Standardní knihovna C++, 2. část
Petr Mejzlík 2. 12. 98

Obsah:

- Příklad šablonové třídy (Stack)
- Multimap
- Seznamy
- Funkční objekty
- Kontejnery polymorfních objektů
- Šablony jako prostředek pro výpočty v době překladu

Příklad šablonové třídy

Následující jednoduchý příklad pochází ze Stroustrupovy knihy The C++ Programming Language (3rd ed.).

```
template<class T> class Stack {
  T* v;
  int max_size;
  int top;
public:
  class Underflow { };
  class Overflow { };
  Stack(int s);   // konstruktor
  ~Stack( );   // destruktor
  void push(T);
  T pop();
};
```

```
template<class T>void Stack<T>::push (T c)
{
if (top==max_size) throw Overflow ( );
v[top] = c;
top = top +1;
}

template<class T> T Stack<T>::pop ( )
{
if (top==0) throw Underflow ( );
top = top -1;
return v[top];
}
```

Máme-li k dispozici uvedené definice, můžeme zásobníky vytvářet a využívat následujícím způsobem:

```
Stack<char> sc;  // zásobník znaků
Stack<complex> scplx;
Stack<list<int> > sli;
// zásobník obsahující seznamy celých čísel

void f ( )
{
sc.push('c');
if (sc.pop( ) != 'c') throw Bad_pop( );

scplx.push(complex(1,2));
if (scplx.pop( ) != complex(1,2))
throw Bad_pop( );
}
```

Multimap

```
template
<class Key, class T, class Cmp = less<Key> >
class std::multimap {
public:
typedef Key key_type;
typedef T mapped_type;
typedef pair<const Key, T> value_type;
//...
}
```

- asociativní kontejner, který může (na rozdíl od map) obsahovat více záznamů se stejným klíčem

- má *více* aplikací než map

Operace:

iterator insert(const value_type&)

vloží dvojici (klíč, hodnota), vrátí iterátor, který na ni ukazuje

iterator find(const key_type& k)

vrátí iterátor ukazující na první prvek s klíčem k

iterator lower_bound(const key_type& k)

vrátí iterátor ukazující na první prvek s klíčem k

iterator upper_bound(const key_type& k)

vrátí iterátor ukazující za poslední prvek s klíčem k

pair<iterator,iterator>equal_range(const key_type& k)

vrátí oba iterátory najednou

Příklad:

Opět telefonní seznam, tentokrát ale může mít jeden člověk více telefonních čísel:

```
#include <map>

multimap<string,string> seznam;

seznam.insert(make_pair("Stroustrup","001-12345678"));
seznam.insert(make_pair("Stroustrup","001-87654321"));

typedef multimap<string,string>::const_iterator I;
pair<I,I> b = seznam.equal_range("Stroustrup");
for (I i = b.first; i != b.last; i++) cout << i*.second
<< endl;
```

Operace nad seznamy

```
 template <class T> class list {
public:

void splice(iterator pos, list& x);
// přesune prvky x před pozici pos bez kopírování
void splice(iterator pos, list& x, iterator p);
void splice(iterator pos, list& x, iterator first,
iterator last);

void merge(list&); // sloučí dva setříděné seznamy
template <class Cmp> void merge(list&, Cmp);
// verze s porovnávací funkcí zadanou v parametru

void sort(); //setřídí seznam
template <class Cmp> void sort(Cmp);
// verze s porovnávací funkcí zadanou v parametru

void unique(); //odstraní po sobě jdoucí duplicity
...
}
```

Funkční objekty

- generické algoritmy (for_each, find, count, ...) volají uvnitř f(), kde f je jejich parametr

- f přitom *nemusí* být nutně funkce, poslouží i třída s předefinovaným operátorem ()

Příklad:

```
template <class T> class Sum {
T res;
public:
Sum(T i=0) : res(i) {} // inicializace
void operator() (T x) { res += x; }
T result() const { return res; }
}

void soucet(list<double>& ld)
{
Sum<double> s;
for_each(ld.begin(), ld.end(), s);
cout << "Součet = " << s.result() << endl;
}
```

Seznamy polymorfních objektů

Příklad:

```
class shape {
virtual void draw() = 0; // čistě virtuální
funkce
...
};
class circle : public shape {...};
class rectangle : public shape {...};
```

Jak vytisknout seznam obrázků?

Řešení č.1:

```
void draw(shape* p)
{ p-> draw(); }

void f(list<shape*>& sh)
{
for_each(sh.begin(), sh.end(), draw);
}
```

Musíme ale psát pro každý takový případ speciální globální funkci?

Řešení č.2:

```
void g(list<shape*>& sh)
{
for_each(sh.begin(), sh.end(), mem_fun(&shape::draw);
}
```

mem_fun() je šablona ze standardní knihovny, která patří do kategorie *adaptérů*. Vrací výsledek členské funkce, která je jejím parametrem. (Jak to funguje: &shape::draw se uvnitř funkce mem_fun použije jako parametr konstruktoru třídy mem_fun_t, což je návratový typ funkce mem_fun. Tento konstruktor uloží adresu shape::draw do pomocné prměnné pmf. Ve třídě mem_fun_t je pak definován operátor (), který volá pmf().)

Šablony jako prostředek pro výpočty v době překladu

Program, při jehož překladu se vypočte faktoriál čísla 3:

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

template<int N> class factorial {
public:
enum {value=N*factorial<N-1>::value};
};

class factorial<1> {
public: enum {value=1};
};

int main() {
const int c = factorial<3>::value;
cout << "Faktoriál 3 = " << c << endl;
}
```

Obecně lze přinutit překladač C++ aby provedl v podstatě jakýkoliv výpočet. Využívá se toho při *optimalizaci* programů: lze tak napsat přenositelné optimalizované knihovny, např. pro numerické výpočty. Výsledky jsou plně srovnatelné s optimalizovaným kódem FORTRANu 77. Viz
http://monet.uwaterloo.ca/blitz/.