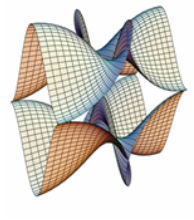

Prirodoslovno-matematički fakultet
Matematički odsjek
Sveučilište u Zagrebu

RAČUNARSKI PRAKTIKUM I

Vježbe 11 - Funktori

v2018/2019.

Sastavio: Zvonimir Bujanović

Funkcijski objekti (funktori)

Objekt klase koja ima nadograđen `operator()` zovemo funkcijski objekt ili funktor.

```
template<class T>
struct dupler
{
    void operator()( T &x ) { x = x+x; }
};

int main()
{
    int zz = 5; string oo = "AB";

    dupler<int> DI;
    DI( zz ); // zz=10

    dupler<string>()( oo ); // oo="ABAB"

    return 0;
}
```

Funkcijski objekti (funktori)

`dupler` je primjer unarnog funktora.
Možemo napraviti funktore s proizvoljnim brojem parametara (pa i 0).

```
template<class T>
struct max // binarni funktor
{
    T operator()( T x, T y )
    {
        return (x < y) ? y : x;
    }
};

int main ()
{
    max<string> MS;
    string oo = MS( "ab", "aac" ); // oo="ab"

    int zz = max<int>()( 3, 7 );   // zz=7
}
```

Funkcijski objekti (funktori)

Kombinacijom funktora i iteratora možemo implementirati razne algoritme koji rade s proizvoljnim spremnicima (`vector`, `list`, ...).

```
template<class Iter, class Functor>
void for_each( Iter st, Iter en, Functor f )
{
    for( Iter i = st; i != en; ++i )
        f( *i );
}

int main()
{
    list<int> L;
    L.push_back( 2 ); L.push_back( 6 ); // L = (2, 6)

    for_each( L.begin(), L.end(), dupler<int>() );
    // sada: L = (4, 12)
}
```

Funkcijski objekti (funktori)

Promjenom funktora poslanog `for_each` možemo s elementima liste napraviti nešto sasvim drugo.

```
template<class T>
struct printer
{
    void operator()( T x ) { cout << x << " "; }
};

int main()
{
    list<int> L;
    L.push_back( 2 ); L.push_back( 6 ); // L = (2, 6)

    for_each( L.begin(), L.end(), printer<int>() );
    // ispis na ekran: "2 6 "
}
```

Zadatak 1

Napišite generičku funkciju za sortiranje spremnika, te pripadne binarne funktore tako da imamo sljedeću funkcionalnost:

```
int X[] = { 2, 6, 4, 7, 8 };
vector<int> V( X, X+5 );

sort( V.begin(), V.end(), manji<int>() );

for_each( V.begin(), V.end(), printer<int>() );
// ispise "2 4 6 7 8 "

sort( X, X+5, veci<int>() );

for_each( X, X+5, printer<int>() );
// ispise "8 7 6 4 2 "
```

Ovdje su `veci`, `manji` (binarni) funktori koji vraćaju `bool`.
Takve funktore zovemo (binarni) predikati.

Funktori i algoritmi u STL-u

```
#include <algorithm>
```

`for_each` i `sort` već dolaze u STL-u, koriste se na posve jednak način; `sort` je izuzetno efikasno implementiran.

Postoje i predikati `greater`, `less` koji rade istu stvar kao redom predikati `veci`, `manji` koje smo implementirali u Zadatku 1:

```
sort( V.begin(), V.end(), greater<int>() );
```

`set` i `map` mogu primati i dodatni funktor (default=`less`) koji kaže kako treba sortirati elemente:

```
set<int, greater<int>> S;
S.insert( 1 ); S.insert( 7 ); // S = (7, 1);
```

Postoji cijeli niz gotovih algoritama koji rade s funktorima.

Pretraživanje

`find` – vraća iterator prvog elementa jednakog trećem parametru.

```
list<int> L;
L.push_back(3); L.push_back(1); L.push_back(7);

list<int>::iterator r = find( L.begin(), L.end(), 1 );
// r = iterator na element jednak 1
```

`find_if` – vraća iterator prvog elementa za kojeg je istinit unarni predikat poslan kao treći parametar.

```
struct pozitivan {
    bool operator()(int x) { return x > 0; }
};

list<int> L;
L.push_back(-3); L.push_back(-1); L.push_back(7);

list<int>::iterator r = find_if( L.begin(), L.end(), pozitivan() );
// r = iterator na prvi pozitivni element (7)
```

Pretraživanje

`count` – vraća broj elemenata jednakih trećem parametru.

```
list<int> L;
L.push_back(1); L.push_back(1); L.push_back(7);

int r = count( L.begin(), L.end(), 1 );
// r = 2
```

`count_if` – vraća broj elemenata za koje je istinit unarni predikat poslan kao treći parametar.

```
list<int> L;
L.push_back(3); L.push_back(-1); L.push_back(7);

int r = count_if( L.begin(), L.end(), pozitivan() );
// r = 2
```

copy/replace

`copy` – kopira sve elemente iz intervala određenog s prva dva parametra na mjesto koje počinje s trećim parametrom.
Uoči: u ciljnom spremniku mora biti dovoljno mjesta!

```
int X[] = {5, 2, 3, 6, 5};
list<int> L(5);

copy( X, X+5, L.begin() );
```

