# *Лабораторная работа 2*

# **РАЗРАБОТКА КОНТЕЙНЕРА ПО СТАНДАРТУ STL**

**Цель работы**

Знакомство с компонентами стандартной библиотекой шаблонов (STL) и принципами ее организации, изучение стандартов разработки компонентов STL, получение практического навыка разработки собственного контейнера и итератора.

**Теоретические сведения**

В основе стандартной библиотеки шаблонов (***Standard Template Library, STL***) лежат шесть понятий:

* ***контейнеры*** – объекты, содержащие последовательность элементов типа *value\_type*;
* ***итераторы*** представляющие собой не зависящие от структуры данных абстракции, позволяющие получать доступ к элементам контейнера и обходить диапазоны элементов;
* ***алгоритмы*** осуществляют обобщенное манипулирование диапазонами элементов, не требующее знания о типах самих элементов и структурах данных, в которых они размещаются;
* ***объекты-функции*** определяют обобщенные операции, применяемые к элементам, которыми манипулируют алгоритмы;
* ***адаптеры*** позволяют согласовать между собой несопоставимые типы путем изменения их интерфейсов. Различают адаптеры класса (они адаптируют сами типы) и адаптеры экземпляра (они адаптируют экземпляры типов);
* ***распределители*** абстрагируют операции распределения памяти и конструирования объектов, выполняемые контейнерами.

**Контейнеры**

В стандарте определено четыре основных *последовательных* и четыре *ассоциативных* контейнера, а также три *адаптера* последовательных контейнеров. Операции, предусмотренные стандартными контейнерами, представлены на рисунке 1.

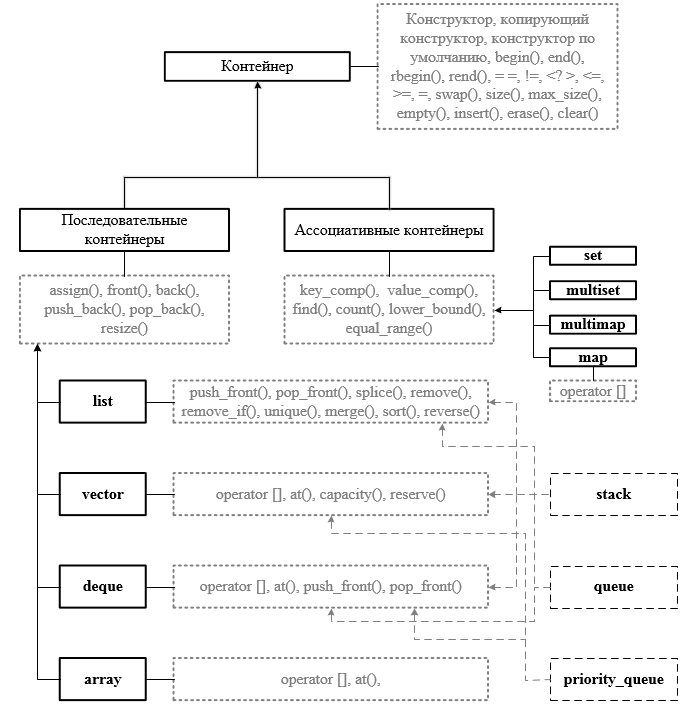


Рисунок 1 – Стандартные контейнеры STL и их операции

***Последовательный контейнер*** поддерживает строгую линейную упорядоченность своих элементов. Три наиболее употребительных шаблонных класса, соответствующих последовательным контейнерам:

* *deque* – двусторонняя очередь – представляет собой последовательный индексированный контейнер, который позволяет быстро вставлять и удалять элементы с начала и с конца;
* *list* – двусвязный список – поддерживает быструю вставку и удаление элементов из любой позиции в контейнере, но не предоставляет быстрый произвольный доступ;
* *vector* – последовательный контейнер, инкапсулирующий массивы переменного размера и обеспечивающий *непрерывное* хранение элементов в памяти;
* *array* –последовательный контейнер, представляющий собой массив фиксированного размера.

