C++

Pataki Norbert

Eötvös Loránd Tudományegyetem,
Programozási Nyelvek és Fordítóprogramok
Tanszék, Budapest
patakino@elte.hu

ELTE Nyári egyetem 2011

Tartalom

Bevezetés

Referenciák

Objektum-orientált programozás

Sablonok

Standard Template Library

Template Metaprogramozás

C++0x

Összefoglalás

A C++ alapjai

- 1980-as évek óta; Bjarne Stroustrup
- C++ Szabvány: 1998/2003
- C++0x
- A C programozási nyelvre alapul; reverse kompatibilitás
- Hatékonyság
- Multiparadigmás programozás
- Fordítóprogramok: g++, Microsoft Visual Studio, Comeau, Intel, stb.

A C alapjai

- Imperatív, procedurális programozás
- Vezérlési szerkezetek, függvények
- Saját típusok: rekordok
- szabványkönyvtár

Hello World

```
#include <stdio.h>

int main()
{
  printf( "Hello World!\n" );
  return 0;
}
```

Bővítések C-hez képest

- Függvény túlterhelés
- Referenciák (álnevek)
- Objektum-orientált paradigma: osztályok, polimorfizmus
  - Osztályok
  - Öröklődés
  - Polimorfizmus
- Kivételkezelés
- Sablonok
- Kényelmesebb, biztonságosabb, bővebb szabványkönyvtár
- ...

Típusok

- `sizeof(char)` az egység
- `sizeof(char)` $\leq$ `sizeof(short)` $\leq$ `sizeof(int)` $\leq$ `sizeof(long)`
- `sizeof(float)` $\leq$ `sizeof(double)` $\leq$ `sizeof(long double)`
- `sizeof(bool)` $\leq$ `sizeof(long)`

Fordítás menete

- Preprocesszor: szövegátalakítás
- Nyelvi fordító: tárgykódok (object-ek) előállítás
- Linker: tárgykódok összefésülése

Hello World

```
#include <iostream>

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

Referenciák koncepciója

```
int i = 5;  // tárterület lefoglalása és
            // azonosító hozzárendelése
            // a tárterülethez

int& r = i; // azonosító hozzárendelése
            // a tárterülethez
```

Referenciák

- Valamilyen létező objektumnak az álneve
- Kötelező inicializálni
- Mindig ugyanannak az objektumnak az álneve, nem változhat meg, hogy mire hivatkozik
- Nincs „nullreferencia”

Referencia-szerinti paraméterátadás

```
void f( int& i )
{
  // ...
}

int x = 10;
f( x );   // OK
f( x+2 ); // ford.hiba
f( 5 );   // ford.hiba
```

Referencia-szerinti paraméterátadás

- Az alprogram lokális változója (`i`) egy álneve (alias-a) a meghíváskor átadott paraméternek (`x`).
- Nincs másolás: az eredeti változóval dolgozik az alprogram
- A függvényhívás során, amikor a függvényben `i` megváltozik, akkor a külső `x` is megváltozik (hiszen a kettő ugyanaz)
- A függvényhívás után `x` értéke megváltozhat a függvényhívás előtti állapothoz képest.
- Információ közvetítése a hívó irányába
- Nincs létrehozási, másolási, felszabadítási költség

Konstans referencia-szerinti paraméterátadás

```
void f( const int& i )
{
  // ...
}

int x = 10;
f( x );   // OK
f( x+2 ); // OK
f( 5 );   // OK
```

Konstans referencia-szerinti paraméterátadás

- Az alprogram lokális változója (`i`) egy olyan álneve (alias-a) a meghíváskor átadott paraméternek (`x`), amelyen keresztül nem változik meg az értéke.
- Nincs másolás: az eredeti változóval dolgozik az alprogram
- A függvényhívás után `x` értéke ugyanaz, mint a függvényhívás előtti.
- Nincs létrehozási, másolási, felszabadítási költség

Példák

```
void swap( int a, int b )
{
  int tmp = a;
  a = b;
  b = tmp;
}

void swap( int& a, int& b )
{
  int tmp = a;
  a = b;
  b = tmp;
}
```

Osztályok

