Домашняя работа №2
Стандартная библиотека шаблонов STL

Оглавление

Цель, задачи и порядок выполнения домашней работы

Целью выполнения домашней работы является освоение технологии обобщенного программирования с использованием библиотеки стандартных шаблонов (STL) языка C++.

Основными задачами выполнения домашней работы являются:

1. Познакомиться со структурой STL.
2. Изучить основные контейнеры, итераторы, алгоритмы библиотеки STL.
3. Освоить порядок работы со стандартными контейнерами, итераторами и алгоритмами библиотеки STL.

Основное содержание работы:

- Написать три программы с использованием STL.
- Первая и вторая программы должны демонстрировать работу с контейнерами и итераторами STL.
- В третьей программе используются алгоритмы STL.
- Подготовить отчёт.

Содержание отчета:

1. Титульный лист.
2. Цели, задачи домашней работы.
3. Постановка задачи.
4. Определение пользовательского класса.
5. Определения используемых в программах компонентных функций для работы с контейнером, включая конструкторы.
6. Объяснение этих функций.
7. Перечень (с объяснениями) стандартных контейнеров STL, используемых в программах.
8. Перечень стандартных итераторов для контейнеров, используемых в программах.
9. Объяснение используемых в программах алгоритмов STL.
10. Результаты работы программ.

Общие сведения

Стандартная библиотека шаблонов STL (Standard Template Library) это часть стандартной библиотеки C++. Она предоставляет различные типобезопасные контейнеры для хранения коллекций связанных объектов. *Контейнеры* представляют собой шаблоны классов. При объявлении переменной контейнера указывается тип элементов, которые будет удерживаться контейнером. Контейнеры могут создаваться с использованием списков инициализаторов. Они имеют специальные методы для добавления и удаления элементов и выполнения других операций.

Итерация элементов в контейнере и доступ к отдельным элементам осуществляются с помощью *итераторов*. Можно использовать итераторы явно, используя их функции `begin()` и `end()`. Можно использовать их неявно, например, с помощью основанного на диапазоне выражения `for`. Итераторы для всех контейнеров стандартной библиотеки C++ имеют общий интерфейс, но каждый контейнер определяет собственные специализированные итераторы.

Алгоритмы выполняют операции над содержимым контейнера. Существуют алгоритмы для инициализации, сортировки, поиска, замены содержимого контейнеров и др.

Контейнеры

Контейнеры можно разделить на три основные категории:

1. последовательные контейнеры,
2. ассоциативные контейнеры
3. контейнеры-адаптеры.

Последовательные контейнеры

Последовательные контейнеры (или ещё «контейнеры последовательности») – это контейнерные классы, элементы которых находятся в последовательности. Их определяющей характеристикой является то, что вы можете вставить свой элемент в любое место контейнера.

Начиная с C++11 STL содержит 6 контейнеров последовательности:

1. **vector**: массив переменного размера.
   a. Поддерживает произвольный доступ к любому элементу в контейнере.
   b. Обеспечивает добавление и удаление элементов из любого места контейнера.
2. **deque**: двусторонняя очередь.
   a. Поддерживает произвольный доступ к любому элементу в контейнере.
   b. Обеспечивает удаление и добавление элементов в начале и в конце контейнера.
3. **list**: двусвязный список
   a. Поддерживает только последовательный двунаправленный доступ к элементам.
   b. Обеспечивает удаление и добавление элементов в начале и в конце контейнера.
4. **forward_list**: односвязный список.
   a. Поддерживает только однонаправленный последовательный доступ к элементам.
   b. Обеспечивает удаление и добавление элементов в начале и в конце контейнера.
5. **array**: массив фиксированного размера.
   a. Поддерживает произвольный доступ к любому элементу в контейнере.
   b. Добавлять или удалять элементы из контейнера нельзя.
6. **string**: представляет контейнер, аналогичный вектору, который состоит из символов, то есть строку.

Vector

Vector (или просто «**вектор**») – это динамический массив, способный увеличиваться по мере необходимости для содержания всех своих элементов. Класс `vector` обеспечивает произвольный доступ к своим элементам через оператор индексации [], а также поддерживает вставку и удаление элементов.

В следующей программе мы вставляем 5 целых чисел в вектор и с помощью перегруженного оператора индексации [] получаем к ним доступ для их последующего вывода:

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

int main()
{

    vector<int> vect;
    for (int count = 0; count < 5; ++count)
        vect.push_back(10 - count); // вставляем числа в конец массива

    for (int index = 0; index < vect.size(); ++index)
        cout << vect[index] << ' ';

    cout << '\n';
}
```

```
10 9 8 7 6
```

Рис. 1. Результат выполнения программы

Deque

Класс **deque** (или просто «**дек**») – это двусторонняя очередь, реализованная в виде динамического массива, который может расти с обоих концов.

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

int main()
{
   deque<int> deq;
    for (int count = 0; count < 4; ++count)
    {
        deq.push_back(count);  // вставляем числа в конец массива
        deq.push_front(10 - count); // вставляем числа в начало массива
    }

    for (int index = 0; index < deq.size(); ++index)
       cout << deq[index] << ' ';

    cout << '\n';
}
```

```
7 8 9 10 0 1 2 3
```

Рис. 2. Результат выполнения программы

List

List (или просто «**список**») – это двусвязный список, каждый элемент которого содержит 2 указателя: один указывает на следующий элемент списка, а второй – на предыдущий элемент списка. `list` предоставляет доступ только к началу и к концу списка – произвольный доступ запрещён. Если вы хотите найти значение где-то в середине, то вы должны начать с одного конца и перебирать каждый элемент списка до тех пор, пока не найдёте то, что ищете. Преимуществом двусвязного списка является то, что вставка элементов происходит очень быстро, если вы, конечно, знаете, куда хотите вставлять. Обычно для перебора элементов двусвязного списка используются итераторы

Итераторы для этого контейнера перемещаются последовательно операциями `++` и `--`.

Для иллюстрации работы с таким контейнером реализуем задачу нахождения всех простых чисел, не превосходящих N:

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

inline ostream& operator <<(ostream& out, const list<int>& obj) {
	const int line = 10;
	int k = 0;
	for (auto j = obj.begin(); j != obj.end(); j++) {
		cout << *j << "\t";
		if (0 == ++k % line) cout << endl;
	}
	if (k % line != 0) cout << endl;
	return out;
}

int main(int argc, char** argv) {
	int n = stoi(argv[1]);		// верхняя граница
	list<int> a;

	for (int i = 2; i < n; i++)
		a.push_back(i);

	list<int>::iterator k = a.begin();

	while (*k * *k <= n) {
		int div = *k++;
		auto j = k;
		while (j != a.end()) {
			auto i = j++;
			if (0 == *i % div) a.erase(i);
		}
	}
```

```
        cout << a;
}
```

Здесь первая половина кода (оператор << для списка) только декоративная реализация для диагностики и удобства, а в остальной части проделывается следующее:

- Формируется список `list<int>` натуральных чисел диапазона [2…N] – **строки 19 – 22**;
- Для каждого (оставшегося) числа из всего последующего списка **удаляются** кратные ему элементы – **строка 31**;
- И так до тех пор, пока очередное проверяемое число не превысит квадратный корень из N – **строка 26** (в теории чисел показано, что делимость можно проверять не до N-1, а до числа, превышающего корень квадратный N);

Из особенностей кода, и **контейнеров list** вообще, нужно обратить внимание на то, что после удаления элемента, указываемого текущим значением итератора, значение итератора становится неопределённым. Если нам нужно продолжать итерацию, то мы должны работать с **копией** итератора (**i** в показанном коде).

