység. Ezután jön a tényleges nyelvi fordítás, itt kapunk már alacsonyabb szintű kódokat (a.obj, b.obj). Ez maga tárgykód, semmi köze a JAVA-s Objecthez. A linker ezen a ponton a fordítási egységekből feloldja a külső hivatkozásokat, és ebből készül el a futtatható állomány.

A mai előadáson magával a preprocesszálási fázissal fogunk foglalkozni. A header fájlokban a #-kal kezdődő sorok az úgynevezett *preprocesszálási direktívák*. Ez szabályozza, hogy a szöveget, forráskódot hogyan alakítsuk át, leírjuk, hogy ezt hogyan kell átalakítani, ezt egy szövegszerű átalakító eszköz alakítja át. (//„Sok az átalakítás szó.")

Milyen direktívákkal találkozhatunk és mire lehet ezeket használni?

- #include <iostream
  - ebben az esetben meg kell keresni az iostream tartalmát, és be kell másolni a #include <iostream> helyére;
  - azaz az *include* azt jelenti, hogy az adott fájl tartalmát be kell másolni erre a helyre;
  - ez a fájl hol van, honnan tudja a preprocesszor, hol találja?
  - ezek operációs rendszerektől, fájlrendszerektől függ;
  - viszont ez a szabványkönyvtárnak a része;
  - ha az ember feltelepíti a programot, telepítéskor bekerül, hogy milyen könyvtárakban kell keresni, ha < > közé tesszük az adott fájl nevét;
  - ez változik, ha újabb verzióra update-eljük a programunkat;
- #include „x.h"
  - ilyenkor ez egy relatív útvonalat ír le;
  - ebben az esetben az aktuális könyvtárban kell megkeresni a megadott fájlt, ez esetben az *x.h*-t;
  - ez így viszont mire jó?
  - a koncepció hasonló: bizonyos fordítási egységek máshol, máskor fordulnak le, szükségünk van rá, hogy a fordítóprogram leellenőrizze, hogy pl. jól hívom-e le a függvényt, jó paramétereket adok-e meg, stb;
    - ezeket header fájlokban szoktunk megoldani;

Példakód:

a.cpp

```
int f( int i )
{
        …
}
```

b.cpp

```
int f( int );
```

c.cpp

```
int f( int );
```

Ha az f függvényt „blabla” paraméterrel hívom meg, az hiba. Ha két paraméterrel hívom meg, az is baj. Ezt a compiler hogy tudja validálni? Jellemzően úgy csináljuk, hogy a *b.cpp*-be leírjuk a függvény *deklarációját*. Ha látja, hogy az f függvény int-et vár, akkor fordítási hibát vár, ha „blabla”-t adok meg. Ezt ha nem csak a *b.cpp*-ből, hanem a *c.cpp*-ből is akarom használni, az is megoldható, le lehet írni mindenhova. Viszont lehet, hogy az f függvény két paramétert fog későbbiekben várni. Ilyenkor ez az információ úgy kerül oda tartalmilag, ha a következőképp változtatjuk meg a fenti példakódot:

a.h

```
int f( int );
```

b.cpp

```
#include „a.h”
```

c.cpp

```
#include „a.h”
```

Ilyenkor az *a.h* tartalma bekerül végül a *b.cpp* és a *c.cpp* kódjaiba. Ez minden fordításnál megváltozik, nem kell minden egyes kódot karbantartani, ha az f függvény paraméterezése változik, ez a fordítás részeként meg tud változni.

Azzal együtt a C++ ismert arról, hogy elég lassan tud fordulni, lassabban mint más nyelvek. Ha minden kódot úgy kezdjük, hogy #include, akkor mindenhol ezt be kell másolni, majd utána mindezen hosszú kódokat lefuttatni, ez az egész lassítja a fordulást. Alapvetően viszont ez eddig egy jó koncepció. A másik jellegzetes C++ tulajdonság, hogy időnként a header fájlokba nemcsak függvény deklarációkat rakunk, hanem ha osztályműködést akarunk megvalósítani, azt is be lehet írni a következő módon. Tegyük fel, hogy egy komplex szám típust akarunk beírni:

complex.h

```
#ifndef COMPLEX__H          *
#define COMPLEX__H          *
class complex
{
        float re, im;
public:
        void set.re( double d) ;
        …
```

```
}
#endif                          *
```

complex.cpp

```
#include „complex.h”

void complex::set.re( double d)
{
        …
}
```

Az utóbbi egy fordítási egység, ebből fogunk kapni egy *complex.o*-t. A headert a fordító sokszor fogja feldolgozni. A C++ filozófiája, hogy inkább a fordítás legyen lassabb, de futás közben a futó program legyen hatékony. Ahol én itt szeretnék komplex számot, oda kell eljutni a *double re, im* tagoknak. Mit kell odagenerálni? Szükséges, hogy milyen adattagokból épül fel a kód, hány bájton kell ábrázolni. Ehhez az kell, hogy ez a két adattag eljusson egy másik fordítási egységhez.

