STL – Standard Template Library

BD5B37PPC – Programování v jazyce C/C++

Stanislav Vítek

Katedra radioelektroniky
Fakulta elektrotechnická
České vysoké učení v Praze

Přehled témat

- Část 1 – Ukázky použití STL
  - Motivace
  - Součet čísel
  - Součin čísel
  - Průměr
  - Medián
- Část 2 – STL kontejnery
  - Obecné vlastnosti
  - Sekvenční kontejnery
  - Asociativní kontejnery
  - Další STL kontejnery
  - Iterátory
- Část 3 – Knihovna algoritmů
  - Generické funkce

Část I

Ukázky použití STL

I. Ukázky použití STL

Motivace

Vyřešíme několik jednoduchých úloh, při řešení využijeme prostředky knihovny STL:

- součet čísel na vstupu – accumulate,
- součin čísel – funktor,
- výpočet průměru – pair
- výpočet mediánu

Příklady jsou úmyslně velmi jednoduché, slouží hlavně pro demonstraci konceptů, které se uplatňují při používání STL.

I. Ukázky použití STL

Součet čísel na standardním vstupu

- Konvenční řešení

```
#include <iostream>

using namespace std;

int main ()
{
  int sum = 0, x;
  while ( cin >> x )
    sum += x;
  cout << sum << endl;

  return 0;
}
```

Součet čísel na standardním vstupu

- Řešení pomocí STL

```
#include <iostream>
#include <numeric>
#include <iterator>

using namespace std;

int main ( void )
{
  cout << accumulate (istream_iterator<int> (cin),
    istream_iterator<int> (), 0) << endl;

  return 0;
}
```

Součet čísel na standardním vstupu

- Ukázkové STL řešení je postavené na 2 principech: iterátoru a generické funkci.
- Iterátor je datový typ definovaný v STL, který
  - odkazuje na nějaká data či pozici,
  - data mohou být v nějaké kolekci (`vector`, `set`, ...), nebo např. v souboru,
  - ukázkový iterátor umožní čtení celých čísel z daného streamu (zde `cin`),
  - iterátor je typicky generická třída,
  - rozhraní iterátoru umožňuje čtení (dereference), posun vpřed (`++`) a porovnávání dvojice iterátorů (`==`, `!=`),
  - některé iterátory mají ještě další rozhraní, viz dále.
- `istream_iterator<int>(cin)` je volání konstruktoru iterátoru. Tento iterátor bude při dereferenci číst data typu integer ze standardního vstupu.
- `istream_iterator<int>()` je volání konstruktoru iterátoru. Tento iterátor znamená konec vstupu (EOF).

Součet čísel na standardním vstupu

- Generická funkce `accumulate`:
  - generická funkce definovaná v STL,
  - slouží k výpočtu součtu dat, rozsah vstupních dat je daný dvojicí iterátorů,
  - funkce pomocí iterátorů projde zadaný rozsah dat,
  - získané hodnoty přičte k existující hodnotě,
  - počáteční hodnota je daná posledním parametrem (zde 0),
  - výsledek je návratovou hodnotou funkce (zde požadovaný součet).
- Podobných generických funkcí pracujících se dvojicemi iterátorů je v STL celá řada:
  - kopírování,
  - mazání,
  - filtrování,
  - řazení,
  - ...

I. Ukázky použití STL

Součin čísel na vstupu

```
#include <iostream>
#include <numeric>
#include <iterator>

using namespace std;

int multiply ( int a, int b )
{
  return a * b;
}

int main ( void )
{
  cout << accumulate (istream_iterator<int> (cin),
    istream_iterator<int> (), 1, multiply) << endl;
  return 0;
}
```

Součin čísel na vstupu

- Generická funkce `accumulate` je v STL přetížena:
  - v obecné variantě lze zadat operaci, která nahradí sčítání,
  - zde je použita funkce `multiply`, která skládá čísla násobením,
  - funkce je zavolaná pro každou vstupní hodnotu,
  - v parametrech dostane dosavadní výsledek a načtenou hodnotu,
  - návratovou hodnotou je upravený výsledek,
  - počáteční hodnotu jsme změnili na 1.
- Tímto způsobem lze hledat např. maximum, minimum, ...
- Doplnění funkce a její předání do nějaké generické funkce je v STL běžné. Lze předat funkci, případně i funkční objekt – funktor.