`replace` – svaka pojava trećeg parametra zamjenjuje se četvrtim.

```
list<int> L;
L.push_back(1); L.push_back(4); L.push_back(1);

replace( L.begin(), L.end(), 1, 5 );
// L = (5, 4, 5)
```

fill/generate

`fill` – popuni raspon određen prvim dvama parametrima s trećim parametrom.

```
list<int> L(5);

fill( L.begin(), L.end(), 3 );
// L = (3, 3, 3, 3, 3)
```

`generate` – kao `fill`, ali treći parametar je funktor bez parametara, u donjem slučaju pointer na C-ovu funkciju `rand()`.

```
list<int> L(5);

generate( L.begin(), L.end(), rand );
// L = (234324, 823, 9162, 12312, 5685), npr.
```

transform

`transform(st1, en1, st2, f)` – nad svakim elementom raspona [`st1`, `en1`⟩ izvrši unarni funktor `f` i rezultate spremi u raspon koji počinje sa `st2`.

```
struct negiraj
{
    int operator()( int x ) { return -x; }
};

list<int> L;
L.push_back(3); L.push_back(5); // L = (3, 5)
vector<int> V(5);

transform( L.begin(), L.end(), V.begin(), negiraj() );
// V = (-3, -5, 0, 0, 0)
```

transform

`transform(st1, en1, st2, st3, f)` – nad svakim elementom raspona [`st1`, `en1`⟩ i odgovarajućim elementom raspona koji počinje na `st2` izvrši binarni funktor `f` i rezultate spremi u raspon koji počinje sa `st3`.

```
struct zbrajaj
{
    int operator()( int x, int y ) { return x+y; }
};

list<int> L; L.push_back(3); L.push_back(5); // L = (3, 5)
list<int> M; M.push_back(6); M.push_back(7); // M = (6, 7)
vector<int> V(5);

transform( L.begin(), L.end(), M.begin(), V.begin(), zbrajaj() );
// V = (9, 12, 0, 0, 0)
```

Umjesto `zbrajaj` možemo koristiti i već gotovi `plus<int>`.

reverse/random_shuffle

`reverse` – preokreće redoslijed elementa u rasponu [`st`, `en`⟩.

```
list<int> L;
L.push_back(3); L.push_back(5); L.push_back(7); // L = (3, 5, 7)

reverse( L.begin(), L.end() );
// L = (7, 5, 3)
```

`random_shuffle` – na slučajan način promjeni redoslijed elemenata u rasponu [`st`, `en`⟩; radi samo sa `vector`-om ili poljem.

```
vector<int> V;
V.push_back(3); V.push_back(5); V.push_back(7); // V = (3, 5, 7)

random_shuffle( V.begin(), V.end() );
// V = (5, 7, 3), npr.
```

next_permutation

Zamislimo da sve permutacije elemenata nekog spremnika poredamo u niz koji je leksikografski sortiran.

`next_permutation` – mijenja poredak elemenata u spremniku tako da on bude iduća po redu pemutacija u nizu, vraća `true` ako iduća permutacija postoji, a `false` ako ne postoji.

```
int X[] = {1, 2, 3};
while( 1 )
{
    for_each( X, X+3, printer<int>() ); cout << endl;

    if( !next_permutation( X, X+3 ) )
        break;
}

// Ispis:
// (1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)
```

next_permutation

`next_permutation` može primiti i dodatni parametar (binarni predikat) ako je niz permutacija sortiran na drugačiji način.

```
int X[] = {3, 2, 1};
while( 1 )
{
    for_each( X, X+3, printer<int>() ); cout << endl;

    if( !next_permutation( X, X+3, greater<int>() ) )
       break;
}

// Ispis:
// (3, 2, 1), (3, 1, 2), (2, 3, 1), (2, 1, 3), (1, 3, 2), (1, 2, 3)
```

Zadatak 2

Odaberite bilo koja 3 opisana algoritma i napišite svoju implementaciju.

Za popis svih dostupnih algoritama vidi:
http://www.cplusplus.com/reference/algorithm/

Funktori i funkcije

Kao parametre algoritmima možemo slati i (pokazivače na) funkcije.

```
int zbr( const int &a, const int &b ) { return a+b; }

transform( L.begin(), L.end(), M.begin(), V.begin(), zbr ); // OK!
```

Kod deklaracija (npr. set-a) u predlošku trebamo navesti tip. Funkcija nije tip, pa ju ne možemo koristiti na tom mjestu (no vidi i ovo).

```
bool veci( const int &a, const int &b ) {
    return a < b ? false : true;
}
set<int, veci> S; // compile error, veci nije tip!
```

Pozive naših algoritama i onih iz STL-a razlikujemo ovako:

```
::for_each( V.begin(), V.end(), printer<int>() );     // nas
std::for_each( V.begin(), V.end(), printer<int>() ); // STL-ov
```