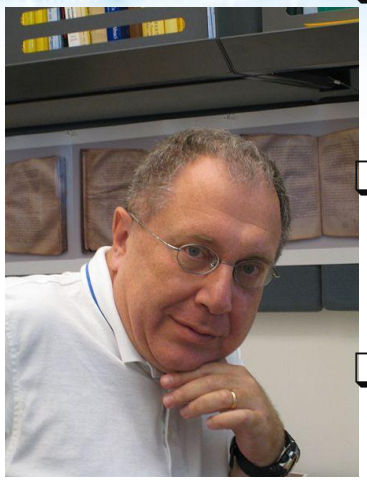
Основные концепции стандартной библиотеки шаблонов.

Липкин Николай

О чем пойдет речь

- Стандартная библиотека шаблонов
- Итераторы
- Контейнеры
- Алгоритмы
- Вспомогательные компоненты библиотеки

Стандартная библиотека шаблонов



- Стандартная библиотека шаблонов разработана Александром Степановым и Менгом Ли (Hewlett-Packard)
- Стандартная библиотека шаблонов (Standard Template Library, STL) входит в стандартную библиотеку языка C++.
- В нее включены реализации и снова. наиболее часто используемых контейнеров и алгоритмов, что избавляет программистов от их рутинного переписывания.
- Библиотека STL использует наиболее продвинутые возможности языка C++ и является одновременно эффективной и универсальной.

Краеугольные камни STL

- *Контейнер* - это хранилище объектов, как встроенных, так и определенных пользователем типов. Простейшие виды контейнеров встроены непосредственно в язык C++. Стандартная библиотека включает в себя реализации 7 основных типов контейнеров и 3 производных.

- *Алгоритм* - это функция для манипулирования объектами, содержащимися в контейнере. Типичные примеры алгоритмов - сортировка и поиск. В STL реализовано порядка 60 алгоритмов, которые можно применять к различным контейнерам, в том числе к массивам, встроенным в язык C++.

- *Итератор* - это абстракция указателя: объект, который может ссылаться на другие объекты, содержащиеся в контейнере. Основные функции итератора - обеспечение доступа к объекту, на который он ссылается (разыменование), и переход от одного элемента контейнера к другому.

Итераторы

- Итератор - это объект который инкапсулирует указатель на текущий элемент и способ обхода контейнера
- Итераторы используются для доступа к элементам контейнера так же, как указатели - для доступа к элементам обычного массива.
- В языке C++ над указателями можно выполнять следующий набор операций: разыменование, инкремент/декремент, сложение/вычитание и сравнение.
- Некоторые итераторы позволяют работать с объектами в режиме ограниченного доступа: "только чтение" или "только запись»

Итераторы

Общие свойства итераторов:

1) Наличие конструктора по умолчанию

```
vector[i]::iterator i;
```

2) Возможность разыменовывания

```
int n = *i;
```

3) Возможность присваивания если итератор изменяемый

```
*i = 2;
```

4) Доступ к членам с помощью -> если итератор указывает на сложный тип

```
map[i]::iterator i = M.begin();
    cout << i->first;
```

5) Возможность сравнения двух итераторов

```
while( begin != end ) ++begin;
```

Итераторы

В зависимости от набора поддерживаемых операций различают 5 типов итераторов.

Тип итератора	Доступ	Разыменование	Итерация	Сравнение
Итератор вывода (output iterator)	Только запись	*	++	
Итератор ввода (input iterator)	Только чтение	*, ->	++	==, !=
Прямой итератор (forward iterator)	Чтение и запись	*, ->	++	==, !=
Двунаправленный итератор (bidirectional iterator)	Чтение и запись	*, ->	++, --	==, !=
Итератор с произвольным доступом (random-access iterator)	Чтение и запись	*, ->, []	++, --, +, -, +=, -=	==, !=, <, <=, >, >=

Итераторы ввода (input iterator)

- Наиболее простые из всех итераторов STL, и доступны они только для чтения
- Итераторы ввода возвращает только шаблонный класс istream_iterator.
- Вместо итератора ввода может подставляться любой из основных итераторов, за исключением итератора вывода

Итераторы ввода (input iterator)

Пример:

алгоритм for_each

```
template <class InputIterator, class Function>
Function for_each (InputIterator first, InputIterator last, Function f)
{
        while (first != last)
        f(*first++);
        return f;
}
```

Печать массива с использованием for_each

```
#include <iostream>
#include <algorithm>
using namespace std;
void printValue(int num)
 {
        cout << num << "n";
}
main(void)
{
        int init[] = {1, 2, 3, 4, 5};
        for_each(init, init + 5, printValue);
}
```

Итераторы вывода (output iterator)

Итератор вывода служит для ссылки на область памяти, куда выводятся данные.

