Алгоритми

Як правило, вміст контейнерів обробляється алгоритмами. Хоча кожен контейнер визначає свій власний набір основних операцій, більш складні дії описуються у вигляді алгоритмів. Крім того, вони дозволяють одночасно працювати з двома різними типами контейнерів. Для доступу до алгоритмів з бібліотекі STL' в програму необхідно включити заголовок <**algorithm**>.

В бібліотеці STL визначено велику кількість алгоритмів. Вони перераховані в табл. 24.5. Всі алгоритми являють собою шаблонні функції. Це означає, що їх можна застосовувати до будь-якого типу контейнерів. Усі шаблонні алгоритми будуть описані у частині 4, а в наступному розділі наведено ілюстративний приклад.

Таблиця 24.5. стандартні алгоритми

Алгоритм	Призначення
adjacent_find	Знаходить пару сусідніх елементів, які збігаються між собою, і повертає ітератор, що посилається на їх перше входження
binary_search	Виконує бінарний пошук у впорядкованій послідовності
copy	Копіює послідовність
copy_backward	Аналогічний алгоритму copy(), але копіювання починає з останнього елемента
count	Повертає кількість елементів послідовності
cout_if	Повертає кількість елементів послідовності, що задольняють певну умову
equal	Визначає, чи збігаються елементи двох діапазонів
equal_range	Повертає діапазон, в який можна вставити елементи, що не порушують порядок послідовності
fill і fill_n	Заповнює діапазон заданим значенням
find	Знаходить діапазон, що містить задане значення, і повертає ітератор, який посилається на його перше входження
find_end	Знаходить діапазон підпослідовності. Повертає ітератор, установление на кінець підпослідовності, що міститься в заданном діапазоні
find_first_of	Знаходить перший елемент послідовності, що співпадає з елементом з іншої послідовності
find_if	Знаходить перший елемент послідовності, що задовольняє определенном умові
for_each	Застосовує функцію до діапазону елементів
generate і generate_n	Привласнюють елементам діапазону значення, повернуті функцією-генератором
includes	Визначає, чи містить одна послідовність іншу
inplace_merge	Об'єднує один діапазон з іншим. Обидва діапазону повинні бути впорядковані за зростанням. Результатом є упорядкована послідовність
iter_swap	Міняє місцями два значення, на які посилаються аргументи
lexicographical_compare	Виконує лексикографическое порівняння двох послідовностей
lower_bound	Визначає перший елемент послідовності, яка не мень¬ше, ніж задане значення
make_heap	Створює купу на основі послідовності
max	Повертає найбільше з двох значень
max_element	Повертає ітератор, установлений на максимальний елемент діапазону
merge	Об'єднує дві впорядковані послідовності, поміщаючи результат в третю послідовність
min	Повертає найменше з двох значень

min_element	Повертає ітератор, який посилається на мінімальний елемент после¬довательності
mismatch	Знаходить першу розбіжність двох послідовностей. Повертаються ітератори на обидва елементи
next_permutation	Формує наступну перестановку елементів послідовності
nth_element	Впорядковує послідовність так, щоб всі її елементи, менше заданого елемента E, передували йому і іншим елементам
partial_sort	впорядковує діапазон
partial_sort_copy	Впорядковує діапазон, а потім копіює його в результуючу по-ність
partition	Впорядковує послідовність так, щоб всі її елементи, задоволь-летворяющіе заданій умові, передували всім іншим
pop_heap	Міняє місцями перший і останній елементи, а потім перебудовує купу
prev_permutation	Формує попередню перестановку елементів послідовності
push_heap	Заштовхує елемент в кінець купи
random_shuffle	Перетасовував послідовність
remove, remove_if, remove_copy і remove_copy_if	Видаляє елементи із зазначеного діапазону
replace, replace_copy, replace_if i replace_copy_if	Замінює елементи зазначеного діапазону
reverse і reverse_copy	Міняє порядок слідування елементів діапазону на протилежний
rotate і rotate_copy	Виконує ліву циклічну перестановку елементів
search	Знаходить підпослідовність всередині заданої послідовності
search_n	Знаходить n-е входження елемента в послідовність
set_difference	Створює послідовність, що містить незбіжні елементи двох послідовностей
set_intersection	Створює послідовність, що містить збіжні елементи двох послідовностей
set_symmetric_difference	Створює послідовність, яка є симетричною різницею двох послідовностей
set_union	Створює послідовність, яка є об'єднанням двох послідовностей
sort	Впорядковує діапазон
sort_heap	Впорядковує купу всередині зазначеного діапазону
stable_partition	Впорядковує послідовність так, щоб всі елементи, удовле-яка творить певній умові, передували всім осталь¬ним. Розбиття послідовності фіксується. Це значить, що при упорядкуванні зберігається взаємне розташування двох послідовностей
stable_sort	Впорядковує діапазон. Порядок фіксується. Це значить, що рівні елементи залишаються на своїх місцях
swap	Міняє місцями два елементи
swap_ranges	Міняє місцями елементи заданого діапазону
transform	Застосовує функцію до елементів заданого діапазону, зберігаючи ре¬зультат в новій послідовності
unique і unique_copy	Видаляє дублікати із заданого діапазону.
upper_bound	Знаходить перший елемент послідовності, що не перевищує задане значення