Соответственно, в общем случае стандарт рекомендует использовать *list* в случае, если выполняются частые вставки и удаления в середине последовательности, *deque* – если вставки и удаления выполняются преимущественно в начале или в конце последовательности, и *vector* – во всех остальных случаях.

Среди всех контейнеров, определенных в стандартной библиотеке, контейнер *vector* является особенным в смысле совместимости с C API, так как *vector* гарантирует, что для хранения элементов контейнера выделяется непрерывная область памяти. Иными словами, для любого непустого экземпляра такой код корректен:

extern "C" void sort\_ints(int\* p, size\_t n);

std::vector<int> v = {1, 2, 3, 4, 5};

assert(!v.empty());

sort\_ints(&v[0], v.size());

Остальные контейнеры представляют собой *связные списки*, следовательно, стандарт не обеспечивает для них непрерывность выделения памяти.

**Адаптеры контейнеров**

***Адаптеры*** последовательных контейнеров включают следующие три шаблонных класса: *queue* (очередь), *priority\_queue* (очередь с приоритетом) и *stack* (стек). Все они являются адаптерами *класса* (а не *экземпляра*), то есть могут применяться для адаптации любого типа последовательного контейнера, удовлетворяющего определенным требованиям:

* последовательный контейнер, поддерживающий операции *back*, *push\_back* и *pop\_back*, может использоваться для адаптера *stack* (например, *vector*, *list* и *deque*);
* последовательный контейнер, поддерживающий операции *front*, *push\_back* и *pop\_front*, может использоваться для адаптера *queue* (например, *list* и *deque*);
* последовательный контейнер с итератором произвольного доступа (см. далее), поддерживающий операции *front, push\_back* и *pop\_back,* может использоваться для адаптера *priority\_queue* (например, *vector* и *deque*).

***Ассоциативный контейнер*** обеспечивает быстрый поиск элементов по ключам, которые подразделяются на *уникальные* (допускается одно значение для одного ключа) и *равные* (допускается наличие множественных копий значения ключа). Стандартом предусмотрены два контейнера с уникальными ключами:

* *set* (множество);
* *map* (словарь);

А также – два контейнера с равными ключами:

* *multiset* (множество с дубликатами);
* *multimap* (словарь с дубликатами).

Ассоциативные контейнеры представляют собой связанные структуры (деревья) с узлами соответствующего типа *value\_type*. Контейнеры *map* и *multimap* описывают ассоциацию между ключом *Key* и отображаемым типом *T*, т.е. значимый тип *value\_type* для этих контейнеров – *pair<const Key,T>*, в то время как для *set* и *multiset* значимый тип совпадает с типом ключа.

**Итераторы**

***Итератор*** — это объект, предоставляющий доступ к элементам контейнера и позволяющий осуществлять навигацию по контейнеру. Все контейнеры стандартной библиотеки шаблонов предоставляют итераторы, реализуя тем самым механизм, минимизирующий зависимость алгоритмов от структуры данных, с которыми они оперируют.

Модель итератора построена по образцу *указателей* в C/C++. В общем случае к итератору допустимо применять операции инкремента (оператор ++ для перехода к следующему элементу), косвенного доступа (оператор \* для разыменования) и сравнения (операторы == и !=), в некоторых случаях — также операции декремента, указательную арифметику и т.д. Перечень наиболее важных операций над итераторами представлен в таблице 1.