Итераторы вывода можно встретить повсюду, где происходит хоть какая-то обработка информации средствами STL. Это могут быть алгоритмы копирования, склейки и т. п.

Для данного итератора определены операторы присвоения (=), разыменовывания (*) и инкремента (++) (во время присвоения итератор вывода должен быть целевым итератором, которому присваиваются значения).

Итераторы ввода могут быть возвращены итераторами потоков вывода (ostream_iterator) и итераторами вставки inserter, front_inserter и back_inserter

Итераторы вывода (output iterator)

Пример использования итератора вывода

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

main(void) {
int init1[] = {1, 2, 3, 4, 5};
int init2[] = {6, 7, 8, 9, 10};
vector<int> v(10);
merge(init1, init1 + 5, init2, init2 + 5, v.begin());
copy(v.begin(), v.end(), ostream_iterator<int>(cout, "n"));
}
```

Однонаправленные итераторы (forward iterator)

Если соединить итераторы ввода и вывода, то получится однонаправленный итератор, который может перемещаться по цепочке объектов в одном направлении.

Для такого перемещения в итераторе определена операция инкремента (++).

И разумеется, в однонаправленном итераторе есть операторы сравнения (== и !=), присвоения (=) и разыменовывания (*).

Пример: алгоритм замены значения

```
template <class ForwardIterator, class T>
void replace (ForwardIterator first, ForwardIterator last, const T& old_value, const T& new_value) {
        while (first != last) {
                if (*first == old_value) *first = new_value; ++first;
                }
}
```

Однонаправленные итераторы (forward iterator)

Составим программу, заменяющую в исходном массиве все единицы на нули и наоборот (инверсия), используя определенный ранее алгоритм

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

main(void) {
replace(init, init + 5, 0, 2);
replace(init, init + 5, 1, 0);
replace(init, init + 5, 2, 1);
copy(init, init + 5, ostream_iterator<int>(cout, "n"));
}
```

Двунаправленный итератор (bidirectional iterator)

Двунаправленный итератор аналогичен однонаправленному итератору.

В отличие от последнего двунаправленный итератор может перемещаться не только из начала в конец цепочки объектов, но и наоборот.

Это становится возможным благодаря наличию оператора декремента (-).

На двунаправленных алгоритмах базируются различные алгоритмы, выполняющие реверсивные операции, например reverse. Этот алгоритм меняет местами все объекты в цепочке, на которую ссылаются переданные ему итераторы.

Двунаправленный итератор (bidirectional iterator)

Пример использования итератора

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

main(void) {
int init[] = {1, 2, 3, 4, 5};
reverse(init, init + 5);
copy(init, init + 5, ostream_iterator<int>(cout, "n"));
}
```

Итераторы двунаправленного доступа возвращаются несколькими контейнерами STL: list, set, multiset, map и multimap

Итераторы произвольного доступа (random-access iterator)

Итераторы произвольного доступа - самые "умелые" из основных итераторов.

Не только реализуют все функции, свойственные итераторам более низкого уровня, но и обладают большими возможностями.

Как правило, все сложные алгоритмы, требующие расширенных вычислений, оперируют итераторами произвольного доступа.

Ниже приводится пример, в котором мы используются практически все операции, допустимые для этого типа итераторов.

Итераторы произвольного доступа (random-access iterator)