Підрахунок

Однією з головних операцій, що застосовуються до послідовностей, є підрахунок їх елементів. Для цього використовуються функції **count()** або **count_if()** Їх загальний вигляд наведено нижче.

template <class InIter, class T>
prtdiff_t count(Inter start, InIter end, const T &val);
template <class InIter, class UnPred>
prtdiff_t count(InIter start, InIter end, UnPred pfn);

Тип **prtdiff_t** є різновидом типу **int**. Алгоритм **count()** повертає кількість елементів послідовності, що мають значення *val*, починаючи з позиції, заданої ітератором *start*, і закінчуючи позицією, заданої ітератором *end.* Алгоритм *count_if()* повертає кількість елементів послідовності, що задовільняють предикату *pfn*, починаючи з позиції, заданої ітератором *start*, і закінчуючи позицією, заданої ітератором *end*'.

Наступна програма демонструє застосування алгоритму **count ().**

```
// Презентація алгоритма cout ().
#include <vector>
#include <cstdlib>
#include <algorithm>
using namespace std;
int main()
{
	vector<bool> v;
	int i;
	for (i = 0; i < 10; i++)
		if (rand() % 2)  v.push_back(true); else v.push_back(false);

	for (i = 0; i < 10; i++){
	cout << boolalpha << v[i] << "   ";
		cout << endl;
	}
		i = count(v.begin(), v.end(), true);
		cout << "  парних " << i << "  елеманта.\n";
		system("pause");
		return 0;
	}
```

Ця програма виводить на екран наступні результати.

Послідовність:

true true false false true false false false false false

парних 3 елементи.

Спочатку програма створює вектор, що містить випадково згенеровані значення **true** і **false**. Потім для підрахунку значень **true** застосовується функція **count()**.

Наступна програма ілюструє роботу функції **count_if()**. Вона створює вектор, що містить числа від 1 до 19. Потім вона підраховує елементи, кратні 3. Для цього створюється унарний предикат **dividesBy3()**, який повертає значення **true**, якщо його аргумент кратний 3.

```
#include <iostream>
#include <vector>
#include "windows.h"
#include <algorithm>
using namespace std;
bool dividesBy3(int i)
{
	if ((i % 3) == 0) return true;

	return false;
}
int main()
{
	SetConsoleCP(1251);
	SetConsoleOutputCP(1251);

	vector<int> v;
```

```
	int i;
	for (i = 1; i < 20; i++) v.push_back(i);
	cout << "  Послідовність: \n";
	for (i = 0; i < v.size(); i++)
		cout << v[i] << "   ";
	cout << endl;
	i = count_if(v.begin(), v.end(), dividesBy3);
	cout << "  Кількість чисел, кратних 3 =" << i << "  .\n";

	system("pause");
	return 0;
}
```

Результати роботи програми наведені нижче.

Послідовність: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

Кількість чисел, кратних 3, = 6.

Зверніть увагу на те, як закодований унарний предикат **divideBy3**. Аргументами всіх унарних предикатів є об'єкти, що зберігаються у відповідному контейнері. Предикат повинен повертати значення **true** або **false**.

Видалення і заміна елементів

Іноді необхідно створити нову послідовність, що містить лише частину елементів вихідної послідовності. Для цього можна застосовувати шаблон алгоритму **remove_copy().** Його специфікація приведена нижче.

```
template <class InIter, class OutIter, class T>
 OutIter remove_copy(InIter start, InIter end,
          OutIter result, const T &val);
```

Алгоритм **remove_copy()** копіює елементи із заданого діапазону, видаляючи з нього елементи, що мають значення *val*. Він поміщає результат в послідовність, на яку посилається ітератор *result*, і повертає ітератор, установлений на її кінець. Розмір результуючого контейнера повинен бути достатньо великим, щоб у ньому помістився результат.