Таблица 1 – Операции над итераторами

| Операция | Описание |
| --- | --- |
| ++p | Префиксная инкрементация: устанавливает итератор на следующий элемент последовательности; результатом является значение *p+1* |
| p++ | Постфиксная инкрементация: устанавливает итератор на следующий элемент последовательности; результатом является значение *p* (до инкрементации) |
| − − p | Префиксная декрементация: устанавливает итератор на предыдущий элемент последовательности; результатом является значение *p-1* |
| p − − | Постфиксная декрементация: устанавливает итератор на предыдущий элемент последовательности; результатом является значение *p* (до инкрементации) |
| \*p | Доступ (разыменовывание): значение *\*p* относится к элементу, на который указывает итератор *p* |
| p→m | Доступ (доступ к члену); эквивалент выражения *(\*p).m* |
| p[n] | Доступ (индексирование); значение *p[n]* относится к элементу, на который указывает итератор *p + n*; эквивалент выражения  *\*(p + n)* |
| p == q | Равенство: истина, если итераторы *p* и *q* указывают на один и тот же элемент или оба указывают на элемент, следующий за последним |
| p != q | Неравенство: *!( p == q)* |
| p < q  (p > q) | Указывает ли итератор *p* на элемент, расположенный до (после) элемента, на который указывает итератор *q*? |
| p <= q | p < q || p == q |
| p >= q | p > q || p == q |
| p += n  (p − = n) | Вперед на *n* элементов: устанавливает итератор *p* на *n*-й элемент, считая вперед (назад) от элемента, на который он ссылается в данный момент |
| q = p + n  (q = p − n) | Итератор *q* ссылается на *n*-й элемент, считая вперед (назад) от элемента, на который ссылается итератор *p* |
| advance (p, n) | Перемещение вперед: аналог выражения *p += n.* Функцию *advance()* можно использовать, даже если *p* не является итератором произвольного доступа |
| x = difference (p, q) | Разность: аналог выражения *q − p*. Функцию *difference()* можно использовать, даже если *p* не является итератором произвольного доступа |

Не над каждым видом итераторов можно совершать операции, перечисленные в таблице 1. Набор поддерживаемых итератором операций определяют его тип. В стандартной библиотеке шаблонов определено пять типов итераторов: итератор *ввода,* итератор *вывода, однонаправленный* итератор, *двунаправленный* итератор и итератор с *произвольным* доступом.

***Итераторы ввода*** (*Input iterators*) являются наиболее простым типом итераторов, который указывает на область памяти, из которой *считываются* данные, и поддерживает ограниченный набор операций. Итератор ввода обладает конструктором по умолчанию, а также конструктором копирования:

. . .

X u; //конструктор по умолчанию

X u(a); //конструктор копирования

X u1;

u1 = u; // применение оператора копирующего присваивания

Кроме того, для него определены операция копирующего присваивания, операция инкремента (можно перемещаться вперед по контейнеру), отношение эквивалентности (можно сравнивать два экземпляра итератора с помощью операторов == и !=) и разыменовывания (можно единожды считать каждый элемент контейнера с помощью оператора \*):

X u\_start = . . . ; // получить начало диапазона, допускающего чтение

X u\_end = . . . ; // получить конец диапазона, допускающего чтение

for(; u\_start != u\_end ; ++u\_start )

{

std::cout << \*u\_start; //Считать элемент, на который указывает u\_start

}

Особенность итераторов данного типа заключается в том, что алгоритмы с итераторами ввода должны быть *однопроходными* (*single pass*), т.е. никогда не должны пытаться проходить через один и тот же итератор дважды:

. . .

X u2 = u1;

assert(u1 == u2); // корректно

++u1;

++u2;

assert(u1 == u2); // не соответствует single pass алгоритму

***Итераторы вывода*** (*Output iterators*) также поддерживают конструктор по умолчанию, конструктор копирования, выполняют операции копирующего присваивания, инкремента и сравнения. Алгоритмы, работающие с итераторами вывода, также должны быть однопроходными.

Ключевым отличием от рассмотренных выше *итераторов ввода* является операция *разыменовывания*, возвращающая в этом случае *l-значение*, которое может находиться только в левой части оператора присваивания. Другими словами, итератор вывода ссылается на область памяти, в которую *записываются* данные, и запись каждого элемента осуществляется единожды с помощью оператора \*:

X u\_start = . . . ; // получить начало диапазона, допускающего запись

X u\_end = . . . ; // получить конец диапазона, допускающего запись

for(; u\_start != u\_end ; ++u\_start )

{

\*u\_start = V (); //Записать значение, возвращенное V(), в \*u\_start

}

***Однонаправленные итераторы*** (*Forward iterators*) объединяют в себе свойства итераторов ввода и вывода (т.е. поддерживают как чтение, так и запись элемента, на который ссылается итератор) и, кроме того, могут применяться не только в однопроходных алгоритмах (к примеру, допускается многократное разыменовывание). Кроме того, если однонаправленный итератор указывает на объект класса, то для доступа к его члену можно использовать итератор →.