```
#include <algorithm>
#include <iostream>
#include <vector>
#define space " "
using namespace std;
int main(void) {
const int init[] = {1, 2, 3, 4, 5};
vector<int> v(5);
//Создаем переменную типа "итератор произвольного доступа"
typedef vector<int>::iterator vectItr;
vectItr itr ;
//Инициализируем вектор значениями из массива констант и присваиваем адрес его первого элемента итератору произвольного доступа:
copy(init, init + 5, itr = v.begin());
//Последовательно читаем элементы вектора, начиная с конца, и выводим их на экран:
cout << *( itr + 4 ) << endl;
cout << *( itr += 3 ) << endl;
cout << *( itr -= 1) << endl;
cout << *( itr = itr - 1) << endl;
cout << *( -itr ) << endl;
//Итератор, претерпев несколько изменений, снова указывает на первый элемент вектора.
for(int i = 0; i < (v.end() - v.begin()); i ++) cout << itr[i] << space;    cout << endl;
}
```

Реверсивные итераторы (reverse iterator)

Некоторые классы-контейнеры спроектированы так, что по хранимым в них элементам данных можно перемещаться в заданном направлении.

В одних контейнерах это направление от первого элемента к последнему, а в других - от элемента с самым большим значением к элементу, имеющему наименьшее значение.

Существует специальный вид итераторов, называемых реверсивными. Такие итераторы работают “наоборот": т. е. если в контейнере итератор ссылается на первый элемент данных, то реверсивный итератор ссылается на последний.

Получить реверсивный итератор для контейнера можно вызовом метода rbegin(), а реверсивное значение "за пределом" возвращается методом rend().

```
main(void) {
        const int init[] = {1, 2, 3, 4, 5};
        vector<int> v(5);
        copy(init, init + 5, v.begin());
        copy(v.begin(), v.end(), ostream_iterator<int>(cout, " "));
        copy(v.rbegin(), v.rend(), ostream_iterator<int>(cout, " "));
}
```

Контейнеры

- Контейнер предназначен для хранения объектов.
- Каждый контейнер обязан предоставить строго определенный интерфейс, через который с ним будут взаимодействовать алгоритмы. Этот интерфейс обеспечивают итераторы.
- Каждый контейнер обязан иметь соответствующий ему итератор (и только итератор). Важно подчеркнуть, что никакие дополнительные функции-члены для взаимодействия алгоритмов и контейнеров не используются.
- Каждый контейнер реализует определенный тип итераторов. При этом выбирается наиболее функциональный тип итератора, который может быть эффективно реализован для данного контейнера.

Контейнеры

- Вне зависимости от фактической организации контейнера (вектор, список, дерево) хранящиеся в нем элементы можно рассматривать как последовательность.
- Итератор первого элемента в этой последовательности возвращает функция begin(), а итератор элемента, следующего за последним - функция end(). Это очень важно, так как все алгоритмы в STL работают именно с последовательностями, заданными итераторами начала и конца.
- Кроме обычных итераторов в STL существуют обратные итераторы (reverse iterator), позволяющие применять алгоритмы как к прямой, так и к обратной последовательности элементов.

Контейнеры

- Контейнер может хранить объекты определенного программистом типа.
- Все контейнеры динамически изменяют свой размер и следят за используемой ими памятью, то есть при удалении контейнера гарантируется удаление всех объектов в контейнере.

```
#include <cassert>          // для функции assert (проверка корректности утверждения)
#include <vector>
#include <iostream>
#include <algorithm>
using namespace std;
int main(int argc, char* argv[])
{
    vector[i] V;
    assert( V.size() == 0 );
    V.push_back( 23 );
    V.push_back( 34.56 );
    assert( V.size() == 2 );
    assert( V[0] == 23 );
    assert( V[1] == 34 );
    copy( V.begin(), V.end(), ostream_iterator[i]( cout, "n" ) );
        return 0;
}
```

Контейнеры

Контейнеры делятся на два типа :

1) Последовательные (массив, список, дек)
2) Ассоциативные (множество, ассоциативный массив)

- У последовательных контейнеров элементы линейно упорядочены (пронумерованы)
- В ассоциативных контейнерах элементы могут храниться в произвольном порядке
- У всех последовательных контейнеров есть функция push_back для добавления элементов в конец, если необходимо добавить элемент в произвольную позицию можно воспользоваться функцией insert