Щоб у процесі копіювання замінити один елемент послідовності іншим, слід застосовувати алгоритм **rep1ace_copy ()**. Його специфікація приведена нижче.

```
template <class InIter, class OutIter, class T>
 OutIter remove_copy(InIter start, InIter end,
       OutIter result, const T &old, const T &new);
```

Алгоритм **rep1ace_copy()** копіює елементи із заданого діапазону, замінюючи елемент *old* елементом *new*. Він поміщає результат в послідовність, на яку посилається ітератор *result*, і повертає ітератор, установлений на її кінець. Розмір результуючого контейнера повинен бути достатньо великим, щоб в ньому помістився результат.

Алгоритми **remove_copy()** і **rep1ace_copy()** ілюструються в наступній програмі. Вона створює послідовність символів, а потім видаляє з неї всі пробели, замінюючи їх двокрапками.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include "windows.h"

using namespace std;
int main() {
	SetConsoleCP(1251);
	SetConsoleOutputCP(1251);

	char str[] = "Шаблони STL мають велику силу";
	vector<char> v, v2(30);   int i;
```

```
	for (i = 0; str[i]; i++) v.push_back(str[i]);
	cout << "  Вихідна послідовність:\n";
	for (i = 0; i < v.size(); i++) cout << v[i];
	cout << endl;
	remove_copy(v.begin(), v.end(), v2.begin(), ' ');
	cout << "Результат після видалення пропусків: \n";

	for (i = 0; i < v2.size(); i++) cout << v2[i];
	cout << endl << endl;

	cout << "  Вихідна послідновність: \n";
	for (i = 0; i < v.size(); i++) cout << v[i];  cout << endl;
	replace_copy(v.begin(), v.end(), v2.begin(), ' ', ':');
	cout << "Результат після заміни пропусків двокрапками: \n";
	for (i = 0; i < v2.size(); i++) cout << v2[i];  cout << endl << endl;

	system("pause");
	return 0;
}
```

Результат роботи цієї програми наведено нижче.

Вихідна послідовність:
Бібліотека STL володіє великою силою.
Результат після видалення пропусків:
БібліотекаSTLволодієвеликоюсилою.
Вихідна послідовність:
Бібліотека STL володіє великою силою.
Результат після заміни пропусків двокрапками:
Бібліотека:STL:володіє:великою:силою.

Зміна порядку проходження елементів послідовності

Для зміни порядку проходження елементів послідовності на зворотний застосовується алгоритм **reverse()**. Його специфікація має наступний вигляд

template <class BiIter> void reverse(BiIter start, BiIter end);

Алгоритм **reverse()**. змінює порядок проходження елементів в діапазоні між ітераторами *start* і *end* на зворотний.

Проілюструємо алгоритм **reverse()** наступною програмою.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include "windows.h"

using namespace std;

int main()
{
	SetConsoleCP(1251);
	SetConsoleOutputCP(1251);

	vector<int> v;
	int i;

	for (i = 0; i < 10; i++) v.push_back(i);

	cout << "Вихідна послідовність: ";
	for (i = 0; i < v.size(); i++) cout << v[i] << " ";
	cout << endl;

	reverse(v.begin(), v.end());

	cout << " зворотня послідовність : ";
	for (i = 0; i < v.size(); i++) cout << v[i] << " ";
```

```
    cout << endl;
    system("pause");

    return 0;
}
```

Результати її роботи виглядають так.

Вихідна послідовність: 0 1 2 3 4 5 6 7 8 9

Зворотний послідовність: 9 8 7 6 5 4 3 2 1 0

Перетворення послідовності

Один з найцікавіших алгоритмів – **transform()** - модифікує кожен елемент заданого діапазону за допомогою функції, заданої програмістом. Алгоритм **transform()** має два різновиди.

```
template <class InIter, class OutIter, class Func,
  OutIter transform(InIter start, inIter end,
                    OutIter result, Func unaryfunc);
template <class InIter1, class InIter2,
        class OutIter, class Func,
  OutIter transform(InIter1 start1, inIter1 end1,
                    InIter2 result2, OutIter result,
                    Func binaryfunc);
```

Цей алгоритм застосовує функцію до всіх елементів заданого діапазону і зберігає результат в об'єкті *result*. У першому варіанті діапазон визначається ітераторами *start* і *end*, функція задається параметром unaryfunc. Вона отримує значення елементів свого параметра і повертає перетворену послідовність. У другому варіанті перетворення послідовності виконується функтором *binaryfunc*. Першим параметром цієї функції є значення елемента з цієї послідовності, а другим - елемент з другої послідовності. Обидві версії функції **transform(),** повертають ітератор, який посилається на кінець результуючої послідовності.

