C++ alprogramok

Pataki Norbert

2013. március 22.

Bevezetés

- Alprogram: olyan nyelvi szerkezet, amelynek segítségével új nevet rendelhetünk egy kódrészlethez, hogy azt később, amikor csak akarjuk, egyszerűen végrehajthassuk.
- A kódrészlet végrehajtásának kezdeményezése, azaz az *alprogram meghívása* a kódrészlethez rendelt név (és esetleg paraméterek) megadásával történik.

Példa

```
int max( int a, int b )
{
  if ( a < b )
     return b;
  else
     return a;
}

...
int k = 4;
int l = 15;
int m = 7;

std::cout << max( k, max( l, m ) ) << std::endl;
```

Alprogramok előnyei

- Karbantarthatóság, újrafelhasználhatóság
- Olvashatóság: Azonosító - kifejezi a funkcionalitást
- Felhasznált változók láthatóságának csökkenése
- Könyvtárak

Alprogramok

1. Pontosan egy típus – visszatérési érték (esetleg `void`)
2. Alprogram neve
3. Paraméterlista (0, 1, 2, . . .): paraméterek típusa, illetve az alprogramban használt neve

```
int fnev( int a, double b, char c )
{
  // függvény törzs
}
```

Deklaráció, definíció

- Deklaráció:

```
void f( int i );
```

- Minimális info, hogy az aktuális fordítási egységben hogyan használható a függvény azonosító
- Nem definiálja az algoritmust; nem garantálja annak helyességét, stb.
- Más fordítási egységben megírt függvények meghívására
- Linker oldja fel a hivatkozásokat a fordítás után

Deklaráció, definíció

- Definíció

```
void f( int i )
{
  std::cout << 2 * i << std::endl;
}
```

- One Definition Rule

Függvények, eljárások

- A függvények: a paraméterekből kiszámolnak valamilyen információt (pl. sin)
- Az eljárások: a paramétereket átalakítják, nem visszaadják a megváltoztatott információt; (pl. rendezés)
- A C/C++ nem különbözteti meg a függvényeket és az eljárásokat, minden alprogram ,,függvény".
- Ha nem akarunk semmilyen információt visszaadni: `void` visszatérési típust adhatunk meg.
- A visszatérési értéket nem kötelező eltárolni a hívó oldalon.
- Mellékhatás, eredmény

Függvények, eljárások

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

void setup( std::vector<int>& v, int i )
{
  if ( i < 0 )
    return;
  // ...
}
```

Függvények, eljárások

```
std::string reverse( std::string s )
{
  std::string r;
  for( int i = s.size() - 1; i >= 0; --i )
    r.push_back( s[i] );
  return r;
}

std::cout << reverse( "gezakekazeg" );
```

Függvények, eljárások

```
void reverse( std::string& s )
{
  const int n = s.size();
  int i = 0;
  while( i < n - 1 - i )
  {
    std::swap( s[i], s[n - 1 - i] );
    ++i;
  }
}

std::string s = "indulagorogaludni";
reverse( s );
std::cout << s;
```

Példák

```
#include<cmath>
// Derékszögű háromszögek átfogójának meghatározása
// a két befogó ismeretében:
double atfogo( double bef1, double bef2 )
{
  return pow( bef1 * bef1 + bef2 * bef2, 0.5 );
}
```

Rekurzív alprogramok

- Rekurzív alprogramok: közvetve vagy közvetlenül önmagukat meghívják
- A program végrehajtásának egy pontján több példányban is aktív lehet

```
int factorial( int n )
{
  if ( 0 == n )
    return 1;
  else
    return n * factorial( n - 1 );
}
```

Rekurzív alprogramok

```
int max( int i, int j )
{
  return i < j ? j : i;
}

int max( std::vector<int> v )
{
  if ( 1 == v.size() )
    return v[0];
  else
  {
    int i = v.back();
    v.pop_back();
    return max( i, max(v) );
  }
}
```

Paraméterek

