ООП C++. STL (Часть 1)

Встреча 28 (2 пары)

Библиотека стандартных шаблонов (STL) (англ. *Standard Template Library*) — набор согласованных обобщённых алгоритмов, контейнеров, средств доступа к их содержимому и различных вспомогательных функций в C++.

Библиотека стандартных шаблонов до включения в стандарт C++. была сторонней разработкой, вначале — фирмы HP, а затем SGI. Стандарт языка не называет её «STL», так как эта библиотека стала неотъемлемой частью языка, однако многие люди до сих пор используют это название, чтобы отличать её от остальной части стандартной библиотеки (потоки ввода-вывода (iostream), подраздел Си и др.).

Проект под названием STLPort, основанный на SGI STL, осуществляет постоянное обновление STL, iostream и строковых классов. Некоторые другие проекты также занимаются разработкой частных применений стандартной библиотеки для различных конструкторских задач. Каждый производитель компиляторов C++ обязательно поставляет какую-либо реализацию этой библиотеки, так как она является очень важной частью стандарта и широко используется.

Архитектура STL была разработана Александром Степановым и Менг Ли.

Vector (Вектор)

Vector — это замена стандартному динамическому массиву, память для которого выделяется вручную, с помощью оператора **new.**

Разработчики языка рекомендуют в использовать именно vector вместо ручного выделения памяти для массива. Это позволяет избежать утечек памяти и облегчает работу программисту.

Пример:

```
#include <iostream>
#include <vector>
using namespace std;
void main()
{
    // Вектор из 10 элементов типа int
    vector<int> v1(10);

    // Вектор из элементов типа float
    // С неопределенным размером
    vector<float> v2;

    // Вектор, состоящий из 10 элементов типа int
    // По умолчанию все элементы заполняются
нулями
    vector<int> v3(10, 0);
}
```

Методы Vector:

push_back() — добавить последний элемент

pop_back() — удалить последний элемент

clear() — удалить все элементы вектора

empty() — проверить вектор на пустоту

size() — возврат размера

Итераторы для вектора

Итераторы обеспечивают доступ к элементам контейнера. С помощью итераторов очень удобно перебирать элементы. Итератор описывается типом **iterator**. Но для каждого контейнера конкретный тип итератора будет отличаться. Так, итератор для контейнера **list<int>** представляет тип **list<int>::iterator**, а итератор контейнера **vector<int>** представляет тип **vector<int>::iterator** и так далее.

Для получения итераторов контейнеры в C++ обладают такими функциями, как **begin()** и **end().** Функция **begin()** возвращает итератор, который указывает на первый элемент контейнера (при наличии в контейнере элементов). Функция **end()** возвращает итератор, который указывает на следующую позицию после последнего элемента, то есть по сути на конец контейнера. Если контейнер пуст, то итераторы, возвращаемые обоими методами **begin и end** совпадают. Если итератор **begin** не равен итератору **end**, то между ними есть как минимум один элемент.

Обе этих функции возвращают итератор для конкретного типа контейнера:

```
std::vector<int> v = { 1,2,3,4 };
std::vector<int>::iterator iter = v.begin();  // получаем итератор
```

Операции с итераторами

С итераторами можно проводить следующие операции:

***iter**: получение элемента, на который указывает итератор

++iter: перемещение итератора вперед для обращения к следующему элементу

--iter: перемещение итератора назад для обращения к предыдущему элементу. Итераторы контейнера forward_list не поддерживают операцию декремента.

iter1 == iter2: два итератора равны, если они указывают на один и тот же элемент

iter1 != iter2: два итератора не равны, если они указывают на разные элементы

```
#include <iostream>
#include <vector>

int main()
{
   std::vector<int> v = {1, 2, 3, 4, 5};
   auto iter = v.begin();
   while(iter!=v.end())    // пока не дойдем до конца
   {
      *iter = (*iter) * (*iter); // возводим число в квадрат
      ++iter;
   }

   for(iter = v.begin(); iter!=v.end(); ++iter)
   {
      std::cout << *iter << std::endl;
   }

   return 0;
}
```

List

Контейнер **list** представляет двухсвязный список. Для его использования необходимо подключить заголовочный файл **list**:

```
#include <list>
```

Создание списка:

```
std::list<int> list1;      // пустой список
std::list<int> list2(5);  // список list2 состоит из 5 чисел, каждый элемент имеет значение по умолчанию

std::list<int> list3(5, 2);          // список list3 состоит из 5 чисел, каждое число равно 2
std::list<int> list4{ 1, 2, 4, 5 };   // список list4 состоит из чисел 1, 2, 4, 5
std::list<int> list5 = { 1, 2, 3, 5 }; // список list5 состоит из чисел 1, 2, 4, 5
std::list<int> list6(list4);         // список list6 - копия списка list4
std::list<int> list7 = list4;         // список list7 - копия списка list4
```

Получение элементов

В отличие от других контейнеров для типа list не определена операция обращения по индексу или функция at(), которая выполняет похожую задачу.