Наступна програма використовує для перетворення послідовності просту функцію **reciprocal()**, яка замінює числа, що входять в список, оберненими величинами. Зверніть увагу на те, що результуюча послідовність записується в вихідний список.

```
#include <iostream>
#include <list>
#include <algorithm>
#include "windows.h"

using namespace std;
double reciprocal(double i) {
    return 1.0 / i;
}
int main()
{
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);

    list<double> vals;
    list<double>::iterator p;
    int i;
    for (i = 1; i < 10; i++) vals.push_back((double)i);
    cout << "  Вихідний вміст списка vals: \n";
    p = vals.begin();
    while (p != vals.end()) {
        cout << *p << "   ";
        p++;
    }
    cout << endl;

    transform(vals.begin(), vals.end(), vals.begin(), reciprocal);
    cout << "  Відображення вміст списка vals: \n";
    p = vals.begin();
    while (p != vals.end()) {
```

```
        cout << *p << "   ";
        p++;
    }
    cout << endl;
    system("pause");
    return 0;
}
```

Результат роботи цієї програми показаний нижче.

Вихідний вміст списку vals: 1 2 3 4 5 6 7 8 9

Перетворений вміст списку vals: 1 0.5 0.33333 0.25 0.2 0.166667 0.142857 0.125 0.111111

Застосування функторів

. Нагадаємо, що функтор - це клас, в якому визначена операторна функція **operator ()**. Бібліотека STL містить велику кількість вбудованих функторів, наприклад, **less**, **minus** і ін. Вона дозволяє також визначати свої власні функтори.

Унарні і бінарні функтори

Подібно унарним і бінарним предикатам, існують унарні і бінарні функтори. Унарний функтор має один аргумент, а бінарний - два. Унарний і бінарний функтори не можуть заміняти один одного. Наприклад, якщо алгоритм використовує бінарний функтор, йому потрібно передавати саме бінарний, а не унарний функтор.

Застосування вбудованих функторів

В бібліотеці STL передбачений широкий вибір вбудованих функторів. Перечислимо бінарні функтори.

plus	**minus**	**multplies**	**divides**	**modulus**
equal_to	**not_equal_to**	**greater**	**greater_equal**	**less**
less_equal	**logical_and**	**logical_or**		

До унарним належать такі функтори.

logical_not **negate**

Функтори виконують операції, визначені їх іменами. Єдине ім'я, яке не є самоочевидним, - назва функтора **negate()**, що змінює знак свого аргументу.

Вбудовані функтори є шаблонними класами, в яких перевантажений оператор (), який повертає результат обраної операції. Наприклад, щоб застосувати бінарний функтор **plus()** до даних типу **float()**, слід використовувати наступну синтаксичну конструкцію.

plus<float> ()

Для виклику вбудованих функторів в програму необхідно включити заголовок <**functional**>.

Розглянемо простий приклад. Наступна програма використовує алгоритм **transform()**, описаний в попередньому розділі, і функтор **negative** для зміни знака значень, перерахованих у списку.

//Застосування унарного фунтора.

```
#include <iostream>
#include <list>
#include <functional>
#include <algorithm>
#include "windows.h"

using namespace std;
int main() {
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);

    list<double> vals;
```

```
    int i;
    for (i = 1; i < 10; i++) vals.push_back((double)i);
    cout << "  Вихідний зміст списка vals: \n";
    list<double>::iterator p = vals.begin();
    while (p != vals.end()) {
        cout << *p << "   ";
        p++;
    }
    cout << endl;
    transform(vals.begin(), vals.end(), vals.begin(), negate<double>());
    // Викликаєм функтор.
    cout << "  Змінений зміст списка  vals: \n";
    p = vals.begin();
    while (p != vals.end()) {
        cout << *p << "   ";
        p++;
    }
    cout << endl;
    system("pause");
    return 0;
}
```

Програма виводить на екран наступні результати.