Ha szeretnénk egy jól működő kódot arra, hogy másodfokú, komplex együtthatókkal rendelkező másodfokú egyenlet egyik megoldását megadjuk, ahhoz ez kell:

root.h

```
#include „complex.h”
complex root( complex a, complex b, complex c);
```

root.cpp

```
#include „root.h”
complex root( complex a, complex b, complex c)
{
        …
}
```

main.cpp

```
#include „complex.h”
#include „root.h”

int main()
{
        …
}
```

A gond az, hogy az #include-ok miatt kétszer fog bekerülni a *complex.h*. Kétszer van leimplementálva van a *complex.h*, megsérül az egyszer-definiálás szabálya, pedig erre szükség. Viszont innentől oda kell figyelni, hogy a szabályt be kell tartani. Erre van egy *include guard*, ebben levédhetem, hogy egy adott fordítási egység egyszer kerülhet be (a fenti kódban *-gal jelölt sor utólag került beírásra). Ez a megoldás arra szolgál, hogy ha már bekerült az adott header, akkor többször már ne másolja be!

Megjegyzés: az ifndef és define sorban elég egy _ jel is, de a kettő azért volt, hogy ez *include guard* jelleggel került be, mármint hogy ezt mi is tudjuk, illetve a preprocesszornak is egyértelműbb.

Ennek a #-olásnak régen nagy szerepe volt.

C++-ban	C-ben
• tömbök méretét fordítási időben ismertnek kell, hogy legyen	• ezt a 99-es szabványig nem figyelték a tömböknél

```
int v[20];
int x[20];

#define N 20
#define M 25
int a[N];
int b[N];
int c[M];
//Search & replace történik, ha meg akarjuk változtatni a tömb méretét. Ha a 20 nem elég, átírjuk fent;
```

Az ilyen define-okat még paraméterhetővé is lehet tenni. Tegyük fel, hogy:

```
#define SQ(x) x*x  //ez a makró;
std::cout<<SQ(5);  //kiírja, hogy 25;
std::cout<<SQ(1+4); //kiírja, hogy 1+4*1+4, aminek az értéke 9, mert nem paraméter-, hanem szövegszerű átadás történik;
```

Kiértékelésre mi a lehetőség? megoldás:

```
#define SQ(x)  ((x)*(x))
```

Bezárójelezés esetén már a helyes eredményt kapjuk, mert előbb értékeli ki az x-et, és utána helyettesíti be: ((1+4)*(1+4)) = 25;

(//Ugye milyen kurva izgalmas? – Mateu)

```
int i=3;
std::cout<<SQ(i);   //((i)*(i)) -> 9
std::cout<<SQ(i++); //((i++)*(i++)) -> 12
```

<u>Egyéb preprocesszor direkívák:</u>

```
#undef        //preprocesszor szimbólum megszüntetése
#ifndef
#ifdef        //az előző negálása (ha létezik)
#if
#else
#elif
```

Miért izgalmas ez? Arról szól ez a preprocesszor, hogy olyan kódokat fordítsunk le, ami ott érvényes. Például, közelről egy Linuxban és egy Windows-ban parancssorban máshogy törlünk le egy fájlt. Ezekkel a direktívákkal *feltételes fordítás*t lehet létrehozni, mert így lehet például kideríteni, hogy milyen platformon vagyok és ne futás közben derüljön ki, hogy pl. Linuxot használok-e, vagy Windowst.

Vannak ráadásul előredefiniált preprocesszor szimbólumok is, ezekre példák:

- op.rendszer;
  - le lehet kérdezni, hogy milyen op.rendszeren futtatom a kódot (pl. Linuxon *rm* a törlés, Windowson a *del*, utóbbin semmi szükség az *rm*-re);
- C / C++ szabvány:

- C-ben vagy C++-ban dolgozunk-e (ez hogyhogy kérdés?);

Sokszor a compiler teljesen mást lát, mint a programozó. Egy konkrét példa erre:

```
#define N 25;

int v[N];
```

Itt a compiler azt dobja vissza, hogy az *int v[N]*-nél gond van, pedig ezt adja át a fordítóprogram: *int v[25;]*. Ezért is érdemes minimalizálni a preprocesszor használatát.

Azzal együtt, visszatérve a korábbi példára, azáltal, hogy ezt makróként használják, van létjogosultsága. A függvényhívás egy absztrakció a számítógép szemszögéből nézve. Ha egy makró ezt a preprocesszor szintén meg tudja oldani, akkor nem kell a függvényhívás költségeit megfizetni.

A linkerre még rátérve, a linkert nem érdekli, hogy milyen programozási nyelven volt megírva a kód. Ha valamelyik kódnak viszont nincs meg a törzse, *undefinied reference* címen hibát fog visszadobni. Ezzel ellentétben létezik egy olyan fogalom, hogy *dinamikus linkelés*. A többség látott már Windowson *.dll* fájlokat. A felhasználó gépén ott van a lefordított állomány, de még nincs futtatva. Ezt futás közben keressük meg. Ez azért is praktikus, mert több ilyen *.dll* van egy helyen, és nem kell, hogy minden egyes fájlba be legyen linkelve. Ha viszont találnak hibát az implementációban, akkor letöltenek egy frissebb változatot úgy, hogy ezt a felhasználó észre sem veszi.