```
class Complex
{
  double re;
  double im;
};
```

Osztályok, konstruktor

```
class Complex
{
  double re;
  double im;
public:
  Complex( double r = 0.0, double i = 0.0 )
  {
    re = r;
    im = i;
  }
};
```

Konstruktorok

```
int main()
{
  Complex zero;
  Complex i( 0.0, 1.0 );
  Complex nc = zero;
  Complex seven( 7.0 );
}
```

Osztályok, tagfüggvények

```
class Complex
{
  double re, im;
public:
  Complex( double r = 0.0, double i = 0.0 )
  {
    re = r;
    im = i;
  }

  double get_re() const { return re; }
  double get_im() const { return im; }
};
```

Osztályok, tagfüggvények

```
Complex c( 5.32, 7.34 );
std::cout << c.get_im()
          << std::endl;
```

Operátorok

- Túlterhelhető (pl. `operator+`) ill. Nem túlterhelhető operátorok (pl. `operator.`)
- Fix argumentumszám (kivétel `operator()`)

Operátorok

```
Complex a( 2.1, 4.4 );
Complex b( 4.2, 3.8 );
Complex c = a + b;

// a.operator+( b ) tagfüggvényként
// operator+( a, b ) globálisként

std::cout << c;

// std::cout.operator<<( c )  tagfüggvényként
// operator<< ( std::cout, c ) globálisként
```

Hibás operátor

```
class Complex
{
  double re, im;
public:
  // ...
  Complex operator+( const Complex& c ) const
  {
     return Complex( re + c.re, im + c.im );
  }
};
```

Hibás operátor

```
Complex a( 8.13, 5.4 );
Complex b = a + 7.3; // OK...
          // a.operator+( 7.3 )

Complex b = 7.3 + a; // ?
            7.3.operator+( a );
```

Operátorok

```
class Complex{...};

std::ostream& operator<<( std::ostream& o,
                          const Complex& c )
{
  o << c.get_re() << " + "
    << c.get_im() << 'i';
  return o;
}

Complex operator+( const Complex& a,
                   const Complex& b )
{
  return Complex( a.get_re() + b.get_re(),
                  a.get_im() + b.get_im() );
}
```

Öröklődés

```
#include <string>
#include <vector>
using namespace std;

int main()
{
  vector<int> v( 5 );   // OK
  string s1( 5 );       // ford. hiba
  string s2( 0 );       // futási hiba
}
```

Öröklődés

```
class safe_string
{
  std::string s;
public:
  safe_string( const char* p ): s( p ? p : "" )
  {
  }

  void push_back( char t ) { s.push_back( t) };

  int size() const { return s.size(); }
  // ...
};
```

Öröklődés

```
class safe_string: public std::string
{
public:
  safe_string( const char* p )
    : std::string( p ? p : "" )
  {
  }
  // konstruktorok nem öröklődnek...
};
```

Polimorfizmus

```
struct Base
{
  virtual void f() const
  {
    std::cout << "Base::f()"
              << std::endl;
  }

  virtual ~Base() {}
};
```

Polimorfizmus

```
struct Der
{
  virtual void f() const
  {
    std::cout << "Der::f()"
              << std::endl;
  }

  virtual ~Der() {}
};
```

Polimorfizmus

```
Base* bp = new Base();

bp->f();

delete bp;
bp = new Der();

bp->f();
```

Igény a sablonokra

- Típussal (és egyéb fordítási idejű adatokkal) paraméterezés
- Függvény és osztály sablonok létrehozása: nem kell előre megadni, hogy milyen típussal dolgozik a sablon

Függvénysablonok

```
template <class T>
const T& max( const T& a, const T& b )
{
  return a < b ? b : a;
}

...