```
double sin( double x )
{
  // x: formális paraméter
  // ...
}

double d = 1.43;
double sd = sin( d ); // d: aktuális paraméter
```

Paraméterátadás

- érték-szerint
- referencia-szerint
- konstans referencia-szerint

Érték-szerinti paraméterátadás

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

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

Érték-szerinti paraméterátadás

- A függvény meghívásakor létrejön egy új változó `i` névvel, amely az alprogram lokális változója.
- Létrejöttekor az `i`-be másolódik az aktuális paraméter értéke (copy konstruktor)
- A függvény végrehajtása közben az aktuális paraméterre nincs hatással, annak egy másolatával dolgozunk.
- Költségek: létrehozás, másolás, felszabadítás

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

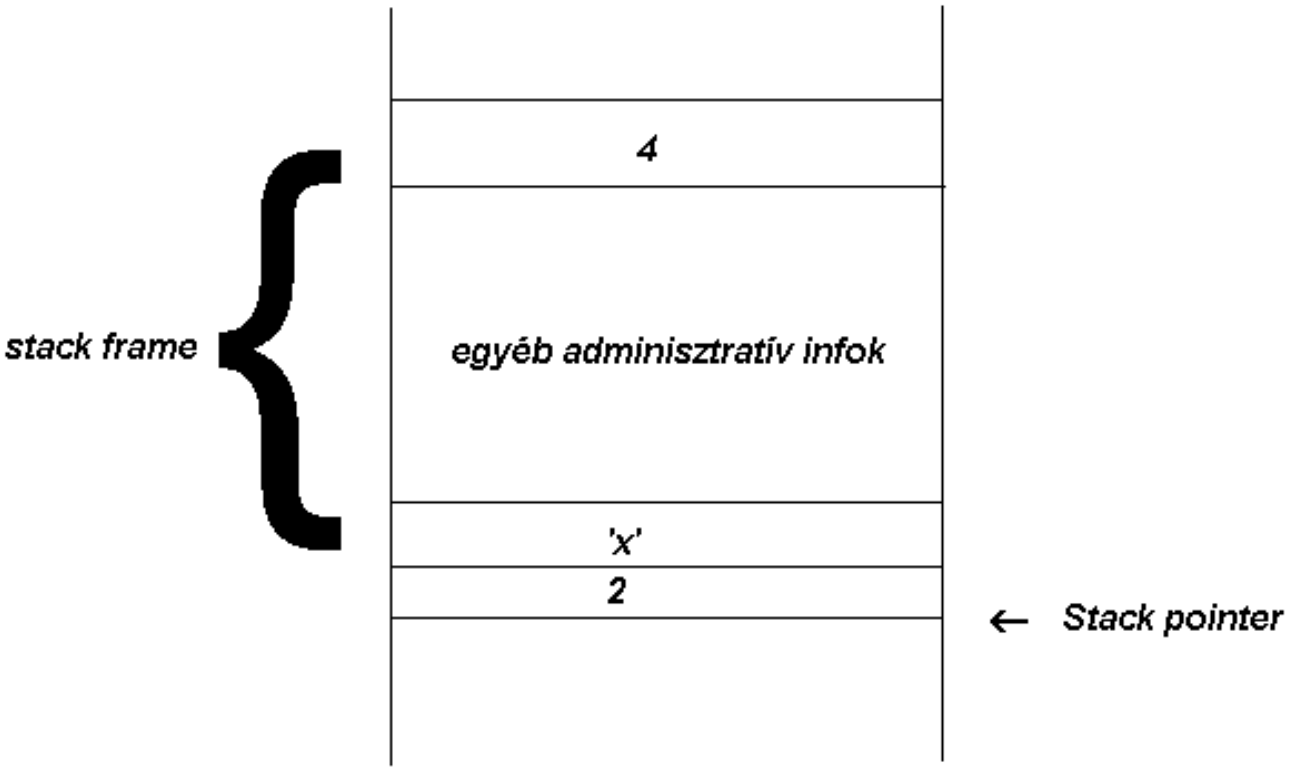
- Az alprogram lokális változója (`i`) egy olyan álneve (alias-a) a meghíváskor átadott paraméternek (`x`), amelyen keresztül nem változik meg az értéke.
- Nincs másolás: az eredeti változóval dolgozik az alprogram
- A függvényhívás után `x` értéke ugyanaz, mint a függvényhívás előtti.
- Nincs létrehozási,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

Stack




```
void f(int i, char c)
{
   int k  = 4;
   g(i , k);

}


f(2, 'x');
```

Függvények túlterhelése – overloading

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

void f( std::string s )
{
  // ...
}

// ...
f( 4 );
f( "blabla" );
// de visszatérési érték alapján nem megy...
```

Példák

```
double accumulate( const std::vector<double>& v )
{
  double sum = 0.0;
  for( int i = 0; i < v.size(); ++i )
  {
     sum += v[i];
  }
  return sum;
}
```

Példák

```
int count( const std::vector<int>& v, const int& t )
{
  int cnt = 0;
  for( int i = 0; i < v.size(); ++i )
  {
     if ( v[i] == t )
       ++cnt;
  }
  return cnt;
}
```

Példák

```
int max( const std::vector<int>& v )
{
  int m = v[0];
  for( int i = 0; i < v.size(); ++i )
  {
     if ( m < v[i] )
       m = v[i];
  }
  return m;
}
```

Példák

```
void sort( std::vector<double>& s )
{
  for( int i = 0; i < s.size(); ++i )
  {
    for( int j = 0; j < i; ++j )
    {
      if ( s[i] < s[j] )
        std::swap( s[i], s[j] );
    }
  }
}
```

STL Konténerek

- Szekvenciális: `vector`, `list`, `deque`
- Asszociatív: `set`, `multiset`, `map`, `multimap`
- Adapter: `stack`, `queue`, `priority_queue`

Példák

```
std::string reverse( const std::string& s )
{
  std::stack<char> st;
  for( int i = 0; i < s.size(); ++i )
    st.push( s[i] );

  std::string r;
  while( !st.empty() )
  {
    r.push_back( st.top() );
    st.pop();
  }
  return r;
}
```

Példák

```
double accumulate( const std::list<double>& l )
{
  double sum = 0.0;
  for( std::list<double>::const_iterator ci = l.begin();
       ci != l.end();
       ++ci )
  {
     sum += *ci;
  }
  return sum;
}
```

Tagfüggvények

- `v.pop_back();`
- `st.push( s[i] );`
- `r.push_back( s[i] );`
- `v.size()`
- `l.begin()`
- `l.end()`