  V C++ 11 lze pak na místě funktorů použít i lambda funkce.

I. Ukázky použití STL

Průměr čísel na vstupu

```
#include <iostream>
#include <numeric>
#include <iterator>

using namespace std;

pair<int,int> avg (const pair<int,int> & a, int b)
{
  return make_pair (a.first + b, a.second + 1);
}

int main ( void )
{
  pair<int,int> x = accumulate (istream_iterator<int>(cin),
    istream_iterator<int> (), make_pair(0,0), avg);
  cout << (double) x . first / x . second << endl;
  return 0;
}
```

Průměr čísel na vstupu

- STL deklaruje generickou strukturu `pair`.
  - má dvě složky: first a second,
  - typ složek je dán generickými parametry,
  - lze využít pro rychlé nadeklarování struktury, kde potřebujeme 2 složky. Např. zde pro výpožet průměru potřebujeme znát součet a počet čísel na vstupu.
- V C++ 11 existuje typ `tuple`, který umožní sloučení libovolného počtu složek.
- Snadnost deklarace typů `pair` a `tuple` svádí k použití i tam, kde to není vhodné. Jména složek nejsou moc vypovídající, často bývá lepší zavést si vlastní typ s výstižněji pojmenovanými složkami.

Příklad – pair

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

int main ()
{
  pair<int,int> pair1, pair3; // pár integer - integer
  pair<int,string> pair2;     // pár integer - string

  pair1 = make_pair(1, 2);    // vlož 1 a 2 do pair1
  pair2 = make_pair(1, "XXL") // vlož 1 a "XXL" do pair2
  pair3 = make_pair(2, 4)
  cout << pair1.first << endl;  // 1
  cout << pair2.second << endl; // XXL

  if (pair1 == pair3) cout<< "Pary jsou stejne\n";
  else cout << "Pary nejsou stejne\n";
  return 0;
}
```

Příklad – tuple

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

int main()
{
  tuple<int, int, int> tuple1;        // tuple celých čisel
  tuple<int, string, string> tuple2; // integer a 2 stringy

  tuple1 = make_tuple (1,2,3);        // vlož 1, 2 a 3
  tuple2 = make_tuple (1, "Size", "XXL");

  int id; string first_str, last_str;

  tie (id, first_str, last_str) = tuple2;
  cout << id << " " << first_str << " " << last_str;
  /* 1 Size XXL  */
}
```

Průměr čísel na vstupu

- Uvažujme rozšíření, kdy chceme zpracovávat vstup a počítat průměr pouze z čísel, která jsou větší než zadaná mez:
- můžeme opustit funkci accumulate a nahradit ji konvenčním cyklem,
- to však nemusí být vhodné, pokud bychom měli procházet složitější struktury (např. strom, tam se přístup s iterátory hodí),
- do funkce pro zpracování dat bychom chtěli přidat "kontext" – informaci o prahu, kde se již vstupní hodnoty započítáváme,
- v STL se k tomu používá funkční objekt – funktor. Vlastní výpočet se provede v přetíženém operátoru () funkčního objektu,
- funkční objekt (funktor) – objekt, který lze použít jako funkci. Třída musí přetěžovat operátor (). Pak lze za instanci zapsat () s parametry a tím "zavolat" funkci.

Průměr čísel na vstupu

```
#include <iostream>
#include <numeric>
#include <iterator>
#include <sstream>

using namespace std;

class Mez
{
  int mez, pocet, suma;
public:
  Mez (int t) : mez (t), pocet (0), suma (0) { }
  double operator () (double dummy, int x) {
    if (x > mez)
      pocet++; suma += x;
    return (double) suma / pocet;
  }
};
```

Průměr čísel na vstupu

```
...
int main ( int argc, char * argv[] )
{
  int thr;
  if (argc < 2 || !(istringstream (argv[1]) >> thr))
    return 1; // error
  cout << accumulate (istream_iterator<int> (cin),
    istream_iterator<int> (), 0.0, Mez (thr)) << endl;
  return 0;
}
```

Průměr čísel na vstupu

- Instance třídy `Mez` je předaná jako parametr funkci `accumulate`.
- Tento objekt obsahuje kontext (součet, počet a práh pro vstupní hodnoty).
- S každým dalším vstupem je zavolán operátor `()` této instance.
- Instance třídy `Mez` na základě vstupu upraví své členské proměnné.
- Návratovou hodnotou je aktuální hodnota průměru.
- Vstupní parametr `dummy` je ignorován (není potřeba se zajímat o předchozí hodnotu průměru).