std::cout << max( 4, 2 );
std::cout << max( 6.3, 7.8 );
```

Sablonok

- Sablonok példányosítása fordítási időben
- Paraméterdedukció
- Nem generálódik kód, amíg nem példányosítjuk
- Típus megszorítások

Az STL komponensei

- Konténerek
- Algoritmusok
- Iterátorok
- Funktorok
- Adapterek

Konténerek STL-ben

- Szekvenciális: `vector`, `list`, `deque`
- Asszociatív: `set`, `multiset`, `map`, `multimap`

vector

- Egybefüggő tárterületen azonos típusú elemek sorozata
- Elemek indexelve (0-tól kezdve)
- Elemek elérése – gyors
- Elem beszúrása vector végére – gyors
- Elem beszúrása máshova – lassabban

vector

```
#include <vector>
using namespace std;

vector<int> v;  // üres vector létrehozása
v.push_back(4); // elem beszúrása vector végére
v.pop_back();   // utolsó elem törlése

// Ötelemű vector létrehozása, minden eleme 3.2
vector<double> vd(5, 3.2);
int s = vd.size();

vd[1] = 4.76;      // Elemek elérése
vd.back() = 4.17; // Utolsó elem elérése
vd.front() = 1.2; // Első elem elérése
```

list

- Elemek szétszórtan helyezkednek el
- Elemek nincsen indexelve
- i-edik elérése lassú
- Elem beszúrása bárhova – gyors
- Következő / előző elem érhető el gyorsan

list

```
#include <list>
using namespace std;

list<double> l;     // üres vector létrehozása
l.push_front(4.5); // elem beszúrása lista elejére
l.pop_front();      // utolsó elem törlése
l.push_back(1.1);  // elem beszúrása lista végére

list<double> vl(3, 6.32);
int s = vl.size();

l.back() = 4.17;    // Utolsó elem elérése
l.front() = 1.2;    // Első elem elérése
l.sort();           // Lista rendezése
l.remove(1.2);      // Adott értékek törlése
l.reverse();        // Lista megfordítása
```

deque

- Kettős végű sor
- Egybefüggő tárterületek szétszórtan helyezkednek el
- Elemek indexelve vannak
- Elemek elérése – gyors
- Elem beszúrása tároló elejére / végére – gyors
- Elem beszúrása közepére – lassabban

deque

```
#include <deque>
#include <string>
using namespace std;

deque<string> d;
d.push_front( "Hello" );
d.push_back( "World" );
d[0] = "Goodbye";
d.back() = "Cruel World";
d.pop_back();
d.pop_front();
```

set

- Elemek sorrendje: rendezettség
- Minden elem legfeljebb egyszer szerepelhet
- Műveletek kihasználják a rendezettséget: gyors keresés, gyors beszúrás, stb.

set

```
#include <set>
#include <cstdlib>
using namespace std;

srand(time(0));
set<int> nums;
while( nums.size() < 5 )
{
  nums.insert( (rand() % 90)+1 );
}
```

multiset

- Elemek sorrendje: rendezettség
- Azonos elemek többször is szerepelhetnek
- Műveletek kihasználják a rendezettséget: gyors keresés, gyors beszúrás, stb.

multiset

```
#include <set>
using namespace std;

multiset<int> m;
m.insert(3);

int s = m.size();
int i = m.count(3);
```

map

- Asszociatív tömb: elemek indexelve
- Nem feltétlenül 0-tól kezdve
- Nem feltétlenül egymás utáni indexek
- Nem feltétlenül egészek
- Kulcs alapján rendezett tárolás

map

```
#include <map>
#include <string>
#include <iostream>
using namespace std;

map<string, string> phones;
phones["X.Y."] = "36(20)555-1234";
phones["A.B."] = "36(30)555-5555";
// ...
cout << phones["X.Y."];
```

Algoritmusok STL-ben

- Konténer-független, újrafelhasználható, globális függvénysablonok
- Általánosítás (generalizáció), paraméterezhetőség, de nem mehet a hatékonyság rovására
- Megoldások gyakori feladatokra
- Nem biztos, hogy egy algoritmus az összes konténerrel működik
- Pl. `find`, `sort`, `count`, `for_each`

Iterátorok STL-ben

- Konténerek bejárása
- Algoritmusok és konténerek közötti kapcsolat megteremtése
- Az iterátorok interface-e a pointer-aritmetikán alapul:
  - `operator++`, `operator*`, `operator==`, stb.

Funktorok STL-ben

- Egyszerű felhasználói osztályok, amelyeknek van `operator()`-a
- Ezen az `operator()`-on keresztül felhasználói kódrészletek hajtódnak végre a könyvtáron belül
- Tipikus alkalmazások: felhasználói rendezések, predikátumok, műveletek
- Unáris, Bináris funktorok
- Előny: inline-osítás, ...

Példa