Тем не менее для контейнера list можно использовать функции **front()** и **back()**, которые возвращают соответственно первый и последний элементы.

Чтобы обратиться к элементам, которые находятся в середине (после первого и до последнего элементов), придется выполнять перебор элементов с помощью циклов или итераторов:

```
#include <iostream>
#include <list>

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

    int first = numbers.front();  // 1
    int last = numbers.back();  // 5

    // перебор в цикле
    for (int n : numbers)
        std::cout << n << "\t";
    std::cout << std::endl;

    // перебор с помощью итераторов
    for (auto iter = numbers.begin(); iter != numbers.end(); iter++)
    {
        std::cout << *iter << "\t";
    }
    std::cout << std::endl;
    return 0;
}
```

Размер списка

Для получения размера списка можно использовать функцию **size()**:

```
std::list<int> numbers = { 1, 2, 3, 4, 5 };
int size = numbers.size();   // 5
```

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

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

С помощью функции resize() можно изменить размер списка. Эта функция 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

```
std::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

```
std::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}

std::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:

```
std::list<int> list1 = { 1, 2, 3, 4, 5 };
std::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 контейнер list применяется ряд функций.

- **push_back(val)**: добавляет значение val в конец списка
- **push_front(val)**: добавляет значение val в начало списка
- **emplace_back(val)**: добавляет значение val в конец списка
- **emplace_front(val)**: добавляет значение val в начало списка
- **emplace(pos, val)**: вставляет элемент val на позицию, на которую указывает итератор pos. Возвращает итератор на добавленный элемент
- **insert(pos, val)**: вставляет элемент val на позицию, на которую указывает итератор pos, аналогично функции emplace. Возвращает итератор на добавленный элемент
- **insert(pos, n, val)**: вставляет n элементов val начиная с позиции, на которую указывает итератор pos. Возвращает итератор на первый добавленный элемент. Если n = 0, то возвращается итератор pos.
- **insert(pos, begin, end)**: вставляет начиная с позиции, на которую указывает итератор pos, элементы из другого контейнера из диапазона между итераторами begin и end. Возвращает итератор на первый добавленный элемент. Если между итераторами begin и end нет элементов, то возвращается итератор pos.
- **insert(pos, values)**: вставляет список значений values начиная с позиции, на которую указывает итератор pos. Возвращает итератор на первый добавленный элемент. Если values не содержит элементов, то возвращается итератор pos.

Функции push_back(), push_front(), emplace_back() и emplace_front():

```
std::list<int> numbers = { 1, 2, 3, 4, 5 };
numbers.push_back(23);  // { 1, 2, 3, 4, 5, 23 }
numbers.push_front(15); // { 15, 1, 2, 3, 4, 5, 23 }
numbers.emplace_back(24);   // { 15, 1, 2, 3, 4, 5, 23, 24 }
numbers.emplace_front(14);  // { 14, 15, 1, 2, 3, 4, 5, 23, 24 }
```

Добавление в середину списка с помощью функции **emplace()**:

```
std::list<int> numbers = { 1, 2, 3, 4, 5 };
auto iter = ++numbers.cbegin(); // итератор указывает на второй
элемент
numbers.emplace(iter, 8); // добавляем после первого элемента
numbers = { 1, 8, 2, 3, 4, 5};
```

Добавление в середину списка с помощью функции **insert()**:

```
std::list<int> numbers1 = { 1, 2, 3, 4, 5 };
auto iter1 = numbers1.cbegin(); // итератор указывает на первый элемент
numbers1.insert(iter1, 0); // добавляем начало списка
//numbers1 = { 0, 1, 2, 3, 4, 5};

std::list<int> numbers2 = { 1, 2, 3, 4, 5 };
auto iter2 = numbers2.cbegin(); // итератор указывает на первый элемент
numbers2.insert(++iter2, 3, 4); // добавляем после первого элемента три
четверки
//numbers2 = { 1, 4, 4, 4, 2, 3, 4, 5};

std::list<int> values = { 10, 20, 30, 40, 50 };
std::list<int> numbers3 = { 1, 2, 3, 4, 5 };
auto iter3 = numbers3.cbegin(); // итератор указывает на первый элемент
// добавляем в начало все элементы из values
numbers3.insert(iter3, values.begin(), values.end());
//numbers3 = { 10, 20, 30, 40, 50, 1, 2, 3, 4, 5};

std::list<int> numbers4 = { 1, 2, 3, 4, 5 };
auto iter4 = numbers4.cend();   // итератор указывает на позицию за
последним элементом
// добавляем в конец список из трех элементов
numbers4.insert(iter4, { 21, 22, 23 });
//numbers4 = { 1, 2, 3, 4, 5, 21, 22, 23};
```

Удаление элементов

Для удаления элементов из контейнера list могут применяться следующие функции:

clear(p): удаляет все элементы

pop_back(): удаляет последний элемент

pop_front(): удаляет первый элемент

erase(p): удаляет элемент, на который указывает итератор p. Возвращает итератор на элемент, следующий после удаленного, или на конец контейнера, если удален последний элемент

erase(begin, end): удаляет элементы из диапазона, на начало и конец которого указывают итераторы begin и end. Возвращает итератор на элемент, следующий после последнего удаленного, или на конец контейнера, если удален последний элемент

```
std::list<int> numbers = { 1, 2, 3, 4, 5 };
numbers.pop_front();    // numbers = { 2, 3, 4, 5 }
numbers.pop_back();     // numbers = { 2, 3, 4 }
numbers.clear();    // numbers ={}

numbers = { 1, 2, 3, 4, 5 };
auto iter = numbers.cbegin(); // указатель на первый элемент
numbers.erase(iter);    // удаляем первый элемент
// numbers = { 2, 4, 5, 6 }

numbers = { 1, 2, 3, 4, 5 };
auto begin = numbers.begin(); // указатель на первый элемент
auto end = numbers.end();       // указатель на последний элемент
numbers.erase(++begin, --end);  // удаляем со второго элемента до последнего
//numbers = {1, 5}
```

Map

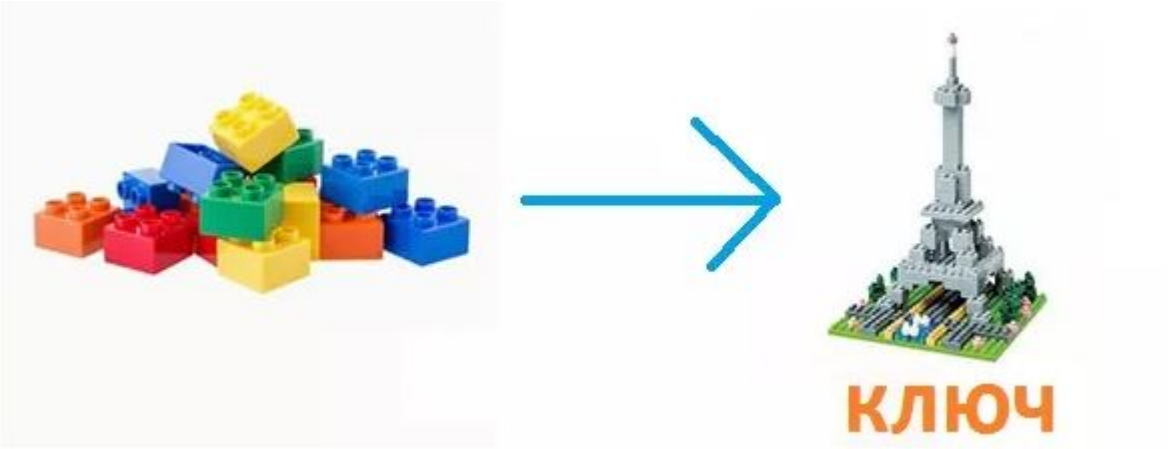
Что такое map

Это ассоциативный контейнер, который работает по принципу — [ключ — значение]. Он схож по своему применению с вектором и массивом, но есть некоторые различия:

1. Ключом может быть все что угодна. От обычной переменной до класса

```
1  mp1[0] = 7; // ключ - число
2
3  mp2["zero"] = 4;  // ключ - строка
4
5  pair <int, int> p = make_pair(1, 3);
6  mp3[p] = 3;  // ключ - пара
```

2. При добавлении нового элемента контейнер будет отсортирован по возрастанию.



Как создать map

Сперва понадобится подключить соответствующую библиотеку:

```
#include <map>
```

Чтобы создать map нужно воспользоваться данной конструкцией:

```
map < <L>, <R> > <имя>;
```

<L> — этот тип данных будет относиться к значению ключа.
<R> — этот тип данных соответственно относится к значению.