***Двунаправленные итераторы*** *(Bidirectional iterators)* включает все требования к однонаправленным итераторам и еще добавляет операцию декремента, позволяющую выполнять обход диапазона в обратном порядке (используя оператор − −).

***Итераторы с произвольным доступом*** *(Randomaccess iterators)* предоставляют широкий спектр операций: позволяют перемещаться вперед (с помощью операторов ++ и + =) и назад (с помощью операторов − − и − =), а также считывать и записывать не константные элементы с помощью оператора \* или []. К итераторам произвольного доступа допустимо применение индексации, добавление и вычитание произвольного целого числа (с помощью операторов + и − ), вычисление расстояния между двумя экземплярами итераторов, установленными на одну и ту же последовательность и их сравнение (с помощью операторов <, <=, >, >=).

Наборы операций каждого упомянутого типа итератора надстраиваются один над другим, образуя иерархию, показанную на рисунке 2.

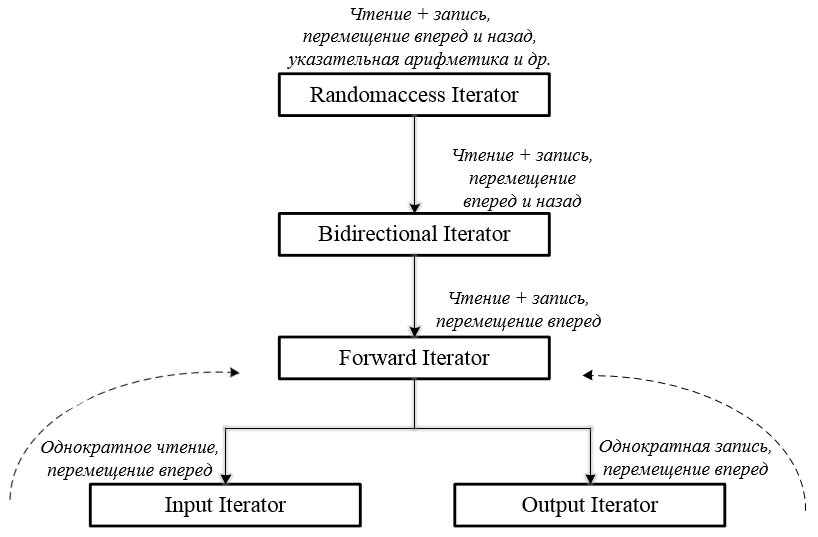


Рисунок 2 – Иерархия итераторов

Каждый контейнер предоставляет свой итератор, принадлежащий тому или иному типу:

Таблица 1 – Соответствие между контейнером и типом итератора

| Контейнер | Тип итератора |
| --- | --- |
| *deque* | Итераторы произвольного доступа |
| *list* | Двунаправленные итераторы |
| *vector* | Итераторы произвольного доступа |
| *set* | Двунаправленные итераторы |
| *multiset* | Двунаправленные итераторы |
| *map* | Двунаправленные итераторы |
| *multimap* | Двунаправленные итераторы |

Однонаправленные итераторы применяются в работе с такими контейнерами, как *unordered\_set, unordered\_map, unordered\_multiset, unordered\_map*.

**Адаптеры итераторов**

Помимо пяти категорий итераторов стандарт также определяет несколько адаптеров операторов. Первый из них – ***reverse\_iterator*** – адаптер класса, применяемый к типу *двунаправленного* итератора или итератора с *произвольным доступом* для определения соответствующего типа обратного итератора, который выполняет итерации через структуру данных в противоположном направлении.

Кроме того, стандартная библиотека шаблонов определяет несколько адаптеров экземпляра для итераторов: ***back\_insert\_iterator****,* ***front\_insert\_iterator***и***insert\_iterator*** , которые позволяют осуществлять вставку элементов в контейнер путем вызова методов *push\_back()*, *push\_front()* и *insert()* соответственно.