```
vector[i] V;
V.push_back( 1 );
V.insert( V.begin(), 2 );
```

- В ассоциативные контейнеры элементы добавляются функцией insert

Алгоритмы

- Алгоритмы предназначены для манипулирования элементами контейнера.
- Любой алгоритм рассматривает содержимое контейнера как последовательность, задаваемую итераторами первого и следующего за последним элементов. Итераторы обеспечивают интерфейс между контейнерами и алгоритмами, благодаря чему и достигается гибкость и универсальность библиотеки STL.
- Каждый алгоритм использует итераторы определенного типа.
- Вместо менее функционального итератора можно передать алгоритму более функциональный, но не наоборот.
- Все стандартные алгоритмы описаны в файле algorithm, в пространстве имен std.

Алгоритмы

Основные виды алгоритмов:

- Математические (расчет сумм, произведений, генерация случайных значений)
- Поиска (минимальное значение, поиск последовательности, подсчет числа значений)
- Сортировки
- Работы с последовательностями (объединение последовательностей, сравнения, обработки последовательности типовой операцией)

Алгоритмы

STL - алгоритмы представляют набор готовых функций, которые могут быть применены к STL коллекциям и могут быть подразделены на три основных группы:

1) Функции для перебора всех членов коллекции и выполнения определенных действий над каждым из них:

count, count_if, find, find_if, adjacent_find, for_each, mismatch, equal, search copy, copy_backward, swap, iter_swap, swap_ranges, fill, fill_n, generate, generate_n, replace, replace_if, transform, remove, remove_if, remove_copy, remove_copy_if, unique, unique_copy, reverse, reverse_copy, rotate, rotate_copy, random_shuffle, partition, stable_partition

2) Функции для сортировки членов коллекции:

Sort, stable_sort, partial_sort, partial_sort_copy, nth_element, binary_search, lower_bound, upper_bound, equal_range, merge, inplace_merge, includes, set_union, set_intersection, set_difference, set_symmetric_difference, make_heap, push_heap, pop_heap, sort_heap, min, max, min_element, max_element, lexographical_compare, next_permutation, prev_permutation

3) Функции для выполнения определенных арифметических действий над членами коллекции:

Accumulate, inner_product, partial_sum, adjacent_difference

Алгоритмы

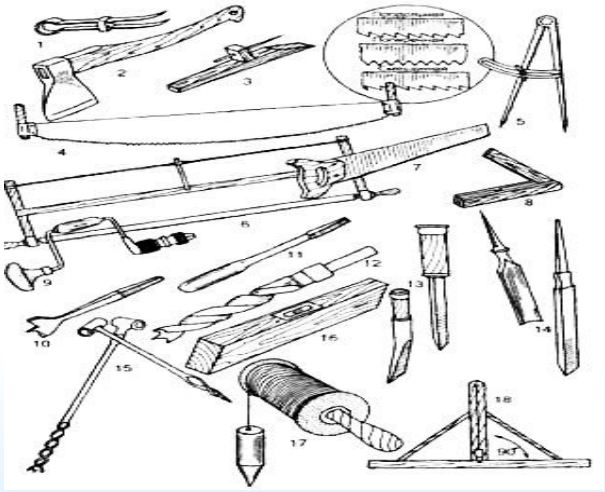
Пример:

Использование алгоритма stable_sort. Сортируем строку вне зависимости от регистра букв

```
inline bool lt_nocase(char c1, char c2)
{ return tolower(c1) < tolower(c2); }
int main()
{
  char A[] = "fdBeACFDbEac";
  const int N = sizeof(A) - 1;
  stable_sort(A, A+N, lt_nocase);
  printf("%s\n", A);
  // Результат "AaBbCcdDeEfF".
}
```

Вспомогательные компоненты библиотеки

- Помимо рассмотренных элементов в STL есть ряд второстепенных понятий, таких как:
  - *Аллокаторы*
  - *Функторы*
  - *Предикаты*
  - *Адаптеры*



Аллокатор

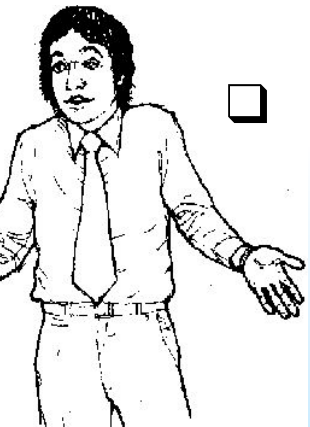
- Аллокатор (allocator) - это объект, отвечающий за распределение памяти для элементов контейнера. С каждым стандартным контейнером связывается аллокатор.

- Если какому-то алгоритму требуется распределять память для элементов, он обязан делать это через аллокатор. В этом случае можно быть уверенным, что распределенные объекты будут уничтожены правильно.

В состав STL входит стандартный класс allocator. Его по умолчанию используют все контейнеры, реализованные в STL.

Функторы

- Функтор (или функциональный объект) - любой объект, к которому можно применить оператор вызова функции - operator(). Такой вызов в C++ применим к имени функции, указателю на функцию или объекту класса, который переопределяет operator().
- Функтор – функция, которая ведет себя как объект, или объект, который ведет себя как функция.
- Реализация функторов проста: функтор оформляется в виде класса, содержащего всего одну функцию



