PROGRAMOZÁSI NYELVEK - CPP. GYAKORLAT JEGYZET

Szerkesztette: *Balogh Tamás*

2013. április 12.

Ha hibát találsz, kérlek jelezd a *info@baloghtamas.hu* e-mail címen!

Előszó

A jegyzet nem teljes és nem hibátlan. Az esetlegesen előforduló hibákért, és kimaradt részekért felelőséget nem vállalok. A jegyzet megtanulása nem biztos, hogy elég a sikeres vizsgához.

Az előadás sorszáma a hetek sorszámával egyezik meg. A szünetek, ünnepnapok miatt elmaradt órákat is számítom. Így számomra könyebb összehangolni a többi előadással, gyakorlattal.

Gyakorlatvezető adatai:

Név: Góbi Attila

E-mail: gobi@pnyf.inf.elte.hu

Weboldal: pnyf.inf.elte.hu/gobi

Szobaszám: D 2.620

Megjegyzések

1. A gyakorlatok a heteknek megfelelően vannak sorszámozva. A szünetek miatti elmaradt órákat is beleszámolom.

2. A jegyzet nem teljes, és nem hibátlan. Előfordulhat, hogy az órán leadott anyag nem teljes egészében szerepel benne.

1. Gyakorlat

Fordítás

g++ -Wall wh.cc
Legfontosabb kapcsolók:

-o <file> Ez lesz a fájl neve.

-o{0,1,2, 3} Optimalizálás.

-c Csak object file-t készít.

-I <path> Beállítja az include könyvtára.

-l <path> Beállítja a library könyvtárat.

-Wall -W Maximálisra állítja a warning szintet.

-Werror Hibánkénti figyelmeztetések.

-g Debug információkat beletesz a binary file-ba.

-ansi Letítlja az ansi szabvánnyal ellentétes kiterjesztéseket.

-pedantic Letilt minden nem szabványos kiterjesztést

Makrók

```
#ifndef                          #ifdef
#define          ~               #undef
#endif                           #endif
```

Deklaráció és definíció közötti különbség

A C++ programokban a neveket(azonosítókat) használat elött be kell vezetnünk. Össze kell kötnünk a változó nevét egy típusal.
`int a;` → Az a változó int típusú.
A deklaráció többet is jelenthet annál, mint hogy egyszerűen egy nevet kapcsol össze a név típusával.
A deklarációk többsége definíció is, azaz meg is határozza azt az egyedet, amelyre a név hivatkozik.
Példák:

```
char ch;
string s;
int count = 1;
const double pi = 3.1415926535897932385;
extern int error_number;
char* name = "Natasa";
char* season[ ] = { "tavasz", "nyár", "ősz", "tél" };
struct Date { int d, m, y; };
int day(Date* p) { return p->d; }
double sqrt(double);
template<class T> T abs(T a) { return a<0 ? -a : a; }
typedef complex<short> Point;
struct User;
enum Beer { Carlsberg, Tuborg, Thor };
namespace NS { int a; }
```

A fenti példák közül, csak a következőek *csak deklarációk*:

```
double sqrt(double);
extern int error_number;
struct User;
```

Az `sqrt` függvény kódját egy másik deklarációjában kell meg meghatározzni. Az `error_number` számára egy másik deklarációban kell lefoglalni a memóriát, és a `User` típus egy másik deklarációjában kell megadni, hogy a típus hogy nézzen ki.

A C++ programokban minden név számára pontosan egy definíció létezhet, ugyanakkor a nevet többször is deklarálhatunk. Egy egyed minden deklarációja meg kell, hogy egyezzen a hivatkozott egyed típusában.
A definíciók közül csak az alábbi nem határoz meg értéket:

```
char ch;
string s;
```

Minden deklaráció, amely értéket határoz meg, egyben definíciónak is minősül.

A deklarációk négy részből állnak: egy opcionális minősítőből, egy alaptípusból, egy deklarátorból, és egy szintén opcionális kezdőérték-adó kifejezésből. A függvény-és névtér-meghatározásokat kivéve a deklaráció pontosvesszőre végződik.
`char* kings[] = { "Antigónusz", "szeleukeusz", "Ptolemaiosz"};`
Itt az alaptípus `char`, a deklarátor a `*king[]`, a kezdőérték-adó rész pedig az `={...}`.
A minősítő egy kulcsszó, mint `virtual, extern`, és nem a típusra jellemző tulajdonságát határozza meg. A deklarátor egy névből és néhány opcionális operátorból áll. Példák deklarátor-operátorokra:

```
* const   //mutató
*const    //konstans mutató
&      //referencia
```

```
[]     //tömb
()     //() függvény
```

2. Gyakorlat

```
int main()
{
  char a = 'a';
  std::cout << a << std::endl;
}
```

1 byte-os, és az, hogy előjeles vagy előjel nélküli, az implementációfüggő.