Сравним два случая:

while (first != last) \*result++ = \*first++; //*result* - обычный итератор, допускающий запись

while (first != last) \*result++ = \*first++; //*result* - итератор вставки

В первом случае произойдет копирование диапазона *[first, last)* в диапазон, начинающийся с *result*. Во втором – элементы диапазона *[first, last)* будут вставлены в контейнер, а не записаны поверх имеющихся. Такой механизм позволяет всем алгоритмам копирования в библиотеке работать в режиме вставки (*insert mode*) вместо обычного режима *наложения* записей.

Следующий вид стандартных адаптеров экземпляра – это *потоковые итераторы*, которые адаптируют потоки ввода и вывода и буферы потоков, представляя их в виде итераторов ввода и вывода: *istream\_iterator*, *ostream\_iterator*, которые можно использовать, например, для чтения набора значений из стандартного потока ввода *cin* (с применением итератора вставки)и для записи этого набора после сортировки в строковый поток:

std::vector<int> values;

std::stringstream sstm;

// Прочитать значения

std::copy(std::istream\_iterator<int>(std::cin), std::istream\_iterator<int>(), std::back\_inserter(values));

// Отсортировать значения

std::sort(values.begin(), values.end());

// Вывести значения

std::copy(values.begin(), values.end(), std::ostream\_iterator<int>(sstm, " "));

**Определение собственного итератора**

Рассмотрим простейший контейнер, инкапсулирующий массив фиксированного размера, содержащий целые числа. Для работы с собственным итератором в данном классе необходимо определить (как минимум) тип *iterator*, а также методы *begin* (возвращает итератор, указывающий на первый элемент контейнера) и *end* (возвращает итератор, указывающий на элемент, следующий за последним элементов контейнера):

class LiteContainer

{

public:

typedef LiteContainer <int> iterator;

LiteContainer (std::initializer\_list<int> values);

iterator begin() {return iterator(data.get());}

iterator end() {return iterator(data.get() + size);}

/\*аналогично можно объявить:

typedef LiteContainer <const int> const\_iterator;

const\_iterator begin() const;

const\_iterator end() const;

и т.д.\*/

private:

const size\_t size;

std::unique\_ptr<int[]> data;

};

Реализация класса итератора осуществляется на базе шаблона *iterator*, имеющего следующий вид:

template<typename \_Category, typename \_Tp, typename \_Distance = ptrdiff\_t,

typename \_Pointer = \_Tp\*, typename \_Reference = \_Tp&>

struct iterator

{

// One of the iterator\_tags tag types

typedef \_Category iterator\_category;

// The type "pointed to" by the iterator

typedef \_Tp value\_type;

// Distance between iterators is represented as this type

typedef \_Distance difference\_type;

// This type represents a pointer-to-value\_type

typedef \_Pointer pointer;

// This type represents a reference-to-value\_type

typedef \_Reference reference;

};

Шаблонный класс итератора имеет пять параметров (при наследовании допустимо указывать только первые два параметра):

* тип итератора, который может принимать следующие значения:
  + - *input\_iterator\_tag;*
    - *output\_iterator\_tag;*
    - *forward\_iterator\_tag;*
    - *bidirectional\_iterator\_tag;*
    - *random\_access\_iterator\_tag;*
* тип, возвращаемый при разыменовывании итератора;
* тип, описывающий «расстояние» между итераторами, указывающими на элементы одной последовательности;
* тип указателя на значение элемента, на который ссылается итератор;
* тип ссылки на значение элемента, на который указывает итератор.

Определим пользовательский класс итератора, имеющего тип *итератора ввода*. Объявим класс контейнера как дружественный, и в качестве параметра конструктора по умолчанию будем передавать указатель на элемент контейнера:

template<typename ValueType>

class LiteIterator: public std::iterator<std::input\_iterator\_tag, ValueType>

{

friend class LiteContainer;

private:

LiteIterator (ValueType\* p) : p(p) { }

public:

LiteIterator (const LiteIterator &it) : p(it.p) { }

bool operator!=( LiteIterator const& other) const;

bool operator==( LiteIterator const& other) const;

typename LiteIterator::reference operator\*() const;

LiteIterator & operator++();

private:

ValueType\* p;

};

**Алгоритмы**

***Алгоритмы*** – это обобщенные шаблоны функций, которые применяются к диапазону элементов внутри контейнера и выполняют операции инициализации, сортировки, поиска, преобразования элементов диапазона и т.д. Диапазон, над которым работает алгоритм, задается парой итераторов.