```
struct functor
{
      int operator()(int i)
      {
       // операции над переменной,
    например
            return i+1;
      };
}
```

Функторы

Что дает применение функторов ?

1) Наследование позволяет выделить общий код в базовый класс. В распоряжении программиста появляется языковая конструкция для объединения логически связанных функций
2) Применение виртуальных функций с наследованием обеспечивает ту же гибкость на стадии выполнения, которая достигается применением указателей на функцию

Функторы

```
class X{
public:
    virtual string operator()(string x){
        return "x "+x;
    }
};

class Y: public X{
    public:
    string operator()(string x){
        return "Y "+x;
    }
};
```

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

int main(){
    X x;
    Y y;
    X *a;
    a = &y;
    cout<<a->operator() ("a")<<endl;
    a=&x;
    cout<<a->operator() ("b")<<endl;
}
```

Функторы

В STL существует понятие адаптированного функтора, это функтор с обявлеными типами параметров и возвращаемого значения

```
struct X
{
	typedef int argument_type;
	typedef int result_type;
	result_type operator()( const argument_type& val)
	{
		return -val;
	}
};
```

Список функторов, выполняющих сравнение:
equal_to, not_equal_to, less, greater, less_equal, greater_equal

Предикаты

- Предикат - функция (или функциональный объект), которая в зависимости от значения аргументов может возвращать значение истина, или ложь.
- Создадим предикат, который определяет четность числа и, если число - четное, возвращает true. Продемонстрируем использование предиката.

```
bool isEven(int num) {
return bool(num % 2);
}
main(void) { list<int> l;
list<int>::iterator t;
for(int i = 1; i <= 10; i++) l.push_back(i);
copy(l.begin(), l.end(), ostream_iterator<int>(cout, " ")); cout << endl;
t = remove_if(l.begin(), l.end(), isEven);
copy(l.begin(), t, ostream_iterator<int>(cout, " "));
}
```

Адаптеры

- Адаптер - класс, который не реализует собственную функциональность, а вместо этого предоставляет альтернативный интерфейс к функциональности другого класса, или
- Адаптеры это объекты, которые изменяют интерфейс адаптируемого объекта, не добавляя при этом функциональности
- В STL адаптеры применяются для самых различных классов (контейнеров, объектов-функций и т. д.). Наиболее типичный пример адаптера - стек. Стек может использовать в качестве нижележащего класса различные контейнеры (очередь, вектор, список), обеспечивая для них стандартный стековый интерфейс (функции push/pop).

Адаптеры

Примером может служить адаптер выходного потока ostream_iterator
(так же существует адаптер и для входного потока)

```
vector[i] V;
copy( istream_iterator[i]( cin ), istream_iterator[i](), back_inserter( V ) );
sort( V.begin(), V.end() );
copy( V.begin(), V.end(), ostream_iterator[i]( cout, " " ) );
```

Алгоритм copy читает из входного потока до тех пор, пока не будет окончен ввод, и добавлять элементы в конец вектора.

Помимо back_inserter существуют также front_inserter и inserter

```
list[i] L;
copy( istream_iterator[i]( cin ), istream_iterator[i](), front_inserter( L ) );
copy( L.begin(), L.end(), ostream_iterator[i]( cout, " " ) );
cout << endl;
set[i] S;
copy( L.begin(), L.end(), inserter( S, S.begin() ) );
copy( S.begin(), S.end(), ostream_iterator[i]( cout, " " ) );
```

Спасибо за внимание