5. Gyakorlat

Típusok

Típuskonverzió

`(int) a` → int-é kényszeríti

```
#include <stdio.h>
#include <iostream>

int main()
{
  long x = 42;
  printf("%s\n, (char *) x);  //SEGFAULT
  printf("%c\n", (char)x);

  long *px = &x;
  char*pc = (char *)px;
  for(int i =0; i < sizeof(px); ++i)
  {
    std::cout << (int)(pc[i]) << " "; //A lenyeg a lehetoseg
    std::cout << std::endl;
  }
}
```

`char *pc`. A `pc` típusa `char*`, így a `*pc` típusa `char`.
`pc+1` → Egy byte-tal arrébb mutató *pointer*. Ha dereferálni akarom, akkor `*(pc+1)`

```
#include <iostream>

void f(int x[]) //== int *x
{
  std::cout << sizeof(x) <<  " " << sizeof(x[0])<< std::endl;  //24 4
}

int main()
{
  int x[6]; //Stack-en 6 db tomb
  std::cout << sizeof(x) << " " << sizeof(x[0])<<std::endl; // 8 4
  f(x);

}
```

`int *x[6];` ==
`int (*x)[6];` == `int[][6];` ==
Minden tömb pointer, de a tömbökből álló tömbnél a tömbök elemeinek meg kell egyeznie. A mátrixban mindig az 1. dimenzió alakul át pointerré. Ha referenciaként akarom átadni a tömböt:

```
void g(int (&x)[6])
{
  std::cout << sizeof(x) <<  " " << sizeof(x[0])<< std::endl;
}
int main()
{
  int y[6] = {0,1,2,3,4,5};
```

```
  std::cout << sizeof(y) << " " << sizeof(y[0]) << std::endl;
  g(y);
}
```

Konstans

```
int main()
{
  const int i = 42;
  i = 63;     //ez nyilvan nem fog mukodni
  *pi = 63;
  int j = 63, k;
  pi = &j;
  int *pi = &i;   //ez megint nem fog mukodni, hisz az egyik int i* masik
     pedig const int i*
  int * const pj = &j   //int-re mutato konstans pointer
  pj = &k;      //nem mukodik, hisz konstans pointer
  const int * const pp;   //konstans intre mutato konstans pointer


}
```

```
#include  <iostream>

void f(int x) {
  std::cout << "f()" << std::endl;
}

void g(int x) {
  std::cout << "g()" << std::endl;
}

int main(int argc, char *[])
{
  void (*fp)() = argc>1?&f:&g;
  fp(argc);
}
```

6. Gyakorlat

+/- megoldás

```
#include <iostream>

int f(long x[2][4])
{
  std::cout << (*x)[2] << std::endl;
  std::cout << x[sizeof(x)/sizeof(*x)][0] << std::endl;
}

int main()
{
  long y[2][4] = { { 0,1,2,3}, {4,5,6,7}};
  f(y+1);
}
```

Kérdés: Mit ad vissza?
Válasz: 6 és 4
//Következő +/- STL-es lesz elvileg

STL

```
#include <iostream>

void sort(int *p, size_t n) // size_t == unsigned int, a változó mérete
{
  for(size_t i=n-1; i >0; --i)
  {
    for(size_t j=i-1; j<n-1; ++j)
    {
      if(p[j] > p[j+1])
      {
        int tmp = p[j];
        p[j] = p[j+1];
        p[j+1] = tmp;
      }
    }
  }
}

int main()
{
  int x[] = {2,6, -4, 3, 2, 1, 5, 9};
  sort(x, sizeof(x)/sizeof(*x));
  for(size_t i = 0; i < sizeof(x)/sizeof(*x); ++i)
  {
    std::cout <<s;
  }
}
```

Ez a kód nem elég *C++*-os. Túl sok aritmetikai művelet. pl `n-1`, stb . . . Ezt lehetne javítani.:

```
#include <iostream>

void sort(int *p, size_t n) // size_t == unsigned int, a változó mérete
```

```
{
  for(int * i=p+n-1; i > p; --i)
  {
    for(int * j=i; j<p+n; ++j)
    {
      if(*(j-1)> *j)
      {
        int tmp = *(j-1);
        *(j-1) = *j;
        *j = tmp;
      }
    }
  }
}

int main()
{
  int x[] = {2,6, -4, 3, 2, 1, 5, 9};
  sort(x, sizeof(x)/sizeof(*x));

  for(size_t i = 0; i < sizeof(x)/sizeof(*x); ++i)
  {
    std::cout << x[i] << ",  ";
  }

  std::cout << std::endl;


  //Általánosabb megoldás:
  for(int * p =x; p!= x + sizeof(x)/sizeof(*x); ++p)
  {
    std::cout << *p << ",  ";
  }
}
```

Láncolt lista

```
struct node
{
  node *next;
  int value;
};

nod*list_empty()
{
  return 0;
}

node *list_push(node *list, int value)
{
  node * p = new node;
  node->next = list;
  node->@alude = value;
  return p;
}
```

```
int main()
{
  node n = {0, 42}; //Inicializálás
  node begin = {&n, 137};

  for(node *p = &begin; p!=0; p = p->next)
    std::cout << p->value << ", ";
  std::cout << std::endl;


  list_push(list_push(list_empty(), 42), 137);
  return 0;
}
```

Ez az iterátok alapötlete. Van léptetés, ! = vizsgálat.