I. Ukázky použití STL

Výpočet mediánu

Náš program má načíst celá čísla ze vstupu a určit medián:

- počet čísel na vstupu není známý,
- pro určení mediánu potřebujeme čísla uložit do pole,
- pole seřadíme,
- zobrazíme prostřední prvek.

Pro uložení čísel využijeme pole proměnné velikosti – `vector`:

- vector funguje jako dynamicky alokované pole zvoleného typu,
- logika přidávání alokuje potřebný prostor, pokud vyčerpáme aktuální kapacitu.

Výpočet mediánu

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

using namespace std;

int main ( void )
{
  vector<int> numbers;
  for (int x = 0; cin >> x; )
    numbers.push_back (x);
  sort (numbers.begin (), numbers.end ());
  cout << numbers [numbers.size ()/2] << endl;
  return 0;
}
```

Výpočet mediánu

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

using namespace std;

int main ( void )
{
  vector<int> numbers;
  copy (istream_iterator<int> (cin),
    istream_iterator<int> (), back_inserter (numbers) );
    sort (numbers.begin (), numbers.end () );
  cout << numbers [numbers.size() / 2] << endl;
  return 0;
}
```

Výpočet mediánu

STL funkce `copy` kopíruje hodnoty z rozmezí daného dvojicí iterátorů do cíle daného posledním iterátorem:

- problém je, že cíl je zatím prázdný,
- chtěli bychom data přidávat na konec,
- k tomu lze použít specializovaný iterátor `back_inserter`,
- tento iterátor nepřepisuje prvky existující v kontejneru, ale přidává nové prvky na konec (s alokací potřebného prostoru).

Řazení obsahu zvládne funkce `sort`:

- řadí se část (okno) kontejneru daná dvojicí iterátorů,
- kritériem řazení je porovnání prvků operátorem `<`,
- pokud je kritérium jiné, lze funkci sort předat porovnávací funkci nebo funktor.

Ukázkové řešení neuvažuje situaci, kdy je počet čísel na vstupu sudý (průměr z prostředních hodnot).

Část II

STL kontejnery

II. STL kontejnery

STL kontejnery

- Kontejner (kolekce) je abstraktní datový typ určený na organizované skladování prvků konkrétního typu podle určitých pravidel. Na standardní knihovně je připravena celá řada šablon užitečných kontejnerů.
- Kontejnery se od sebe významně liší způsobem přístupu k prvkům, možnostmi vkládání a rušení prvků a také časovou složitostí jednotlivých operací.
- Mezi kontejnery můžeme počítat i typ `std::string` pro zpracování řetězců znaků.
- Všechny STL kontejnery mají public kopírující konstruktor a operátor přiřazení (=).
- Základní dělení:
  - Sekvenční kontejnery – sekvenční přístup k prvkům
  - Asociativní kontejnery – libovolný (náhodný) přístup k prvkům

II. STL kontejnery

STL kontejnery – array

- Zapouzdřené pole fixní velikosti.
- Lze kontrolovat indexy, snadné kopírování.

```
#include <array>

array<int, 10> x;
x.fill (-1);
x[3] = 100; // unchecked access
array<int, 10> y(x);
for (array<int,10>::size_type i = 0; i < x.size(); i++)
  x[i] += 100;

copy (x.begin(), x.end(),
  ostream_iterator<int> (cout, "\n"));

array<int,20> z(x); // !!!
```

STL kontejnery – vector

- Zapouzdřené pole proměnné velikost.
- Lze kontrolovat indexy, snadné kopírování, rozšiřování.

```
#include <vector>

vector<int> x;
x.resize (10); // 0 0 0 ...  0
x.push_back (100);
x.insert (x.begin() + 5, 200); // make room
x[3] = 120; // overwrite, no index check
x.at (4) = 150; // overwrite, check index
copy (x.begin(), x.end (),
  ostream_iterator<int> (cout, "\n"));
// copy range
vector<int> y (x.begin () + 1, x.begin () + 9);
sort (y.begin(), y.end ());
copy (y.begin(), y.end (),
  ostream_iterator<int> (cout, "\n" ));
```