Вихідне вміст списку vals:

1 2 3 4 5 6 7 8 9 0

Змінене вміст списку vals:

-1 -2 -3 -4 -5 -6 -7 -8 -9 -0

Зверніть увагу на те, як у цій програмі викликається функтор **negative()**. Оскільки список **vals** зберігає числа типу **double**, функтор **negative()** викликається з допомогою конструкції **negative<double>**. Алгоритм **transform()** автоматично застосовує функтор **negative()** до кожного елементу послідовності. Таким чином, єдиний параметр функтора **negative()** є елемент послідовності.

Наступна програма демонструє застосування бінарного функтора **divides()**. Вона створює два списки дійсних чисел і ділить значення одного списку на значення іншого. Для цього використовується бінарна форма алгоритму **transform()**.

```
#include <iostream>
#include <list>
#include <functional>
#include <algorithm>
#include "windows.h"

using namespace std;
int main() {
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);

    list<double>vals;
    list<double> divisors;
    int i;
    for (i = 10; i < 100; i += 10) vals.push_back((double)i);
    for (i = 1; i < 10; i++) divisors.push_back(3.0);
    cout << "вихідне значення списку vals:\n";
    list<double>::iterator p = vals.begin();
    while (p != vals.end()) {
        cout << *p << " ";
        p++;
    }

    cout << endl;
    transform(vals.begin(), vals.end(), divisors.begin(), vals.begin(),
divides<double>());
    // Викликаємо функтор.

    cout << "Змінений вміст списку vals:\n";
    p = vals.begin();
    while (p != vals.end()) {
```

```
        cout << *p << " ";
        p++;
    }
    cout << endl;
    system("pause");
    return 0;
}
```

Програма виводить на екран наступні результати.

Вихідний вміст списку vals: 10 20 30 40 50 60 70 80 90

Змінений вміст списку vals: 3.33333 6.66667 10 13.3333 16.6667 20 23.3333 26.6667 30

В даному випадку бінарний функтор **divides()** ділить елементи першої послідовності на відповідні елементи другої послідовності. Таким чином, функтор **divides()** отримує свої аргументи в наступному порядку.

divides(*first, second*);

Цей порядок можна узагальнити. Незалежно від застосовуваного бінарного функтора, його аргументи завжди перераховуються в зазначеному порядку *first, second.*

Створення функтора

Крім вбудованих функторів, можна створювати і застосовувати свої власні функтори. Для цього достатньо створити клас, в якому перевантажується операторна функція **operator()**. Однак, щоб досягти більшої гнучкості, для створення функторів слід використовувати один з наступних базових класів, визначених у класі STL.

```
template <class Argument, class Result> struct unary_function {
  typedef Argument argument_type;
  typedef Result result_type;
};

template <class Argument1, class Argument2, class Result>
struct binary_function {
  typedef Argument1 first_argument_type;
  typedef Argument2 second_argument_type;
  typedef Result result_type;
};
```

Ці шаблонні класи конкретизують імена узагальнених типів даних, що використовуються функтором. Вони вельми зручні, і практично завжди застосовуються для створення функторів.

Наступна програма демонструє узагальнений функтор. Вона перетворює функцію **reciprocal()**, використану при описі алгоритму **transform()**, в функтор.

```
// Створення функтора reciprocal
#include <iostream>
#include <list>
#include <functional>
#include <algorithm>
using namespace std;

//Звичайний функтор.
class reciprocal: unary_function<double, double> {
public:
    result_type operator() (argument_type i)
    {
       return (result_type) 1.0/i; //Повертаємо зворотнє число.
    }
};

int main()
```

```
{
   list<double> vals;
   int i;

   //Вводимо значення в список.
   for (i=1; i<10; i++) vals.push_back((double)i);

   cout << "вихідний вміст списку vals:\n";
   list<double>::iterator p = vals.begin();
   while (p != vals.end())  {
     cout << *p << " ";
     p++;
   }
cout << endl;

//Застосування функтора reciprocal.
p = transform (vals.begin(), vals.end(),
              vals.begin(),
              reciprocal());  //Виклик функтора.

cout << "Змінений вміст списку vals:\n";
p = vals.begin();
while (p != vals.end()) {
   cout << *p << " ";
   p++;
}
return 0;
}
```

Зверніть увагу на два важливих аспекти, пов'язаних з функтором **reciprocal()**. По-перше, він успадковує властивості базового класу **unary_function**, а також надає доступ до типів **argument_type** і **result_type**. По-друге, цей фактор визначає операторну функцію **operator()**, що повертає зворотне значення аргументу. Як правило, щоб створити функтор, достатньо створити клас, похідний від одного з базових класів, зазначених вище, і перевантажити оператор (). Це дійсно просто.