7. Gyakorlat

Láncolt lista

Van egy `Node struct` és egy `List struct`, ami ilyen `Node`-okat tartalmaz, és a lista függvényeit. A `void empty()` függvény nem jó megoldás, hisz meg lehet bármikot hívni. Ezt kiváltandó függvény a *konstruktor* (`list()`). A `void destroy()` függvény szintén nem jó megoldás, mert bármikor meg lehet hívni, és akkor megszüntetjük a listánkat. Ezt kiváltandó függvény a *destuktor* (`~list`).

Kitekintés

A {} közötti rész egy-egy elszeparált kódrész a C++-ban. Az itt létrehozott objektum a } után megszűnik. Példa:

```
int main()
{
  {
  list p;
  p.push(4);
  }
  //itt már nem hivatkozhatok a p-re
}
```

Adatelrejteés

- Public
- Private
- Protected

El kell rejteni a `first` adattagot, hogy ne lehessen nullázni. Viszont a konstruktort, destruktort nem helyes elrejteni.

Inicializálós lista

```
list () : first(0) {}
//ehelyett:
list()
{
  this->first = 0
}
```

Végigiterálás a listán

Ehhez szükség van egy `bool hasNext()` függvényre, ami megmondja, hogy van-e még következő elem egy `current` pointerre, ami mindig az aktuális elemre mutat, és egy `next()` függvényre, ami lépteti az aktuális elemet. A `current`-et természetesen el kell tárolni privát adattagként.

Tömb kiírása, iterátor

```
int main()
{
  int v[10] = (0,1,2,3,4,5,6,7,8,9);

  for(int i = 0; i < sizeof(v)/sizeof(v[0]); ++i)
  {
    std::cout << v[i] << std::endl;
  }
}
```

```
int main()
{
  int v[10] = (0,1,2,3,4,5,6,7,8,9);

  int *begin = v;
  int *end = sizeof(v)/sizeof(v[0]);

  for(int* i = begin; i != end; ++i)
  {
    std::cout << *i << std::endl;
  }
}
```

- Eleje
- Vége
- Léptetés
- Aktuális

9. Gyakorlat

Racionális szám

```
gcd()
{

}

struct rat
{
  //gcd(szaml, nev) = 1 && nev !=0
  int szaml, nev;

  rat(int n=0) : szaml(n), nev(1) {}
};

std::ostrream & operator << (std::ostream &s, const rat& r)
```

```
{
  return s << r.szaml << "/" << r.nev;
}

rat &operator *= (const rat &rhs)
{
  r.szaml *= rhs.szaml;
  r.nev *= rhs.nev / gcd(szaml, rhs.nev);
  return *this;
}

rat operator * (const rat &lhs, const rat &rhs)
{
  rat r;
  r.szaml = lhs.szaml * rhs.szaml;
  r.nev = lhs.neve * rhs.nev;
  return r;
}

int main()
{
  rat a,b = 2;
  std::cout << a << " " << b << std::endl;
  std::cout << a * b << std::endl;
  std::cout << 5 * a << std::endl;
}
```

Ha a * operátor a `struct`-on belül lenne, akkor csak a jobb oldalni tudna konvertálódni, a bal oldali nem, hiszen arra az adott objektumra hívnánk.

&&

A && mohó kiértékelésű.

Másoló konstruktor

// dengling pointer
Érték szerint átadunk valamit, amire nem írtunk másoló konstruktort, majd a függvény végén az meghívja a destruktort, így kitörli az átadott valamit, és majd később arra is destruktor, és máris segfault.
Példa: Lista másoló konstruktora

Lista másoló konstruktora

```
list (const list &l)
{
  if(l.first) //Hozzáférünk a first-höz, hisz osztályon belül vagyunk, és
     CPP-ben ez működik
  {
    first = new node;
    first->value = l.first->value;
    first->next = 0;

    node *from = l.first->next;
    node *to = first;
    while(from)
    {
      node *tmp = new node;
```

```
      tmp->next = 0;
      tmp->value = from->value;
      to->next = tmp;

      from = from->next;
      to = tmp;
    }
  }
}
```

`list p = q` utasítás során nem az értékadás operátor hívódik meg, hanem a másoló konstruktor.

Értékadás operátor

Figyelni kell rá, hogy `l = l` is lehetséges, így a kinyír, másol az segfaultot ad, ezért kell az if.

```
list &operator =(const list &rhs)
{
  if(this != &rhs)
  {
    //kinyir();
    //masol(rhs);
    list tmp(rhs);
    std::swap(tmp.first, first);
  }
  return *this;
}
```

Forrás

- ELTE IK programtervezői informatikus szak 2013 tavaszi féléves Programozási nyelvek - CPP gyakorlat alapján írt órai jegyzetem.
- Bjarne Stroustrup: A C++ programozási nyelv, 2001