STL kontejnery – deque

- Oboustranná fronta, dokáže nahradit stack i queue.
- Lze přidávat i odebírat z obou konců, přístup přes index.

```
#include <deque>

deque<int> x;
x.push_back (100); x.push_front (200);
x.push_back (300); x.push_front (400);
x.insert (x.begin() + 1, 500);
// copy, read, display & pop
for (deque<int> y (x); !y.empty (); y.pop_front())
  cout << y.front () << endl;
x.erase (x.begin () + 1, x.begin () + 3 );
// iterate in reverse direction
deque<int>::reverse_iterator it;
for (it = x.rbegin(); it != x.rend (); ++it)
  cout << *it << endl;
```

STL kontejnery – list

- Obousměrný spojový seznam.
- Vkládání/odebírání prvku z libovolné z pozice (začátek, konec, iterátorem).

```
#include <list>

list<int> x;
x.push_back (100); x.push_back (200); x.push_front (300);
list<int>::iterator pos = x . begin ();
pos++;
x.insert (pos, 400); x.erase (pos + 1); // !!!
pos++;
x.erase (pos);
// iterate forward
list<int>::iterator it;
for (it = x . begin(); it != x.end (); ++it)
  cout << *it << endl;
```

II. STL kontejnery

STL kontejnery – set

- Množina prvků (prvek buď je nebo není obsažen).
- Vkládání/mazání/testování přítomnosti prvku.

```
#include <set>

set<int> x;
x.insert (20);
x.insert (100);
x.insert (1000);
cout << (x.count (20) == 1 ? "present" : "not present" )
  << endl;
set<int>::iterator pos = x.find (1000);
if (pos != x.end ()) x.erase (pos);
set<int>::iterator it;
for (it = x . begin(); it != x.end (); ++it)
  cout << *it << endl;
```

STL kontejnery – map

- Tabulka (klíč – hodnota).
- Vkládání/mazání/čtení prvku.

```
#include <map>

map<string,int> x;
x.insert (make_pair ("test", 10 ));
x["key"] = 20;
x["testkey"] = x["test"] + x["key"];
map<string,int>::const_iterator it;
for (it = x . begin(); it != x . end (); ++it )
  cout << it -> first << "->" << it -> second << endl;
map<string,int>::iterator pos = x.find ("test");
cout << (pos != x.end () ? "present" : "not present") <<
  endl;
x.erase (pos);
```

II. STL kontejnery

Další STL kontejnery

- `forward_list`, C++ 11
  - jednosměrný spojový seznam,
  - obdoba `list` s nižší paměťovou režií, ale s omezením operací,

    Např. nelze snadno vkládat před pozici iterátoru.
  - sekvenční kontejner, ForwardIterator.
- `stack`
  - zásobník,
  - implementován jako wrapper na deque (příp. list nebo vector),
  - omezení rozhraní na LIFO,
  - sekvenční kontejner, nemá iterátor.
- `queue`
  - fronta, implementována jako wrapper na deque příp. list,
  - omezení rozhraní na FIFO,
  - sekvenční kontejner, nemá iterátor.

Další STL kontejnery

- `priority_queue`
  - fronta s prioritami,
  - prioritní ukládání je dosaženo tím, že se data ukládají do datové struktury halda,
  - sekvenční kontejner, nemá iterátor.
- `multiset`, `multimap`
  - třídy odpovídají `set` a `map`,
  - stejná hodnota klíče může být uložena vícekrát,
  - `#include <map>`, `#include <set>`
  - asociativní kontejnery, ForwardIterator.
- `bitset`
  - pole `bool` hodnot zadané velikosti,
  - kompaktní uložení (po jednotlivých bitech),
  - nemá iterátor.