```
2       3       5       7       11      13      17      19      23      29
31      37      41      43      47      53      59      61      67      71
73      79      83      89      97      101     103     107     109     113
127     131     137     139     149     151     157     163     167     173
179     181     191     193     197     199     211     223     227     229
233     239     241     251     257     263     269     271     277     281
283     293
```

Рис. 3. Результат выполнения программы

Forward_list

Контейнер `forward_list` представляет односвязный список. Для его использования необходимо подключить заголовочный файл `forward_list`.

Примеры создания односвязного списка:

```
forward_list<int> list1;
// пустой список
forward_list<int> list2(5);
// list2 имеет 5 элементов, каждый из которых имеет значение по умолчанию
forward_list<int> list3(5, 2);
// list3 состоит из 5 чисел, каждое число равно 2
forward_list<int> list4{ 1, 2, 4, 5 };
// list4 состоит из чисел 1, 2, 4, 5
forward_list<int> list5 = { 1, 2, 3, 4, 5 };
// list5 состоит из чисел 1, 2, 3, 4, 5
forward_list<int> list6(list4);
// list6 - копия списка list4
forward_list<int> list7 = list4;
// list7 - копия списка list4
forward_list<int> list8({ 1, 2, 3, 4, 5, 6 });
// list8  состоит из чисел 1, 2, 3, 4, 5, 6
```

Получение элементов

Напрямую в списке `forward_list` можно получить только первый элемент.

Для этого применяется функция `front()`. Для перебора элементов также можно использовать цикл:

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

int main()
{
   forward_list<int> numbers = { 1, 2, 3, 4, 5 };

    int first = numbers.front();    // 1

    for (int n : numbers)
        cout << n << "\t";
    cout << std::endl;
    return 0;
}
```

```
1       2       3       4       5
```

Рис. 4. Результат выполнения программы

Также для перебора и получения элементов можно использовать итераторы. Причем класс `forward_list` добавляет ряд дополнительных функций для получения итераторов: `before  begin()` и `cbefore_begin()`. Обе функции

возвращают итератор (вторая функция возвращает константный итератор `const_iterator`) на несуществующий элемент непосредственно перед началом списка. К значению по этому итератору обратиться нельзя.

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

int main()
{
    forward_list<int> numbers = { 1, 2, 3, 4, 5 };

    auto prev = numbers.before_begin();

    auto current = numbers.begin();
    auto end = numbers.end();
    while (current != end)
    {
        cout << *current << "\t";
        current++;
    }
    cout << std::endl;

    return 0;
}
```

```
1       2       3       4       5
```

Рис. 5. Результат выполнения программы

Размер списка

По умолчанию класс `forward_list` не определяет никаких функций, которые позволяют получить размер контейнера. В этом классе только функция `max_size()`, которая позволяет получить максимальный размер контейнера.

Функция `empty()` позволяет узнать, пуст ли список. Если он пуст, то функция возвращает значение `true`, иначе возвращается значение `false`:

```
forward_list<int> numbers = { 1, 2, 3, 4, 5 };
if (numbers.empty())
     cout << "The forward_list is empty" << endl;
else
     cout << "The forward_list is not empty" << endl;
```

Для изменения размера контейнера можно использовать функцию `resize()`, которая имеет две формы:

- `resize(n)`: оставляет в списке n первых элементов. Если список содержит больше элементов, то он усекается до первых n элементов.

Если размер списка меньше n, то добавляются недостающие элементы и инициализируются значением по умолчанию

- `resize(n, value)`: также оставляет в списке n первых элементов. Если размер списка меньше n, то добавляются недостающие элементы со значением `value`.

Пример:

```
forward_list<int> numbers = { 1, 2, 3, 4, 5, 6 };
numbers.resize(4);
// оставляем первые четыре элемента - numbers = {1, 2, 3, 4}
numbers.resize(6, 8);
// numbers = {1, 2, 3, 4, 8, 8}
```

Изменение элементов списка

Функция `assign()` позволяет заменить все элементы списка определенным набором. Она имеет следующие формы:

- `assign(il)`: заменяет содержимое контейнера элементами из списка инициализации il
- `assign(n, value)`: заменяет содержимое контейнера n элементами, которые имеют значение value
- `assign(begin, end)`: заменяет содержимое контейнера элементами из диапазона, на начало и конец которого указывают итераторы `begin` и `end`.

Применение функции:

```
forward_list<int> numbers = { 1, 2, 3, 4, 5 };
numbers.assign({ 21, 22, 23, 24, 25 });
// numbers = { 21, 22, 23, 24, 25 }
numbers.assign(4, 3);
// numbers = {3, 3, 3, 3}
list<int> values = { 6, 7, 8, 9, 10, 11 };
auto start = ++values.begin();
// итератор указывает на второй элемент из values
auto end = values.end();
numbers.assign(start, end);
// numbers = { 7, 8, 9, 10, 11 }
```

Функция `swap()` обменивает значениями два списка:

```
forward_list<int> list1 = { 1, 2, 3, 4, 5 };
forward_list<int> list2 = { 6, 7, 8, 9 };
list1.swap(list2);
// list1 = { 6, 7, 8, 9};
// list2 = { 1, 2, 3, 4, 5 };
```

Добавление элементов

Для добавления элементов в `forward_list` применяются следующие функции:

- `push_front(val)`: добавляет объект val в начало списка
- `emplace_front(val)`: добавляет объект val в начало списка
- `emplace_after(p, val)`: вставляет объект val после элемента, на который указывает итератор p. Возвращает итератор на вставленный элемент. Если p представляет итератор на позицию после конца списка, то результат не определен.
- `insert_after(p, val)`: вставляет объект val после элемента, на который указывает итератор p. Возвращает итератор на вставленный элемент.
- `insert_after(p, n, val)`: вставляет n объектов val после элемента, на который указывает итератор p. Возвращает итератор на последний вставленный элемент.
- `insert_after(p, begin, end)`: вставляет после элемента, на который указывает итератор p, набор объектов из другого контейнера, начало и конец которого определяется итераторами begin и end. Возвращает итератор на последний вставленный элемент.
- `insert_after(p, il)`: вставляет после элемента, на который указывает итератор p, список инициализации il. Возвращает итератор на последний вставленный элемент.

Применение функций:

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

int main()
{
    forward_list<int> numbers = { 7, 8 };
    numbers.push_front(6);
    // добавляем в начало число 6
    // numbers =  { 6, 7, 8 }
    numbers.emplace_front(-3);
    // добавляем в начало число -
    // numbers =  { -3, 6, 7, 8 }
    auto iter = numbers.begin();
    iter = numbers.emplace_after(iter, -2);
    // добавляем после итератора число -2
    // numbers =  { -3, -2, 6, 7, 8 }
    iter = numbers.insert_after(iter, -1);
    // numbers =  { -3, -2, -1, 6, 7, 8 }
    iter = numbers.insert_after(iter, 3, 0);
    // добавляем три нуля
    // numbers =  { -3, -2, -1, 0, 0, 0, 6, 7, 8 }
    list<int> values = { 1, 2, 3 };
```

```
        iter = numbers.insert_after(iter, values.begin(), values.end());
        // добавляем все элементы из values
        // numbers = { -3, -2, -1, 0, 0, 0, 1, 2, 3, 6, 7, 8 }
        numbers.insert_after(iter, { 4, 5 });
        // добавляем список { 4, 5 }
        // numbers = { -3, -2, -1, 0, 0, 0, 1, 2, 3, 4, 5, 6, 7, 8 }
        for (int n : numbers)
            cout << n << "\t";
        cout << endl;
        return 0;
    }
```