При копировании, сравнении и выполнении других операций над двумя контейнерами первый задается парой итераторов *[b:e)*, а второй – только одним итератором *b2*, который считается началом последовательности, содержащей элементы в достаточном для выполнения алгоритма количестве (например, столько же, сколько элементов в первой последовательности: *[b2:b2+(a-b)]*).

Алгоритмы могут использовать *итераторы произвольного доступа* (как это делает, например, *sort*), так *однонаправленные итераторы* (характерно для большинства алгоритмов, в т.ч. для *find*).

Алгоритмы, согласно своему назначению, подразделяются на следующие группы:

* не меняющие последовательность операции (*Non-mutating sequence operations*): поиск (*find*), подсчет (*count*), отличие (*mismatch*), сравнение на равенство (*equal*), поиск подпоследовательности (*search*) и т.д.;
* меняющие последовательность операции (*Mutating sequence operations*): копирование (*copy*), обмен (*swap*), замена (*replace*), удаление (*remove*), исключение повторов (*unique*) и т.д.;
* операции сортировки и отношения (*Sorting and related operations*): сортировка (*sort*, *stable\_sort*, *partial\_sort*, *partial\_sort\_copy*), объединение (*merge*, *inplace\_merge*);
* обобщённые численные операции (*Generalized numeric operations*) представляют собой основные операции над множеством для сортированных структур (*includes, set\_union, set\_difference* и т.д.).

**Порядок выполнения работы**

1. Изучить предлагаемый теоретический материал.
2. По результатам лабораторной работы 1 выбрать оптимальный тип контейнера для реализации.
3. Изучить требования стандарта по отношению к контейнеру, который необходимо реализовать.
4. Разработать шаблонный класс С++ для указанного контейнера (приветствуется максимально возможное соответствие стандарту C++11 и выше).
5. Разработать стандартный итератор, соответствующий контейнеру (см. таблицу 2) согласно требованиям стандарта.
6. Разработать дополнительный итератор-адаптер (по выбору студента).
7. Разработать набор демонстрационных тестов для проверки функционала контейнера (должна демонстрироваться работа c разными типами данных, операторов, конструкторов, операций над контейнером и итераторов).
8. По результатам лабораторной работы 1 делегировать часть бизнес-логики на разработанный контейнер.

*Пример: если в логике программы предполагается выполнять сортировку над коллекцией данных, то можно реализовать контейнер, который при добавлении элементов будет производить вставку в нужное место коллекции.*

1. Дополнить UML-схему из лабораторной работы 1 с учетом разработанного класса контейнера.

**Содержание отчета**

1. Обоснование выбранного типа контейнера.
2. Описание бизнес-логики делегированной на функционал контейнера.
3. Требования STL к разработанному типу контейнера.
4. Причины невозможности удовлетворить ряду требований стандарта.
5. Исходный код разработанной программы.
6. UML-схема разработанной программы

**Использованные материалы**

1. *Страуструп Б.* Программирование. Принципы и практика использования C++.: Пер с англ. – М.: ООО “И.Д. Вильямс”, 2014.

**Дополнительная литература**

1. *Уилсон М.* Расширение библиотеки STL для С++: Пер. с англ. Слинкина

А. А. – М.: ДМК Пресс, СПб, БХВ-Петербург, 2008.

2. *Павловская Т., Щупак Ю.* C/C++. Структурное и объектно-ориентированное программирование. Практикум – M.: ООО Издательство «Питер», 2010.

3. *Мэйерс С.* C/C++. Эффективное использование C++. 55 верных советов улучшить структуру и код ваших программ – M.: ДМК Пресс, 2014.

4. *Саттер Г., Александреску А.* Стандарты программирования на С++ – M.: Издательство «Вильямс», 2019.

5. *Stepanov A.* Notes on programming: <http://stepanovpapers.com/notes.pdf>