```
map <string, int> mp; // пример
```

Также имеется возможность добавить значения при инициализации (C++ 11 и выше):

```
map <string, string> book = {{"Hi", "Привет"},
                             {"Student", "Студент"},
                             {"!", "!"}};

cout << book["Hi"];
```

Итераторы для map

Итераторы для map

Использование итераторов одна из главных тем, если вам понадобится оперировать с этим контейнером. Создание итератора, как обычно происходит так:

```
map <тип данных> :: iterator <имя>;
```

С помощью его можно использовать две операции (**it — итератор**):

Чтобы обратится к ключу нужно сделать так: **it->first**.

Чтобы обратится к значению ячейки нужно сделать так: **it->second**.

Нельзя обращением к ключу **(...->first)** изменять его значение, а вот изменять таким образом значение ячейки **(...->second)** легко.

Нельзя использовать никакие арифметические операции над итератором.

Чтобы не писать циклы для увеличения итератора на большие значения, можно воспользоваться функцией **advance()**:

```
advance(it, 7);
advance(it, -5);
```

Она сдвигает указанный итератор вниз или вверх на указанное количество ячеек. В нашем случае он сначала увеличит на 7, а потом уменьшит на 5, в итоге получится сдвиг на две вверх.

```
#include <iostream>
#include <map>

using namespace std;
int main() {
   setlocale(0, "");
   map <int, int> mp;

   cout << "Введите количество элементов: "; int n; cin >> n;

   for (int i = 0; i < n; i++) {
      cout << i << ") "; int a; cin >> a;
      mp[a] = i;  // добавляем новые элементы
   }

   map <int, int> :: iterator it = mp.begin();
   cout << "А вот все отсортированно: " << endl;
   for (int i = 0; it != mp.end(); it++, i++) {  // выводим их
      cout << i << ") Ключ " << it->first << ", значение " << it->second << endl;
   }

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

Методы map

- **insert**

Это функция вставки нового элемента.

```
mp.insert(make_pair(num_1, num_2));
```

num_1 — ключ.

num_2 — значение.

Мы можем сделать то же самое вот так:

```
mp[num_1] = num_2;
```

- **count**

Возвращает количество элементов с данным ключом. В нашем случае будет возвращать — 1 или 0.

Эта функция больше подходит для multimap, у которого таких значений может быть много.

```
mp[0] = 0;

mp.count(0); // 1
mp.count(3); // 0
```

Нужно помнить, что для строк нужно добавлять кавычки — count("Good")

- **find**

У этой функции основная цель узнать, есть ли *определенный ключ в контейнере*.

- Если он есть, то передать итератор на его местоположение.
- Если его нет, то передать итератор на конец контейнера.

```
#include <iostream>
#include <map>  // подключили библиотеку

using namespace std;

int main() {
  setlocale(0, "");
  map <string, string> book;
  book["book"] = "книга";

  map <string, string> :: iterator it, it_2;

  it = book.find("book");
  cout << it->second << endl;

  it_2 = book.find("books");

  if (it_2 == book.end()) {
      cout << "Ключа со значением 'books' нет";
  }

  system("pause");
  return 0;
}num_1, num_2));
```

- **erase**

Иногда приходится удалять элементы. Для этого у нас есть функция — **erase().**
Давайте посмотрим как она работает на примере:

```
map <string, string> passport;

passport["maxim"] = "Denisov";     // добавляем
passport["andrey"] = "Puzerevsky"; // новые
passport["dima"] = "Tilyupo";      // значения

cout << "Size: " << passport.size() << endl;

map <string, string> :: iterator full_name; // создали
итератор на passport

full_name = passport.find("andrey"); // находим ячейку
passport.erase(full_name);           // удаляем

cout << "Size: " << passport.size();
```

Спасибо за внимание.
Вопросы.