C vagy C++?

Pataki Norbert

Programozási Nyelvek és
Fordítóprogramok Tanszék

Programozási Nyelvek I.

Témák

Motiváció

- A C és a C++ nyelv kapcsolata
- „Visszafele (backward, reverse) kompatibilitás” és következményei
- Objektum-orientált nyelvi eszközök, implementációs lehetőségek
- Érdemi programozás C++-ban
- Programozási Nyelvek II. – összehasonlítás Java-val
- Parancssori eszközök tudatos használata:
  - Fordítóprogram (build eszközök), Statikus elemzés
  - Linker, `nm`
  - Valgrind
  - Stb.

Félév rendje

- Tömbösített előadások/gyakorlatok
- Programozási Nyelv I. – C++
- Programozási Nyelv II. – Java

Számonkérés

- Gyakorlat: 3 db. +/-
- Vizsga: elméleti, gyakorlati rész
- Vizsgák: vizsgaidőszakban

Ajánlott irodalom

- `http://patakino.web.elte.hu/pny1/`
- Bjarne Stroustrup: A C++ Programozási Nyelv / The C++ Programming Language
- Scott Meyers: Hatékony C++ / Effective C++

Bjarne Stroustrup

- 1967 – Simula 67, első objektum-orientált programozási nyelv
- Dennis Ritchie, Ken Thompson: 1970-es évek eleje – C
- Bjarne Stroustrup: 1980-as évek eleje – C++
- Első ISO szabvány C++ – 1998
- További szabványok: C++03, C++11, C++14, C++17, (C++20)

A C++ koncepciói

- C: hatékony, jól fordítható, de nem objektum-orientált
- C++: hatékonyság megőrzésével, objektum-orientált programozás támogatással
- „Backward-kompatibilitás” – kezdetekben és ma
- (Jobb) nyelvi eszközök hibák elkerülésére:
  - Típusbiztos I/O
  - Preprocesszor használatának csökkentése
  - Deklarációk megkövetelése, láthatóság csökkentése
  - Stb.
- További fontos nyelvi elemek:
  - Sablonok, szabványkönyvtár, STL
  - Erőforrás-kezelés (pl. memóriakezelés)
  - Kivételkezelés
  - Stb.

Programozási paradigmák

C++ multiparadigmás nyelv:

- Procedurális programozás, C örökség
- Objektum-orientált programozás
- Standard Template Library: generikus programozás
- Funkcionális nyelvi elemek, metaprogramok, kódgenerálás

Hello World!

```
#include <iostream>

int main()
{
  std::cout << "Hello World"
            << std::endl;
}
```

- Fordítás:
  ```
  $ g++ -W -Wall -pedantic -ansi hello.cpp
  ```
- Futtatás (sikeres fordítás után):
  ```
  $ ./a.out
  ```

Minősített nevek

- `std::cout`, `std::endl`, `std::string`
- `using namespace std;`
  - Szükséges-e a `#include` direktíva?
    - C++: igen.
    - C: nem, de erősen ajánlott.
  - Szükséges-e a using direktíva? – C++: nem.
  - Szabad-e a using direktíva? – C++: nem mindenhol.
- `using std::cout;`
- `std` – a szabványkönyvtár névtere
- Java: namespace – package, using – `import`

Megközelítés C-ben

- Minden függvénynév legyen egyedi egy szoftver forrásában!
- Könyvtárak? Konvenciók. Kizárólag.
- OpenGL:
  - `glBindTexture` – bind a named texture to a texturing target
  - `glCreateShader` – Creates a shader object
  - `glDrawElements` – render primitives from array data
  - stb.
- PVM:
  - `pvm_pack()`
  - `pvm_reduce()`
  - `pvm_send()`
  - stb.

C++: namespace

```
namespace A
{
  int f( int x ) // A::f
  {
    return x + 1;
  }
}

namespace B
{
  int f( int x ) // B::f
  {
    return x * 2;
  }
}
```

Névterek egymásba ágyazása

```
namespace X
{
namespace Y
{
  void f()
  {
    std::cout << "Hello namespace";
  }
}
}

int main()
{
  f();
}
```

Fordítási hiba

```
$ g++ -W -Wall -pedantic -ansi ns.cpp
ns.cpp: In function ‘int main()’:
ns.cpp:20:5: error: ‘f’ was not declared in this scope
   f();
     ^
ns.cpp:20:5: note: suggested alternative:
ns.cpp:9:8: note:   ‘X::Y::f’
```

Névterek egymásba ágyazása – kijavítva

```
namespace X
{
namespace Y
{
  void f()
  {
    std::cout << "Hello namespace";
  }
}
}

int main()
{
  X::Y::f();
}
```

Névtelen névtér

a.cpp:

```
namespace
{
  int x, y;
}

void f()
{
  ++y;
}
```

a.c:

```
static int x, y;

void f()
{
  ++y;
}
```

Névterek megvalósítása

```
int x;
extern int y;

namespace
{
  int a;
}
namespace A
{
  int a;
}
namespace B
{
  int a;
}
```

Névterek megvalósítása, name mangling

```
$ g++ -c s.cpp
$ nm s.o
0000000000000000 B x
000000000000000c b _ZN12_GLOBAL__N_11aE
0000000000000004 B _ZN1A1aE
0000000000000008 B _ZN1B1aE
```