II. STL kontejnery

Iterátory

- Iterátory v STL slouží pro:
  - procházení kontejneru,
  - určení místa v kontejneru

    Např. kam má být vložen prvek, který prvek smazat.
  - určení rozsahu prvků, které má nějaká funkce zpracovávat

    Řadit, projít, sečíst, zkopírovat, ...
  - vyhledávání v kontejneru.
- Iterátory jsou zpravidla šablony tříd, jejich přetížené rozhraní se chová jako ukazatel:
  - `*` – dereference (přístup k prvku),
  - `==` a `!=` – porovnání iterátorů (stejná/různá pozice),
  - `++` – posun vpřed (prefix i postfix forma, prefix bývá efektivnější).
- Inicializace
  - "na začátek" – funkční hodnota členské funkce `begin()`,
  - "za konec" – `end()`.
- Některé kontejnery neumožňují iterování: `stack`, `queue`, `priority_queue`.

Iterátory

InputIterator – dereference pro čtení, posuv vpřed a test na (ne)rovnost. Nelze restartovat (2x dereferencovat). Příklad: `istream`.

OutputIterator – dereferenci pro zápis, posuv vpřed a test na (ne)rovnost. Nelze restartovat. Příklad: `ostream`.

ForwardIterator – jako InputIterator, navíc dereference čtení/zápis i vícekrát na stejné pozici. Příklad: `forward_list`.

BidirectionalIterator – jako ForwardIterator, navíc možnost posunu zpět operátorem `--`. Příklady: `list`, `set`, `map`, ...

RandomAccessIterator – jako BidirectionalIterator, navíc možnost posunu vpřed/zpět o zadaný počet pozic (operace `+=` a `-=`), indexace operátorem `[n]`, rozdíl iterátorů, relační operátory. Příklad: `vector`, `deque`.

Operace s interátory

advance (iterator i, int distance) – zvyšuje nebo snižuje pozici iterátoru `i` o hodnotu `distance`

distance (iterator first, iterator last) – vrací vzdálenost mezi dvěma iterátory

next (iterator i, int n) – vrací iterátor následující o `n` prvků za iterátorem `i`

prev (iterator i, int n) – vrací iterátor předcházející o `n` prvků před iterátorem `i`

begin () – vrací ukazatel na první prvek kontejneru

end () – vrací ukazatel na první prvek za posledním prvkem kontejneru

Část III

Algoritmy

III. Algoritmy

Generické funkce

Generické funkce

for_each(st,en,fn) – zavolá funkci/funktor `fn` pro všechny prvky z rozsahu `st` až `en`,

find(st,en,x) – hledá první výskyt prvku `x` v rozsahu `st` až `en`,

find_if(st,en,fn) – hledá první prvek v rozsahu `st` až `en`, pro který platí `fn`,

copy(st,en,dst) – kopíruje prvky z rozsahu `st` až `en` do cíle `dst` (a dále), volbou iterátoru `dst` lze použít i pro vstup / výstup dat, přesunutí či přidání,

copy_if(st,en,dst,fn) – kopíruje jako `copy`, ale pouze prvky, které vyhoví filtru `fn` (funkce / funktor),

transform(st,en,dst,fn) – kopíruje jako `copy`, prvky při kopírování transformuje `fn`,

remove if(st,en,fn) – v rozsahu `st` až `en` odstraní prvky, pro které platí `fn`,

sort(st,en,fn) – seřadí prvky v rozsahu `st` až `en`, kritériem řazení je `fn`.

Generické funkce

Další běžné úlohy a podpůrné STL funkce:

- binární vyhledávání: lower bound, upper bound, binary search,
- slučování (merge) a operace založené na slučování (průnik, sjednocení seřazených polí): `merge`, `set_union`, `set_intersection`,
- práce s datovou strukturou halda (heap): `make_heap`, `push_heap`, `pop_heap`,
- minimum a maximum: `min_element`, `max_element`,
- agregace: `accumulate`,
- test vlastnosti pro všechny prvky: `all_of`, `any_of`, `none_of`.

Příklady

```
bool myMaxCompare(string a, string b) {
  return (a.size() > b.size());
}

bool myMinCompare(string a, string b) {
  return (a.size() > b.size());
}

int main() {
  int x=4, y=5;
  cout << max(x,y);   // prints 5
  cout << min(x,y);   // prints 4

  string s = "smaller string", t = "longer string---";
  cout << max(s, t, myMaxCompare); // longer string---
  cout min(s, t, myMinCompare);    // smaller string
}
```