```
class Print
{
  std::ostream& os;
public:

  Print( std::ostream& o ): os ( o ) { }

  template <class T>
  void operator()( const T& t )
  {
    os << t << ' ';
  }
};
```

Példa

```
std::set<double> sd;
std::list<std::string> ls;
std::deque<int> di;

std::for_each( sd.begin(), sd.end(),
               Print( std::cout ) );
std::for_each( ls.begin(), ls.end(),
               Print( std::cerr ) );
std::for_each( di.begin(), di.end(),
               Print( std::cout ) );
```

Példa

```
struct is_even: std::unary_function<int, bool>
{
  bool operator()( int i ) const
  {
    return 0 == i % 2;
  }
};
```

Példa

```
std::list<int> li;
// ...
std::list<int>::iterator i =
  std::find_if( li.begin(),
                li.end(),
                is_even() );

std::vector<int> ve;
// ...
std::vector<int>::iterator i =
  std::find_if( ve.begin(),
                ve.end(),
                std::not1( is_even() ) );
```

Példa

```
template <class T>
struct Less: std::binary_function<T, T, bool>
{
  bool operator()( const T& a, const T& b ) const
  {
    return a < b;
  }
};
```

Példa

```
std::set<int, Less<int> > a;

std::vector<double> v;
//...
v.erase(
  std::remove_if(
    v.begin(),
    v.end(),
    std::bind2nd( Less<double>(), 5.5 )
  ),
  v.end()
);
```

Spec. iterátorok

```
// másoljuk a std.input-ot std.output-ra...
std::copy(
  std::istreambuf_iterator<char>( cin ),
  std::istreambuf_iterator<char>(),
  std::ostreambuf_iterator<char>( cout ) );
```

Inserterek motivációja

```
const int N = 10;
double v[N];
...
double cv[N];

// v másolása cv-be:
copy( v, v + N, cv );

// ez általában nem működik a valódi STL
// konténerek esetében...
```

copy implementációja

```
template <class InIt, class OutIt>
OutIt copy( InIt first, InIt last, OutIt dest )
{
  while ( first != last )
  {
    *dest++ = *first++;
  }
  return dest;
}
```

back_inserter

```
double f( double x )
{
  // ...
}

list<double> values;
...
vector<double> results;

// f-et alkalmazzuk values összes elemére, és az
// adatokat results végére szúrjuk:
transform( values.begin(), values.end(),
           back_inserter( results ), f);
```

front_inserter

```
double f(double x)
{
...
}

list<double> values;
...
list<double> results;

// f-et alkalmazzuk values összes elemére, és
// az adatokat results elejére szúrjuk:
transform( values.begin(), values.end(),
           front_inserter( results ), f);
```

inserter

```
double f(double x)
{
  // ...
}

list<double> values;
// ...
vector<double> results;

// f-et alkalmazzuk values összes elemére, és az
// adatokat results közepére szúrjuk:
transform (
  values.begin(), values.end(),
  inserter( results,
            results.begin() + results.size() / 2 ),
            f
);
```

inserter

```
double f(double x)
{
  // ...
}

list<double> values;
// ...
multiset<double> results;

// f-et alkalmazzuk values összes elemére, és
// az adatokat results-ba szúrjuk (rendezett lesz)
transform( values.begin(), values.end(),
           inserter( results, results.begin() ), f );
```

back_inserter implementációja (vázlat)