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

Рис. 7. Результат выполнения программы

Удаление элементов

Чтобы удалить элемент из контейнера `forward_list` можно использовать следующие функции:

- `clear()`: удаляет все элементы
- `pop_front()`: удаляет первый элемент
- `erase_after(p)`: удаляет элемент после элемента, на который указывает итератор p. Возвращает итератор на элемент после удаленного
- `erase_after(begin, end)`: удаляет диапазон элементов, на начало и конец которого указывают соответственно итераторы `begin` и `end`. Возвращает итератор на элемент после последнего удаленного

Использование функций:

```
forward_list<int> numbers = { 1, 2, 3, 4, 5, 6, 7 };
numbers.pop_front();
// numbers = { 2, 3, 4, 5, 6, 7};
auto iter = numbers.erase_after(numbers.begin());
// numbers = { 2, 4, 5, 6, 7 };
// iter указывает на элемент 4

numbers.erase_after(iter, numbers.end());
// numbers = { 2, 4 };
```

Array

`array` наиболее близок к С-массивам. Он хранит фиксированное число элементов, но поддерживает и дополнительный набор методов, в том числе и универсальных для всех контейнеров. Преимущество данного контейнера – максимально быстрый доступ к своих элементам.

Для работы нужно подключить заголовочный файл `<array>`.

Объявление

```
array<int, 10> mass1;
```

В этом примере объявлен массив `mass1`, тип `int`, размером в 10 элементов. Поскольку в `array` количество элементов фиксировано, то эта величина не может быть переменной величиной.

Инициализация итераторов

Итераторы диапазона можно получить используя функции `begin()` и `end()`, но теперь как методы класса. Вместо указания конкретного типа можно использовать спецификатор `auto`.

```
auto first = mass1.begin();
auto last = mass1.end();
```

Инициализация массива

Списочная инициализация:

```
array<int, 10> mass2 = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 };
```

Списочную инициализацию можно использовать не только при объявлении. В таком случае, объявленный ранее массив, получит новые элементы. Заполнение определенным значением. Функция `fill()`

```
array<int, 10> mass3;
mass3.fill(2);
```

Также доступна функция `swap()`.

Присваивание по имени элементов одного массива – другому.

Количество элементов и тип данных должны быть равными.

```
mass3 = mass2;
```

Ввод и вывод элементов массива, используя традиционный подход с помощью оператора `[]`. Для определения размера массива используется метод `size()`. Для доступа к элементам по индексу можно также использовать метод `at` (т. е. вместо `mas[i]`, `mas.at(i)`), который осуществляет контроль выхода за границы массива. `at(i)` генерирует исключение *out_of_range*, если `i > mas.size()`.

```
#include <iostream>
#include <array>
#include <ctime>
#include <cstdlib>
using namespace std;

int main() {
    array<int, 20> mas;
    srand(time(0));
    for (auto i = 0; i < mas.size(); i++)
        mas[i] = 10 + rand() % 90;
    for (auto i = 0; i < mas.size(); i++)
```

```
        cout << mas[i] << " ";

    return 0;
}
```

```
61 83 87 69 87 42 50 92 13 43 42 42 56 29 55 39 67 63 54 57
```

Рис. 8. Результат выполнения программы

Ввод и вывод элементов с использованием итераторов. (Работает быстрее, так как не пересчитываются индексы, как в предыдущем примере). Необходимо подключить заголовок <iterator>.

```
#include <iostream>
#include <array>
#include <iterator>
#include <ctime>
#include <cstdlib>
using namespace std;

int main() {
    array<int, 20> mas;
    auto first = mas.begin();
    auto last = mas.end();
    srand(time(0));
    while (first != last) {
        *first = 10 + rand() % 90;
        first++;
    }
    first = mas.begin();
    while (first != last) {
        cout << *first << " ";
        first++;
    }
    return 0;
}
```

```
23 16 31 38 95 38 28 23 68 44 79 19 42 30 37 87 82 54 43 41
```

Рис. 9. Результат выполнения программы

Цикл for, основанный на диапазоне. В примере `ar` – переменная цикла, которой передаются значения элементов массива `mas`.

```
#include <iostream>
#include <array>
#include <ctime>
#include <cstdlib>
using namespace std;

int main() {
    array<int, 20> mas;
```

```
    srand(time(0));
    for (int& ar : mas)
        ar = 10 + rand() % 90;
    for (int& ar : mas)
        cout << ar << " ";

    return 0;
}
```

```
23 94 61 12 73 50 83 73 70 20 44 12 36 44 90 80 62 63 88 16
```

Рис. 10. Результат выполнения программы

Операции сравнения.

Для `array` определены операции сравнения ==, !=, <, <=, >, >=

return mas1 > mas2 ? true : false;

Метод `empty` вернет булевский результат проверки на предмет наличия элементов в контейнере (т. е. является ли `begin() == end()`)

String

`String` – это класс STL, основанный на шаблонах, который входит в стандартную библиотеку `C++`. По сути , он и является контейнером так, как Шаблон std::basic_string удовлетворяет всем требованиям концепции контейнера. Для использования данного нужно подключить директиву `<string>`.

Рассмотрим простейший пример использования:

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

int main()
{
    string str;
    cout << "Enter your name: ";
    getline(cin, str);
    cout << "Hello, " << str << "!!! \n";
    return 0;
}:
```

```
Enter your name: Example4U
Hello, Example4U!!!
```

Рис. 11. Результат выполнения программы

Здесь создается экземпляр класса под именем `str`, далее пользователя просят представиться, при помощи `getline` данные записываются из потока `cin` непосредственно в нашу переменную `str`.

Чтобы узнать длину строки, можно воспользоваться функцией-членом `size()`, или `length()`, которые, возвращают саму длину (длина не включает завершающий нулевой символ).

```
str.size();
str.length();
```

Чтобы узнать, пуста ли строка используется специальный метод `empty()`, возвращающий true для пустой строки и `false` для непустой:

```
str.empty();
```

Чтобы узнать, совпадают ли строки, используется оператор сравнения ==:

```
if (str == str2)
```

Чтобы скопировать одну строку в другую достаточно использовать обычную операции присваивания =:

```
string str2; // Создание объекта строки
```

str2 = str; // при помощи оператора присваивания, копируем str в str2

Существует другой вариант, но у него есть своя особенность

```
string str2 = str;
```

Эта запись эквивалентна предыдущей, за исключением того, что в предыдущем коде мы использовали оператор присваивания, а этом случае работает конструктор копирования.

Для конкатенации строк используется операция сложения + или операция сложения с присваиванием +=.

Рассмотрим следующий код:

```
string str1 = "Hello";
string str2 = "World";
```

Мы можем получить строку `"HelloWorld"`, состоящую из конкатенации `str1` и `str2`, таким образом:

```
string str3 = str1 + str2;
```

В результате в строке `str3` будет хранится следующее: `"HelloWorld"`.

Так же подобного результата, мы можем добиться и другим способом:

```
string str3 = str1;
str3 += str2;
```

Здесь мы создаем строку `str3` инициализируя её содержимым `str1`, и далее при помощи оператора += в конец `str3` добавляем `str2`. Результат будет таким, как и в предыдущем примере.

Операция сложения может конкатенировать (соединять, присоединять) экземпляры (объекты) класса `string` не только между собой, но и со строками встроенного типа. Такой код будет вполне работоспособным:

```
char* str = "Hello";
string str2 = str;
str2 += " World";
str2 += '!';
```

В результате в строке `str2`, будет хранится `"HelloWorld!"`

Ещё одна полезная функция это: `c_str()`.

Функция `c_str()` возвращает указатель на символьный массив, который содержит строку объекта string в том виде, в котором она размещалась бы, во встроенном строковом типе. Например:

```
string str;
const char* str2 = str.c_str();
```

Чтобы обращаться к отдельным символам строки типа `string`, можно воспользоваться операцией взятия индекса ([]), или при помощи метода `at()`. Например:

```
string str = "Hello World";
cout << str[7] << str[0] << endl; // На консоли увидим: oH
```

Метод `at()` предлагает похожую схему доступа, за исключением того, что индекс предоставляется как аргумент функции:

```
string str = "Hello World";
cout << str.at(7) << str.at(0) << endl; // На консоли увидим: oH
```

В отличии от оператора [], метод `at()`, обеспечивает проверку границ и генерирует исключение `out_of_range`, если вы пытаетесь получить несуществующий элемент.

Так же в классе `string`, имеется функция, которая возвращает строку, являющуюся подстрокой исходной строки, начиная с позиции `pos` и включая n символов, или до конца строки.

```
str.substr(pos, n);
```

Ассоциативные контейнеры

Ассоциативные контейнеры – это контейнерные классы, которые автоматически сортируют все свои элементы (в том числе и те, которые вставляете Вы). По умолчанию ассоциативные контейнеры выполняют сортировку элементов, используя оператор сравнения <.

1. **set** – это контейнер, в котором хранятся только уникальные элементы, повторения запрещены. Элементы сортируются в соответствие с их значениями.
2. **multiset** – это `set`, но в котором допускаются повторяющиеся элементы.
3. **map** (или ещё «ассоциативный массив») – это `set`, в котором каждый элемент является парой «ключ-значение». Ключ используется для сортировки и индексации данных и должен быть уникальным. А значение – это фактические данные.
4. **multimap** (или ещё «словарь») – это `map`, который допускает дублирование ключей. Все ключи отсортированы в порядке возрастания, и вы можете посмотреть значение по ключу.

Set и multiset

Контейнер `set` реализует такие сущности как множество и мультимножество. По сути, это контейнеры, которые содержат некоторое количество отсортированных элементов. Отличие множества и мультимножества состоит в том, что `set` может хранить только уникальные элементы, когда `multiset` позволяет хранить и совпадающие ключи. Далее следует пример работы с множеством

```
#include <iostream>
#include <set> // заголовочный файл множеств и мультимножеств
#include <iterator>

using namespace std;
int main()
{
    set<char> mySet; // объявили пустое множество

    // добавляем элементы в множество
    mySet.insert('I');
    mySet.insert('n');
    mySet.insert('f');
    mySet.insert('i');
    mySet.insert('n');
    mySet.insert('i');
    mySet.insert('t');
    mySet.insert('y');

    copy( mySet.begin(), mySet.end(), ostream_iterator<char>(cout, " "));
    return 0;
}
```

В данном случае на вывод поступит последовательность `Ifinty`, т.к. использовалось обычное множество. При повторном вводе ключа дублирования не происходит. В случае же, если бы использовалось мультимножество, на вывод поступила бы последовательность `Ifiinnty`, т.к. мультимножество допускает повторение ключей. В этом примере также хорошо заметно, что множества сортируются автоматически, из-за чего необходимо грамотно продумывать работу с ними.

Например, автосортировка не позволяет менять какой-либо из ключей (в таком случае потребовалась бы перестройка множества). Поэтому операция изменения элемента представляет собой совокупность операций удаления и ввода нового элемента. Далее следует пример изменения элемента:

```
#include <iostream>
#include <set>
```

```
// заголовочный файл множеств и мультимножеств
#include <iterator>
#include <cstdlib>

using namespace std;
int main()
{
    srand(time(NULL));
    set<int> mySet; // объявили пустое множество

    // добавляем элементы в множество
    for( int i = 0; i < 10; i++) {
        mySet.insert( rand() % 100 );
    }

    cout << "Элементы множества: ";
    copy( mySet.begin(), mySet.end(),
ostream_iterator<int>(cout, " "));

    int del = 0;
    cout << "\nКакой элемент удалить? ";
    cin >> del;

    cout <<"Элемент " << *mySet.find(del) << " - удален!" <<
endl;
    mySet.erase(del);

    int add = 0;
    cout << "Какой элемент добавить? ";
    cin >> add;

    cout << "Новый элемент добавлен на место старого - " <<
*mySet.lower_bound(add) << endl;
    mySet.insert(add);

    cout << "Мы удалили элемент " << del << " и добавили "
<< add << ".\nВот что получилось: " << endl;
    copy( mySet.begin(), mySet.end(),
ostream_iterator<int>(cout, " "));

    return 0;
}
```

Map и multimap

Ассоциативным массив – массив, индекс которого может относиться к произвольному типу (не обязательно числовому).

Набор пар «ключ/значение» с уникальными ключами (контейнер `map`) можно рассматривать как ассоциативный массив.

При работе с отображениями операция вставки может осуществляться оператором индексирования []:

Индекс используется в качестве ключа и может иметь произвольный тип.

Оператор индексирования в данном случае работает не так, как обычный оператор индексирования массивов. Отсутствие элемента, связанного с индексом, не является ошибкой. При появлении нового индекса (или ключа) создается и вставляется в контейнер новый элемент, ключом которого является указанный индекс. Таким образом, в ассоциативных массивах индекс в принципе не может принимать недопустимое значение. Следовательно, в приведенном коде создаются новые элементы. А оператор присваивания связывает ключи со значениями, преобразованными к типу `float`.

Примеры работы с отображениями представлены ниже:

```
using namespace std;
int main() {
    // Тип коллекции
    typedef map<int, string> IntStringMMap;
    IntStringMMap coll;
    // Контейнер для хранения пар int/string
    // Вставка элементов в произвольном порядке
    // - значение с ключом 1 вставляется дважды.
    coll.insert(make_pair(5,"tagged"));
    coll.insert(make_pair(2,"a"));
    coll.insert(make_pair(1,"this"));
    coll.insert(make_pair(4,"of"));
    coll.insert(make_pair(6,"strings"));
    coll.insert(make_pair(1,"is"));
    coll.insert(make_pair(3,"multimap"));
    // Вывод содержимого контейнера
    // - перебор всех элементов
    // - переменная second содержит значение.
    IntStringMMap::iterator pos;
    for (pos = coll.begin(); pos != coll.end(); ++pos) {
        cout << pos->second << ' ';
    }
}
```

Далее следует пример работы с мультиотображениями:

```
using namespace std;
int main() {
    // Тип коллекции
    typedef multimap<int, string> IntStringMMap;
    IntStringMMap coll;
    // Контейнер для хранения пар int/string
    // Вставка элементов в произвольном порядке
    // - значение с ключом 1 вставляется дважды.
    coll.insert(make_pair(5,"tagged"));
    coll.insert(make_pair(2,"a"));
```