Застосування редакторів зв'язків

При виклику бінарного функтора один з його аргументів можна пов'язати з конкретним значенням. У багатьох ситуаціях ця властивість виявляється досить корисною. Припустимо, що з послідовності потрібно видалити всі елементи, що перевищують певне значення, наприклад, число 8. Для цього необхідно мати можливість пов'язувати число 8 з правим операндом функтора **greater()**. Інакше кажучи, бажано, щоб функтор **greater()** міг виконувати порівняння

val > 8

для кожного елемента послідовності. В бібліотеці STL існує механізм, що дозволяє це робити. Він називається *редактором зв'язків* (binder).

Існують два редактора зв'язків – **bind2nd()** і **bind1st()**, що мають наступний вигляд.

```
bind1st(binfunc_obj, value);
bind2st(binfunc_obj, value);
```

Тут параметр *binfunc_obj* є бінарним функтором. Редактор зв'язків **bind1st()** повертає унарний функтор, у якого лівий операнд функтора *binfunc_obj* пов'язаний із значенням *value*.Редактор зв'язків **bind2nd()** повертає унарний функтор, у котрого зі значенням *value* пов'язаний правий операнд функтора *binfunc_obj*. Цей редактор зв'язків використовується частіше. У кожному разі результатом роботи редактора зв'язків є унарний функтор, пов'язаний з певним значенням.

Продемонструємо роботу редактора зв'язків на прикладі алгоритму **remove_it()**. Він видаляє елементи з послідовності, аналізуючи результат предиката. Його прототип має наступний вигляд.

```
template<class ForIter, class Unpred>
  ForIter remove_it(ForIter start, ForIter end, UnPred func);
```

Алгоритм видаляє елементи з діапазону, визначеного итераторами *start* і *end*, якщо унарний предикат, заданий параметром *func*, є істинним. Алгоритм повертає покажчик на кінець нової послідовності, утвореної внаслідок видалення елементів.

Наступна програма видаляє з послідовності всі елементи, більші за число 8. Оскільки предикат, необхідний редактору зв'язків **remove()**, є унарним, ми не можемо просто викликати бінарний функтор **greater().** Натомість слід пов'язати число 8 з другим аргументом функтора **greater()**, використовуючи редактор зв'язків **bind2nd()**.

```
// Презентація редакторів связі  bind2nd().
#include <iostream>
#include <list>
#include <functional>
#include <algorithm>
using namespace std;
int main ()
{
 list<int> lst;
 list<int>::iterator p, endp;
int i;
for(i=1; i<20; i++) lst.push_back(i);
  cout << ":\n";
  p = lst.begin();
  while(p != lst.end()) {
    cout << *p << " ";
    p++;
  }
  cout << endl;

  endp = remove_if(lst.begin(), lst.end(),
                bind2nd(greate<int>(), 8));

  cout << ":\n";
  p = lst.begin();
  while(p != endp) {
     cout << *p << " ";
     p++;
  }

  return 0;
}
```

Результат роботи цієї програми наведений нижче.

```
Вихідна послідовність:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
Результуюча послідовність:
1 2 3 4 5 6 7 8
```

Поекспериментуйте з цією програмою, застосовуючи різні функтори і редактори зв'язків. Ви переконаєтеся, наскільки редактори зв'язків збільшують міць бібліотеки STL.

І останнє: в бібліотеці є об'єкт, тісно пов'язаний з редактором зв'язків. Він називається інвертором (negator) і повертає заперечення (тобто доповнення) предиката. Його загальний вигляд наведено нижче.

```
noy1(unary_predicate);
not2(binary_predicate);
```

Наприклад, якщо підставити рядок

```
endp = remove_if(lst.begin(), lst.end(),
        not1(bind2nd(greater<int>(), 8)));
```

у попередню програму, вона видалить з послідовності **1st** всі елементи, що не перевищують число 8.

Заключні зауваження про бібліотеку STL

Бібліотека STL є необ'ємною частиною мови C ++. Багато задач програмування можна і потрібно вирішувати з її допомогою. Ця бібліотека володіє більшою міццю, гнучкістю, і, хоча синтаксис шаблонів досить складний, її дуже легко використовувати. Жоден програміст на мові C ++ не має права нехтувати библиотекою, оскільки вона відіграє важливу роль в створенні нових програм.