```
template <class Cont>
class back_insert_iterator
{
  Cont* c;
public:
  back_insert_iterator(Cont& cont): c(&cont) { }

  back_insert_iterator& operator++()
  {
    return *this;
  }

  back_insert_iterator& operator++( int )
  {
    return *this;
  }
```

back_inserter implementációja (vázlat)

```
  back_insert_iterator&
  operator=( typename Cont::const_reference d )
          // const typename Cont::value_type& d
  {
    c->push_back(d);
    return *this;
  }

  back_insert_iterator& operator*()
  {
    return *this;
  }
};
```

back_inserter implementációja (vázlat)

```
template <class Cont>
back_insert_iterator back_inserter( Cont& c )
{
  return back_insert_iterator<Cont>( c );
}
```

Template Metaprogramozás

- Fordítási időben végrehajtodó kódok
- Rekurzív példányosítások, specializációk: Turing-teljes eszköz
- Optimalizációk, fordítási idejű ellenőrzések

Példa

```
template <int N>
struct Factorial
{
  enum { Value = N * Factorial<N - 1>::Value };
};
template <>
struct Factorial<0>
{
  enum { Value = 1 };
};

int main()
{
  std::cout << Fact<5>::Value << std::endl;
}
```

Expression templates

```
Array a, b, c, d, e;
...
e = a + b + c + d;
// e = ( ( ( a + b ) + c ) + d );
```

- Kényelmes, karbantartható
- Lassú, pazarló

Rekurzív template-ek

```
template <class Left, class Right>
class X { };

X< X<A,B>, X<C,D> > fa;
```

Rekurzív template-ek

```
struct plus
{
  static double apply( double a, double b)
  {
    return a+b;
  }
};
```

Rekurzív template-ek

```
template <class Left, class Op, class Right>
struct X
{
  Left     left;
  Right    right;

  X( Left l, Right r) : left( l ), right( r ) { }

  double operator[](int i)
  {
    return Op::apply( left[i], right[i] );
  }
};
```

Array

```
struct Array
{
  // szokásos: adattagok, konstruktorok, stb.

  template <class Left, class Op, class Right>
  void operator=( X<Left,Op,Right> expr)
  {
    for ( int i = 0; i < N; ++i)
      arr[ i ] = expr[ i ];
  }
};
```

Array

```
template <class Left>
X<Left, plus, Array> operator+( Left a, Array b)
{
    return X<Left, plus, Array>(a,b);
}
```

Szabványok

- ISO/IEC 14882:1998
- ISO/IEC 14882:2003
- TR1, C++0x

C++0x újítások

- Type inference: `auto`, `decltype`
- Lambda-kifejezések
- `nullptr`
- variadikus template, template typedef
- `long long` típus
- Párhuzamosság
- type traits, smart pointerek, hasító adatszerkezetek
- ...

Type inference

```
std::map<std::string,
         double,
         std::greater<std::string> > data;
...
std::map<std::string,
         double,
         std::greater<std::string> >::iterator i =
  data.begin();
```

Type inference

```
std::map<std::string,
         double,
         std::greater<std::string> > data;
...
auto i = data.begin();

auto two = 2;

auto pi = 3.14;
```

Type inference

```
const std::vector<int> v(1);
auto a = v[0];
decltype( v[0] ) b = 1;
```

foreach

```
int a[] = { 1, 2, 3, 4, 5 };
for (int &i: a)
{
    i *= 2;
}
```

Lambda függvény motivációja

```
std::vector<int> data;
// ...
int total = 0;
std::for_each( data.begin(),
               data.end(),
               [&total]( int x ) { total += x; });
```

Lambda függvények

- Funktor típus generálása a lambda függvény alapján
- A `operator()` visszatérési érték típusa: `decltype( x + y )`

```
[](int x, int y) { return x + y; }

[](int x, int y) -> int
{
  int z = x + y;
  return z + x;
}
```

Összefoglalás

- A C++ multiparadigmás nyelv
- Szabvány: 1998 óta, C++0x
- C + osztályok + sablonok + ...,
- Objektum-orientáltság: öröklődés, polimorfizmus
- STL: generikus programozás
- Template metaprogramozás