```
    coll.insert(make_pair(1,"this"));
    coll.insert(make_pair(4,"of"));
    coll.insert(make_pair(6,"strings"));
    coll.insert(make_pair(1,"is"));
    coll.insert(make_pair(3,"multimap"));
    // Вывод содержимого контейнера
    // - перебор всех элементов
    // - переменная second содержит значение.
    IntStringMMap::iterator pos;
    for (pos = coll.begin(); pos != coll.end(); ++pos) {
        cout << pos->second << ' ';
    }
}
```

Отличие мультиотображения и отображения сходно с отличием множества и мультимножества. Отображение не допускает дублирование ключей, тогда как мультиотображение разрешает работу с повторяющимися ключами.

Неупорядоченные контейнеры.

В некоторых задачах автоматическая сортировка может рассматриваться как существенный недостаток, если накладные расходы по упорядочиванию элементов будут высоки. В этом случае заменой классов упорядоченных множеств и словарей могут стать классы неупорядоченных ассоциативных контейнеров (далее - неупорядоченные контейнеры) `unordered_set`, `unordered_multiset`, `unordered_map` и `unordered_multimap`. Операции поиска, вставки и удаления в неупорядоченных контейнерах имеют среднюю постоянную временную сложность (O(1)). Данные контейнеры вошли в стандарт начиная с C++11.

Неупорядоченные контейнеры реализованы в виде **хеш-таблиц**. Выполнение операции в хеш-таблице начинается с вычисления **хеш-функции** (хеширования) от ключа. Получающееся хеш-значение (также «хеш», «хеш-код») играет роль индекса в массиве. Хеш-таблица похожа на обычный словарь, в котором литера может считаться хэш-значением.

Хеш-таблицы неупорядоченных контейнеров представляют собой массив ячеек, которые могут содержать неограниченное количество элементов. Такую ячейку мы будем называть сегментом. Фактически сегмент (с которым связан хеш) является связным списком.

Для начала работы с классами `unordered_set` и `unordered_multiset` необходимо подключить заголовок `unordered_set`

Типы неупорядоченных контейнеров включают в себя следующие шаблонные параметры:

- `const Key` — тип ключа (`unordered_set`, `unordered_multiset`, `unordered_map`, `unordered_multimap`)

- `T` – тип значения (`unordered_map`, `unordered_multimap`)
- `alloc` – функция аллокатора (необязательно)
- `size_type bucket_count` – минимальное количество сегментов для использования при инициализации (если не указан, определяется реализацией значением по умолчанию)
- `hash` – хэш-функция (не обязательно)
- `equal` – Функция сравнения, используемая для сравнения ключей (необязательно)

Неупорядоченные контейнеры и их упорядоченные аналоги имеют общие методы за исключением следующих, которые не поддерживаются:

- Операции сравнения >, <, >=, <= (поддерживаются только == и !=)
- `lower_bound` и `upper_bound`
- Реверсивные итераторы: `rbegin`, `crbegin`, `rend`, `crend`

Методы неупорядоченных контейнеров

- `load_factor()` – возвращает коэффициент заполнения (см. выше)
- `max_load_factor()` – если используется без аргумента, то возвращает тип `float` коэффициент максимального заполнения, если использует в качестве аргумента тип `float`, то устанавливает коэффициент максимального заполнения. Контейнер может автоматически увеличить количество сегментов, если коэффициент заполнения превышает этот порог.
- `rehash(size_type count)` – производит рехешинг контейнера так, чтобы количество сегментов (`backet_count`) было `backet_count` >=`count` и `backet_count > size/max_load_factor`
- `reserve(size_type count)` – устанавливает количество сегментов к такому числу, которое необходимо для размещения по крайней мере count элементов без превышения коэффициента максимального заполнения и проведения рехешинга контейнера.

Контейнеры-адаптеры

Адаптеры – это специальные предопределённые контейнерные классы, которые адаптированы для выполнения конкретных заданий. Самое интересное заключается в том, что Вы сами можете выбрать, какой последовательный контейнер должен использовать адаптер.

1. **stack** (стек) – это контейнерный класс, элементы которого работают по принципу LIFO («Last In, First Out» = «Последним Пришёл, Первым Ушёл»). Обычно в стеках используется deque в качестве последовательного контейнера по умолчанию.

2. **queue** (очередь) – это контейнерный класс, элементы которого работают по принципу FIFO («First In, First Out» = «Первым Пришёл, Первым Ушёл»). По умолчанию в очереди используется deque в качестве последовательного контейнера, но также может использоваться и list.

3. **priority_queue** (очередь с приоритетом) – это тип очереди, в которой все элементы отсортированы (с помощью оператора сравнения <). При вставке элемента он автоматически сортируется. Элемент с наивысшим приоритетом (самый большой элемент) находится в самом начале очереди с приоритетом, также, как и удаление элементов – выполняется с самого начала очереди с приоритетом.

Выбор контейнера для задачи

Для того, чтобы выбрать контейнер для конкретной задачи, необходимо ответить на некоторые вопросы:

1. Нужна ли возможность вставки нового элемента в произвольной позиции контейнера? Если нужна, выбирайте последовательный контейнер; ассоциативные контейнеры не подходят.
2. Важен ли порядок хранения элементов в контейнере? Если порядок следования элементов не важен, хэшированные контейнеры попадают в число возможных кандидатов. В противном случае придется обойтись без них.
3. Должен ли контейнер входить в число стандартных контейнеров C++? Если выбор ограничивается стандартными контейнерами, то хэшированные контейнеры, `slist` и `rope`, исключаются.
4. К какой категории должны относиться итераторы? С технической точки зрения итераторы произвольного доступа ограничивают ваш выбор контейнерами `vector`, `deque` и `string`, хотя, в принципе, можно рассмотреть и возможность применения `rope` (структура данных для хранения строки, представляющая из себя двоичное сбалансированное дерево и позволяющая делать операции вставки, удаления и конкатенации). Если нужны двусторонние итераторы, исключается класс `slist` и одна распространенная реализация хэшированных контейнеров.
5. Нужно ли предотвратить перемещение существующих элементов при вставке или удалении? Если нужно, воздержитесь от использования блоковых контейнеров
6. Должна ли структура памяти контейнера соответствовать правилам языка C? Если должна, остается лишь использовать `vector`

7. Насколько критична скорость поиска? Если скорость поиска критична, рассмотрите хэшированные контейнеры, сортированные векторы и стандартные ассоциативные контейнеры – вероятно, именно в таком порядке.

8. Может ли в контейнере использоваться подсчет ссылок? Если подсчет ссылок вас не устраивает, держитесь подальше от `string`, поскольку многие реализации `string` построены на этом механизме. Также следует избегать контейнера `rope`. Конечно, средства для представления строк вам все же понадобятся – попробуйте использовать `vector<char>`.

9. Потребуется ли транзакционная семантика для операций вставки и удаления? Иначе говоря, хотите ли вы обеспечить надежную отмену вставок и удалений? Если хотите, вам понадобится узловой контейнер. При использовании транзакционной семантики для многоэлементных вставок следует выбрать `list` – единственный стандартный контейнер, обладающий этим свойством. Транзакционная семантика особенно важна при написании кода, безопасного по отношению к исключениям. Вообще говоря, транзакционная семантика реализуется и для блоковых контейнеров, но за это приходится расплачиваться быстродействием и усложнением кода.

10. Нужно ли свести к минимуму количество недействительных итераторов, указателей и ссылок? Если нужно – выбирайте узловые контейнеры, поскольку в них операции вставки и удаления никогда не приводят к появлению недействительных итераторов, указателей и ссылок (если они не относятся к удаляемым элементам). В общем случае операции вставки и удаления в блоковых контейнерах могут привести к тому, что все итераторы, указатели и ссылки станут недействительными.

11. Не подойдет ли вам последовательный контейнер с итераторами произвольного доступа, в котором указатели и ссылки на данные всегда остаются действительными, если из контейнера ничего не удаляется, а вставка производится только в конце? Ситуация весьма специфическая, но, если вы с ней столкнетесь – выбирайте `deque`. Следует заметить, что итераторы `deque` могут стать недействительными, даже если вставка производится только в конце контейнера. Это единственный стандартный контейнер STL, у которого итераторы могут стать недействительными при действительных указателях и ссылках.

Вряд ли эти вопросы полностью исчерпывают тему. Например, в них не учитывается тот факт, что разные типы контейнеров используют разные стратегии выделения памяти (некоторые аспекты этих стратегий описаны в советах 10 и 14). Но и этот список наглядно показывает, что алгоритмическая сложность выполняемых операций – далеко не единственный критерий выбора. Бесспорно, она играет важную роль, но приходится учитывать и другие факторы.

Итераторы

Итератор – это объект, который способен перебирать элементы **контейнерного класса** без необходимости пользователю знать реализацию определённого контейнерного класса. Во многих контейнерах (особенно в списке и в ассоциативных контейнерах) итераторы являются основным способом доступа к элементам этих контейнеров.

Существует пять типов итераторов:

1. **Итераторы ввода** (`input_iterator`) поддерживают операции равенства, разыменования и инкремента.

`==, !=, *i, ++i, i++, *i++`

2. Итераторы вывода (`output_iterator`) поддерживают операции разыменования, допустимые только с левой стороны присваивания, и инкремента.

`++i, i++, *i=t, *i++=t`

3. Однонаправленные итераторы (`forward_iterator`) поддерживают все операции итераторов ввода/вывода и, кроме того, позволяют без ограничения применять присваивание.

`==, !=, =, *i, ++i, i++, *i++`

4. Двунаправленные итераторы (`biderectional_iterator`) обладают всеми свойствами forward-итераторов, а также имеют дополнительную операцию декремента (`--i, i--, *i--`), что позволяет им проходить контейнер в обоих направлениях.

5. Итераторы произвольного доступа (`random_access_iterator`) обладают всеми свойствами `biderectional`-итераторов, а также поддерживают операции сравнения и адресной арифметики, то есть непосредственный доступ по индексу.

`i+=n, i+n, i-=n, i-n, i1-i2, i[n], i1<i2, i1<=i2, i1>i2, i1>=i2`

В STL также поддерживаются **обратные итераторы** (`reverse iterators`). Обратными итераторами могут быть либо двунаправленные итераторы, либо итераторы произвольного доступа, но проходящие последовательность в обратном направлении.

Вдобавок к контейнерам, алгоритмам и итераторам в STL поддерживается ещё несколько стандартных компонентов. Главными среди них являются **распределители памяти**, **предикаты** и **функции сравнения**.

У каждого контейнера имеется определенный для него распределитель памяти (**allocator**), который управляет процессом выделения памяти для контейнера.

По умолчанию распределителем памяти является объект класса **allocator**. Можно определить собственный распределитель.

В некоторых алгоритмах и контейнерах используется функция особого типа, называемая **предикатом**. Предикат может быть унарным и бинарным. Возвращаемое значение: истина либо ложь. Точные условия получения того или иного значения определяются программистом. Тип унарных предикатов **UnPred**, бинарных – **BinPred**. Тип аргументов соответствует типу хранящихся в контейнере объектов.

Определен специальный тип бинарного предиката для сравнения двух элементов. Он называется **функцией сравнения** (comparison function). Функция возвращает истину, если первый элемент меньше второго. Типом функции является тип **Comp**.

Особую роль в STL играют **объекты-функции**.

Объекты-функции – это экземпляры класса, в котором определена операция «круглые скобки» (). В ряде случаев удобно заменить функцию на объект-функцию. Когда объект-функция используется в качестве функции, то для ее вызова используется operator ().

Пример:

```
class less {
public:
     bool operator()(int x,int y) {
          return x<y;
     }
};
```

Заголовочные файлы для итераторов

Заголовочные файлы стандартной библиотеки C++ не имеют расширения «.h». Стандартная библиотека C++ содержит последние расширения C++ стандарта ANSI (включая библиотеку стандартных шаблонов и новую библиотеку `iostream`). Она представляет собой набор файлов заголовков. В новых файлах заголовков отсутствует расширение H.

Начало работы

Для создания итератора мы должны с самого начала программы подключить библиотеку `<iterator>`

Далее для его создании нам потребуется использовать вот эту схему:

```
<контейнер> <его тип> :: iterator <имя итератора>;
```

- <контейнер> — указываем требуемый контейнер, на который и будет ссылаться итератор.

Например `map, vector, list`.

- <его тип> — указываем тип контейнера.

Если вы создали итератор и случайно ввели не тот тип данных, который указали при создании контейнера, то ваша программа будет работать неправильно и вообще может сломаться.

Также при инициализации итератора мы можем с самого начала написать, куда он будет указывать:

```
vector <int> i_am_vector;
vector <int> :: iterator = i_am_vector.begin();
```

Методы начала и конца контейнеров

У каждого контейнера имеются два метода, которые, как указатели передают итератору начало или конец контейнера – `begin()` и `end()`.

- Метод `begin()` отправит итератор на начала контейнера.
- А метод `end()` отправит на конец. А если точнее, то на одну ячейку больше последней.

Функционал итераторов

Об итераторе можно думать, как об указателе на определённый элемент контейнерного класса с дополнительным набором перегруженных операторов для выполнения чётко определённых функций:

- Оператор * возвращает элемент, на который в данный момент указывает итератор.
- Оператор ++ перемещает итератор к следующему элементу контейнера. Большинство итераторов также предоставляют оператор −− для перехода к предыдущему элементу.
- Операторы == и != используются для определения того, указывают ли два итератора на один и тот же элемент или нет. Для сравнения значений, на которые указывают два итератора, нужно сначала разыменовать эти итераторы, а затем использовать оператор == или !=.
- Оператор = присваивает итератору новую позицию (обычно начало или конец элементов контейнера). Чтобы присвоить значение элемента, на который указывает итератор, другому объекту, нужно сначала разыменовать итератор, а затем использовать оператор =.

Каждый контейнерный класс имеет 4 основных метода для работы с оператором =:

- `begin()` возвращает итератор, представляющий начало элементов контейнера.
- `end()` возвращает итератор, представляющий элемент, который находится после последнего элемента в контейнере.
- `cbegin()` возвращает константный (только для чтения) итератор, представляющий начало элементов контейнера.
- `cend()` возвращает константный (только для чтения) итератор, представляющий элемент, который находится после последнего элемента в контейнере.

Наконец, все контейнеры предоставляют (как минимум) два типа итераторов:

`container::iterator` – итератор для чтения/записи;

`container::const_iterator` – итератор только для чтения.

Итерация по vector

Для итератора на `vector` вы можете:

- Выполнять операцию разыменования (обращаться к значению элемента, на которое указывает итератор), как мы это делали с указателем.

```
int x = *it;
```

- Использовать инкремент (it++, ++it) и декремент (it--, --it).
- Применять арифметические операции. Так, например, мы можем сместить итератор на пять ячеек в право, вот так:

```
it += 5;
```

- Сравнивать на равенства.

```
if (it == it2) { ...
```

- Передать переменной разницу итераторов.