Változások C-hez képest

- Túlterhelés (overloading)
- Alapértelmezett (default) paraméter-értékek
- Inline függvények

Overloading

```
sum.cpp:
int sum( int a, int b )
{
  return a + b;
}

int sum( const int *p, int n )
{
  int ret = 0;
  for( int i = 0; i < n; ++i )
  {
    ret += p[ i ];
  }
  return ret;
}
```

Overloading megvalósítás

```
$ g++ -W -Wall -pedantic -ansi -c sum.cpp
$ nm sum.o
0000000000000000 T _Z3sumii
0000000000000014 T _Z3sumPKii
```

```
sum.h:
int sum( int, int );
int sum( const int*, int );
```

Overloading

```
#include "sum.h"
#include <iostream>

int main()
{
  std::cout << sum( 4, 3 );
  int v[] = { 7, 2, 1, 9};
  std::cout <<
     sum( v, sizeof( v ) / sizeof( v[ 0 ] ) );
}
```

Default paraméter-érték

```
#include <iostream>

void inc( int* p, int d = 1 )
{
  (*p) += d;
}

int main()
{
  int s = 5;
  inc( &s );
  std::cout << s << std::endl; // 6
  inc( &s, 5 );
  std::cout << s << std::endl; // 11
}
```

Összehasonlítás

```
void inc( int* p,
          int d = 1 )
{
   (*p) += d;
}
```

```
void inc( int* p )
{
   (*p)++;
}

void inc( int* p,
          int d )
{
   (*p) += d;
}
```

C vs C++

x.c:

```
int f( int s )
{
  return s;
}

$ gcc -c x.c
$ nm x.o
0...00000 T f
```

x.cpp:

```
int f( int s )
{
  return s;
}

$ g++ -c x.cpp
$ nm x.o
0...00 T _Z1fi
```

ex.cpp:

```
extern "C"
{

int f( int s )
{
  return s;
}

}

$ g++ -c ex.cpp
$ nm ex.o
0....000 T f
```

Inline függvény

```
inline double square( int d )
{
  return d * d;
}

int main()
{
  double s = square( 3.3 );
  s = square( s );
}
```

Ismétlés: pointer

- Pointer: olyan változó, aminek az értéke egy memóriacím:

```
int a = 3;
int b = 8;
int *p = &b;
*p = 1;
p = &a;
*p = 5;

std::cout << a << ' ' << b; // 5 1
```

Konstansok C++-ban

```
const int page_size = 20;

for( int i = 0; i < page_size; ++i )
{
  // ...
}
```

- Biztonság
- Hatékonyság

Konstansok C++-ban

```
const int page_size = 20;
++page;
// p.cpp:4:5: error: increment of read-only
                     variable ‘page_size’
page = 10;
p.cpp:5:8: error: assignment of read-only
                  variable ‘page_size’
```

Konstans-biztonság

```
const int page_size = 20;
int *p = &page_size;
*p = 10;
```

Konstans-biztonság

```
const int page_size = 20;
int *p = &page_size;
// error: invalid conversion from `const int*'
          to `int*'
*p = 10;
```

Konstans-biztonság deklarációkkal

- `int *ip;` – Megváltozhat, hogy hova mutat és a pointeren keresztül a mutatott érték nem változat (ezért nem mutathat konstansra)
- `const int* cip;` – A pointeren keresztül a mutatott érték nem változhat (mutathat konstansra, változóra)
- `int* const icp;` – Nem változhat meg, hogy a pointer hova mutat (de nem mutathat konstansra)
- `const int* const cicp;` – Akkor ez is mutathat változóra :-)

Konstans-biztonság?

```
char *msg = "Hello";
```

Lefordul: **warning:** deprecated conversion from string constant to ‘char*’

```
msg[ 1 ] = 'a';
```

esetén:

```
$ ./a.out
Segmentation fault (core dumped)
```

Referenciák

```
int a = 6;
int b = 1;
int& r = a;
++r;
std::cout << a; // 7
r = b;
std::cout << a; // 1
++r;
std::cout << a; // 2;
```

Pointer vs. Referencia

Pointer:

- C/C++
- Memóriacím
- Címképzés, dereferálás
- Megváltozhat, hogy hova mutat
- Nullpointer (`0`, `nullptr`)
- Pointer-aritmetika
- Nem kötelező inicializálni

Referencia:

- C++
- Álnév, alias
- -
- Mindig ugyanannak az álneve
- -
- -
- Kötelező inicializálni

Összehasonlítás

```
void inc( int* p)
{
  ++(*p);
}
...
int s = 4;
inc( &s );
inc( 0 ); // Jajj!
```

```
void inc( int& r )
{
  ++r;
}
...
int s = 4;
inc( s );
inc( 0 ); // Ford. hiba
```

Konstans referenciák

```
const int& two = 2;

int s = 5;
const int& r = s;
++s; // OK

++r; // error: increment of
        read-only reference `r'
```

Deklarációk

- `void f( int );`
- `void g( int& );`
- `void h( const int& );`
- `int& f();` – mire adhatunk vissza referenciát?