```
int x = it - it2;
```

Но о использовании арифметических и сравнительных операциях (>, <, ==) с двумя итераторами, вам нужно кое что знать.

Использовать вышеуказанные операции можно только с идентичными итераторами, которые указывают на одинаковый контейнер и тип.

Например, если мы создали два итератора на один и тот же контейнер, но указали для них разный тип данных и решили использовать вышеуказанные операции – то компилятор выдаст ошибку.

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

vector <int> vector_first;
vector <double> vector_second;
vector <int> ::iterator it = vector_first.begin();
vector <double> ::iterator it2 = vector_second.begin();

if (it == it2) {  // ошибка!
    cout << "it == it2";
}
```

Еще пример, где мы заполним вектор 5-тью числами и, с помощью итераторов, выведем значения вектора:

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

int main()
{
    vector<int> myVector;
    for (int count = 0; count < 5; ++count)
        myVector.push_back(count);
    vector<int>::const_iterator it;
    // объявляем итератор только для чтения
    it = myVector.begin(); // присваиваем ему начало вектора
    while (it != myVector.end())
    // пока итератор не достигнет конца
    {
        cout << *it << " ";
       // выводим значение элемента, на который
       // указывает итератор
        ++it; // и переходим к следующему элементу
    }
    cout << '\n';
  }
```

```
0 1 2 3 4
```

Рис. 12. Результат выполнения программы

Итерации для list, set, map

Для итераторов на **`list, set, map`** немного урезан функционал. Так вы не можете:

- Использовать арифметические операции.

```
it += 5;  //
it *= 2;  //
it /= 3;  // ошибка
it -= 5;  //
```

- Применять операции сравнения (> и <):

```
if (it > it_second) { ...  //
                           // ошибка
if (it < it_second) { ...  //
```

Все остальное можно использовать:

- Применять инкремент и декремент.

```
it--;  // все
it++;  // верно
```

- Использовать операцию разыменования.

```
cout << *it;
*it += 5;
```

- Сравнивать два итератора на равенство и неравенства:

```
if (it == it_second) { ...  //
                            // правильно
if (it != it_second) { ...  //
```

Использовать арифметические операции, чтобы увеличить итератор на один, как это делает инкремент – нельзя. Для этого создали функцию – `advance()`, она заменяет операции увеличения и уменьшения над итераторами.

Вот как она работает:

```
advance(<итератор>, <значение>);
```

- <итератор> — сюда мы должны указать итератор, который и нужно изменить.
- <значение> — тут мы должны вписать число на которое должны увеличить или уменьшить итератор.

Если мы увеличиваем итератор, то используем оператор + к числу. Но можно и просто записать число без оператора +.

Если же нужно уменьшить итератор, то мы добавляем оператор -.

```
advance(it, 5);   // сместили на 5 ячеек
```

Итерация по списку

Выполним всё то же, что и с vector, только уже со списком:

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

```
int main()
{
   list<int> myList;
    for (int count = 0; count < 5; ++count)
        myList.push_back(count);

   list<int>::const_iterator it; // объявляем итератор
    it = myList.begin();
    // присваиваем ему начало списка
    while (it != myList.end())
    // пока итератор не достигнет конца
    {
       cout << *it << " ";
       // выводим значение элемента, на который
       // указывает итератор
       ++it; // и переходим к следующему элементу
    }
    cout << '\n';
}
```

```
0 1 2 3 4
```

Рис. 13. Результат выполнения программы

Итерация по set

В следующей программе мы создадим `set` из 5 чисел и, используя итератор, выведем эти значения:

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

int main()
{
   set<int> mySet;
    mySet.insert(8);
    mySet.insert(3);
    mySet.insert(-4);
    mySet.insert(9);
    mySet.insert(2);

    set<int>::const_iterator it; // объявляем итератор
    it = mySet.begin(); // присваиваем ему начало set-а
    while (it != mySet.end()) // пока итератор не достигнет
конца
    {
        cout << *it << " "; // выводим значение элемента, на
который указывает итератор
        ++it; // и переходим к следующему элементу
    }
```

```
    cout << '\n';
}
```

```
-4 2 3 8 9
```

Рис. 14. Результат выполнения программы

Итерация по ассоциативному массиву(map)

Этот пример немного сложнее. Контейнеры `map` и `multimap` принимают пары элементов (определённых как `std::pair`). Мы используем вспомогательную функцию `make_pair()` для создания пар. `std::pair` позволяет получить доступ к элементу (паре ключ-значение) через первый и второй члены. В нашем ассоциативном массиве мы используем первый член в качестве ключа, а второй – в качестве значения:

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

int main()
{
    map<int, string> myMap;
    myMap.insert(make_pair(3, "cat"));
    myMap.insert(make_pair(2, "dog"));
    myMap.insert(make_pair(5, "chicken"));
    myMap.insert(make_pair(4, "lion"));
    myMap.insert(make_pair(1, "spider"));

    map<int, std::string>::const_iterator it;
    // объявляем итератор
    it = myMap.begin(); // присваиваем ему начало вектора
    while (it != myMap.end())
    // пока итератор не достигнет конца
    {
        cout << it->first << "=" << it->second << " ";
        // выводим значение элемента, на который
        // указывает итератор
        ++it; // и переходим к следующему элементу
    }

    cout << '\n';
}
```

```
1=spider 2=dog 3=cat 4=lion 5=chicken
```

Рис. 15. Результат выполнения программы

Обратите внимание, насколько легко с помощью итераторов перебирать элементы контейнеров. Вам не нужно заботиться о том, как ассоциативный массив хранит свои данные.

Таким образом, итераторы предоставляют простой способ перебора элементов контейнерного класса без необходимости знать реализацию определённого контейнерного класса. В сочетании с алгоритмами STL и методами контейнерных классов итераторы становятся ещё более мощными.

Алгоритмы

Алгоритм STL – это специальные шаблонные функции, которые занимаются разного рода обработкой данных в указанных им интервалах. Используются они достаточно активно – они позволяют искать, сортировать и изменять данные, т.е. алгоритмы в STL занимаются задачами, так или иначе связанными с обработкой однотипных данных, хранящихся в контейнерах. При этом алгоритмам не передаётся сам контейнер (в виде объекта), а диапазон обрабатываемых данных задаётся в виде двух итераторов – один указывает на начало данных, которые должен обработать алгоритм, а второй указывает на конец.

Большинство алгоритмов в STL реализованы в двух вариантах – один из них принимает в качестве параметров начало и конец обрабатываемого интервала данных, а второй принимает ещё и третий параметр – так называемый функциональный объект (функтор). Главное назначение функтора – изменить поведение алгоритма STL. Например, передав соответствующий функтор можно выполнять сортировку по тем или иным признакам объекта (скажем, один функтор отсортирует объекты по их цвету, другой может сортировать по размеру и т.д.)

Чтобы использовать алгоритмы стандартной библиотеки C++, необходимо включить в программу заголовочный файл `<algorithm>`:

```
#include <algorithm>
```

Некоторые алгоритмы STL, предназначенные для обработки числовых данных, определяются в заголовочном файле `<numeric>`:

```
#include <numeric>
```

При работе с алгоритмами также часто применяются объекты функций и функциональные адаптеры, их определения находятся в файле `<functional>`:

```
#inc1ude <functional>
```

По названию алгоритма можно получить первое представление о его назначении. Проектировщики STL ввели два специальных суффикса.

Суффикс _if

Суффикс `_if` используется при наличии двух похожих форм алгоритма с одинаковым количеством параметров; первой форме передается значение, а второй – функция или объект-функции. В этом случае версия без суффикса `_if` используется при передаче значения, а версия с суффиксом `_if` – при передаче функции или объекта-функции. Например, алгоритм `find()` ищет элемент с заданным значением, а алгоритм `find_if()` – элемент, удовлетворяющий критерию, определенному в виде функции или объекта функции.

Впрочем, не все алгоритмы, получающие функции и объекты функций, имеют суффикс _if. Если такая версия вызывается с дополнительными

аргументами, отличающими ее от других версий, за ней сохраняется прежнее имя. Например, версия алгоритма `min_element()` с двумя аргументами находит в интервале минимальный элемент, при этом элементы сравниваются оператором <. В версии `min_element()` с тремя аргументами третий аргумент определяет критерий сравнения.

Суффикс _copy

Суффикс `_copy` означает, что алгоритм не только обрабатывает элементы, но и копирует их в приемный интервал. Например, алгоритм `reverse()` переставляет элементы интервала в обратном порядке, а `reverse_copy()` копирует элементы в другой интервал в обратном порядке.

Классификация алгоритмов

Алгоритмы стандартной библиотеки STL разделяются на следующие категории:

1. Не изменяющие последовательные операции
2. Изменяющие последовательные операции
3. Операции с разделами
4. Операции сортировки
5. Бинарные операции поиска (на упорядоченных последовательностях)
6. Операции над множествами (на упорядоченных массивах)
7. Операции над кучами
8. Операции min/max
9. Операции сравнения

Ниже приведены имена некоторых наиболее часто используемых функций-алгоритмов STL.

Немодифицирующие операции

for_earch() выполняет операции для каждого элемента последовательности
find() находит первое вхождение значения в последовательность
find_if() находит первое соответствие предикату в последовательности
count() подсчитывает количество вхождений значения в последовательность
count_if() подсчитывает количество выполнений предиката в последовательности
search() находит первое вхождение последовательности как подпоследовательности
search_n() находит *n*-е вхождение значения в последовательность

Модифицирующие операции

copy()	копирует последовательность, начиная с первого элемента
swap()	меняет местами два элемента
replace()	заменяет элементы с указанным значением
replace_if()	заменяет элементы при выполнении предиката
replace_copy()	копирует последовательность, заменяя элементы с указанным значением
replace_copy_if()	копирует последовательность, заменяя элементы при выполнении предиката
fill()	заменяет все элементы данным значением
remove()	удаляет элементы с данным значением
remove_if()	удаляет элементы при выполнении предиката
remove_copy()	копирует последовательность, удаляя элементы с указанным значением
remove_copy_if()	копирует последовательность, удаляя элементы при выполнении предиката
reverse()	меняет порядок следования элементов на обратный
random_shuffle()	перемещает элементы согласно случайному равномерному распределению (“тасует” последовательность)
transform()	выполняет заданную операцию над каждым элементом последовательности
unique()	удаляет равные соседние элементы
unique_copy()	копирует последовательность, удаляя равные соседние элементы

Сортировка

sort()	сортирует последовательность с хорошей средней эффективностью
partial_sort()	сортирует часть последовательности
stable_sort()	сортирует последовательность, сохраняя порядок следования равных элементов
lower_bound()	находит первое вхождение значения в отсортированной последовательности
upper_bound()	находит первый элемент, больший чем заданное значение
binary_search()	определяет, есть ли данный элемент в отсортированной последовательности
merge()	сливает две отсортированные последовательности

Работа с множествами

includes()	проверка на вхождение
set_union()	объединение множеств
set_intersection()	пересечение множеств
set_difference()	разность множеств

Минимумы и максимумы

min()	меньшее из двух
max()	большее из двух
min_element()	наименьшее значение в последовательности
max_element()	наибольшее значение в последовательности

Перестановки

next_permutation()	следующая перестановка в лексикографическом порядке
pred_permutation()	предыдущая перестановка в лексикографическом порядке

Задание на домашнюю работу

Написать и отладить три программы.

Первая программа демонстрирует использование контейнерных классов для хранения встроенных типов данных.

Вторая программа демонстрирует использование контейнерных классов для хранения пользовательских типов данных.

Третья программа демонстрирует использование алгоритмов STL.

В программе № 1 выполнить следующее:

1. Создать объект первого контейнера в соответствии с вариантом задания и заполнить его данными, тип которых определяется вариантом задания.
2. Просмотреть контейнер.
3. Изменить контейнер, удалив из него одни элементы и заменив другие.
4. Просмотреть контейнер, используя для доступа к его элементам итераторы.
5. Создать второй контейнер этого же класса и заполнить его данными того же типа, что и первый контейнер.
6. Изменить первый контейнер, удалив из него *n* элементов после заданного и добавив затем в него все элементы из второго контейнера.
7. Просмотреть первый и второй контейнеры.

В программе № 2 выполнить то же самое, но для данных пользовательского типа (созданный класс из ЛР №1).

В программе № 3 выполнить следующее:

1. Создать контейнер, содержащий объекты пользовательского типа. Тип контейнера выбирается в соответствии с вариантом задания.
2. Отсортировать его по убыванию элементов.
3. Просмотреть контейнер.
4. Используя подходящий алгоритм, найти в контейнере элемент, удовлетворяющий заданному условию.
5. Переместить элементы, удовлетворяющие заданному условию в другой (предварительно пустой) контейнер. Тип второго контейнера определяется вариантом задания.
6. Просмотреть второй контейнер.
7. Отсортировать первый и второй контейнеры по возрастанию элементов.

8. Просмотреть их.
9. Получить третий контейнер путем слияния первых двух.
10. Просмотреть третий контейнер.
11. Подсчитать, сколько элементов, удовлетворяющих заданному условию, содержит третий контейнер.
12. Определить, есть ли в третьем контейнере элемент, удовлетворяющий заданному условию.

Варианты заданий

№ п/п	Первый контейнер	Второй контейнер	Встроенный тип данных
1	vector	list	int
2	list	deque	long
3	deque	stack	float
4	stack	queue	double
5	queue	vector	char
6	vector	stack	string
7	map	list	long
8	multimap	deque	float
9	set	stack	int
10	multiset	queue	char
11	vector	map	double
12	list	set	int
13	deque	multiset	long
14	stack	vector	float
15	queue	map	int
16	priority_queue	stack	char
17	map	queue	char
18	multimap	list	int
19	set	map	char
20	multiset	vector	int

Контрольные вопросы и задачи

1. Что означает аббревиатура STL?
2. Дайте определение контейнера.
3. Какие виды встроенных контейнеров в C++ Вы знаете?
4. Какие виды доступа к элементам контейнера Вам известны?
5. Чем отличается прямой доступ от ассоциативного?
6. Перечислите операции, которые обычно реализуются для последовательного доступа к элементам контейнера
7. Дайте определение итератора.
8. Что играет роль итератора для массивов C++?
9. Почему для классов-контейнеров деструктор надо писать явным образом?
10. Можно ли пользовательский класс-итератор реализовать как внешний класс? А как вложенный? В чем отличия этих методов реализации?
11. Что такое стандартный алгоритм?
12. Какие бывают виды алгоритмов?
13. Дан массив a из $n \le 10^5$ чисел $a_i \le 10^9$. Необходимо вывести количество различных элементов в данном массиве.
14. Дана строка s из n слов. Необходимо вывести наиболее часто встречающееся слово и количество его вхождений в строку.
15. Дана группа студентов. Известны фамилии и 4 оценки за летнюю сессию. Выведите на экран фамилии студентов с самыми низким и высоким средними результатами за сессию.