Программирование на языке C++

Вводный курс

Александр Морозов
gelu.speculum@gmail.com

ИТМО, весенний семестр 2020

Содержание

Коллекции

Коллекция – набор объектов одного типа (элементов), допускающий как операции со всей коллекцией в целом, так и с отдельными элементами.
Другое название – контейнер.

Экземпляр коллекции хранит элементы внутри себя, копирование или перемещение экземпляра коллекции приводит к копированию или перемещению всех её элементов.

Некоторые свойства коллекций

Общие:

- хранение множества элементов одного типа
- последовательный доступ ко всем элементам
- операции над элементами: немодифицирующий доступ (чтение), модифицирующий доступ (запись), удаление и добавление

Различающиеся:

- эффективность реализации тех или иных операций
- доступ к элементам в определенном порядке
- некоторые специфические операции

Понятие вычислительной сложности

- вычислительные затраты на обработку
- затраты памяти на представление или обработку

$$f(n) \in O(g(N))$$

- $O(1)$
- $O(log(N))$
- $O(N)$
- $O(N^2)$
- $O(2^N)$
- $O(N!)$

Стандартные коллекции: простые последовательности

- `std::array` – массив элементов со статическим размером
- `std::vector` – массив элементов с динамическим размером
- `std::deque` – двусторонняя очередь
- `std::forward_list` – односвязный список
- `std::list` – двусвязный список

Стандартные коллекции: упорядоченные множества

- `std::set` – упорядоченное множество
- `std::map` – упорядоченный ассоциативный массив
- `std::multiset` – упорядоченное мультимножество
- `std::multimap` – упорядоченный ассоциативный массив с повторением ключей

Стандартные коллекции: неупорядоченные множества

- `std::unordered_set` – неупорядоченное множество
- `std::unordered_map` – неупорядоченный ассоциативный массив
- `std::unordered_multiset` – неупорядоченное мультимножество
- `std::unordered_multimap` – неупорядоченный ассоциативный массив с повторением ключей

Адаптеры коллекций

- `std::stack` – LIFO
- `std::queue` – FIFO
- `std::priority_queue` – приоритетная очередь, быстрый доступ к наибольшему элементу

Внутреннее представление

- непрерывная область памяти – `std::array`, `std::vector`
- набор независимых подобластей – `std::deque`
- каждый элемент в отдельном узле списка – `std::forward_list`, `std::list`
- дерево – `std::set`, `std::map`, `std::multiset`, `std::multimap`
- хеш-массив – `std::unordered_set`, `std::unordered_map`, `std::unordered_multiset`, `std::unordered_multimap`

Сложность некоторых операций

	vector	deque	list	map	unordered_map
fill	$O(N)$	$O(N)$	$O(N)$	$O(N \cdot log(N))$	$O(N)$
add	$O(N)$	$O(1)$	$O(1)$	$O(log(N))$	$O(1)/O(N)$
insert	$O(N)$	$O(N)$	$O(1)$	$O(log(N))$	$O(1)/O(N)$
at	$O(1)$	$O(1)$	$O(N)$	$O(log(N))$	$O(1)/O(N)$

Содержание

Итераторы

Итератор – абстракция “указателя” на элемент некоторой последовательности.

Тип итератора может быть связан с конкретным типом коллекции/контейнера, но бывают и иные итераторы. Реализация скрывает в себе подробности доступа к указываемому элементу коллекции и связь с “соседними” элементами.

Каждая коллекция задает некоторый порядок на множестве элементов, если представить это как массив в памяти, итераторы имитируют указатели на элементы этого массива.

Итераторы и указатели: сходство

- Разыменование для доступа к элементу (операторы * и ->)
- Равенство/неравенство
- Инкремент для смещения к следующему элементу
- Итератор может указывать на “после последнего элемента” коллекции
- Некоторые итераторы можно декрементировать для смещения к предыдущему элементу
- Над некоторыми итераторами возможна арифметика, как с указателями
- Некоторые итераторы поддерживают доступ по индексу (аналогично указателям)
- Некоторые итераторы можно сравнивать </>

Итераторы и указатели: различия

Указатель удовлетворяет “концепции итератора”, но итераторы коллекций от указателей отличаются.

- Является сложным типом (классом)
- Эффективность операций зависит от конкретной реализации
- Всегда связан с конкретным объектом коллекции или иной сущности
- Некоторые действия над объектом коллекции могут инвалидировать итератор
- Для итераторов не работает приведение типов, возможное для указателей

Концепция: итератор

- Объект можно копировать
- Определен оператор $*$
- Определен оператор префиксного инкремента (условие – итератор должен быть указывать на элемент коллекции; результатом будет либо итератор, указывающий на следующий элемент, либо на после последнего)

Концепция: итератор на чтение (input)

- Удовлетворяет требованиям итератора
- Объекты можно сравнивать на равенство/неравенство
- Определен оператор ->
- Определен оператор постфиксного инкремента
- Не гарантирована валидность копий итератора после его инкремента

Таким образом, не гарантирована возможность многократного прохода по элементам, только однократного.

Концепция: итератор на запись (output)

- Удовлетворяет требованиям итератора
- Определен оператор постфиксного инкремента
- Валидно выражение `*i = x` где `i` – объект итератора, а `o` некоторое значение; после выполнения этого выражение не гарантируется разыменуемость итератора или валидность его предыдущих копий
- Не гарантирована валидность копий итератора после его инкремента
- Операцию разыменования допустимо использовать только для присвоения значения

Таким образом, не гарантирована возможность многократного прохода по элементам, только однократного.

Концепция: мутабельный итератор (input and output)

Если итератор удовлетворяет обоим концепциям – input и output, то его называют mutable, это значит, что указываемые значения можно читать, а можно присваивать им другие значения.

Концепция: итератор непрерывной области (contiguous)

- Удовлетворяет требованиям итератора
- Элементы коллекции, с которой связан итератор, размещены в непрерывной области в памяти и в том порядке, в котором осуществляется обход с помощью итераторов
- Валидна операция `*(a + n)` и эквивалентна `*(&(*a)+ n)`

Концепция: итератор прямого обхода (forward)

- Удовлетворяет требованиям итератора на чтение
- `a == b` $\Leftrightarrow$ `&(*a)== &(*b)` (либо оба не разыменуются)
- Инкремент итератора не меняет валидность его копий
- Инкремент итератора не меняет результат разыменования его копий
- Равенство итераторов гарантирует равенство результатов их инкремента
- Если итератор является мутабельным, то присвоение через итератор не меняет его валидность или валидность его копий

Таким образом, гарантирована возможность многократного прохода по элементам.

Концепция: итератор двустороннего обхода (bidirectional)

- Удовлетворяет требованиям итератора прямого обхода
- Определен оператор декремента (префиксного и постфиксного)
- Декремент не меняет валидность копий итератора
- Результат декремента итератора, указывающего на первый элемент коллекции не определен

Концепция: итератор произвольного доступа (random access)

- Удовлетворяет требованиям двунаправленного итератора
- Поддерживает арифметику, аналогичную арифметике указателей
- Определен оператор доступа по индексу, подобно как для указателей
- Определены операторы сравнения <, >, <=, >=

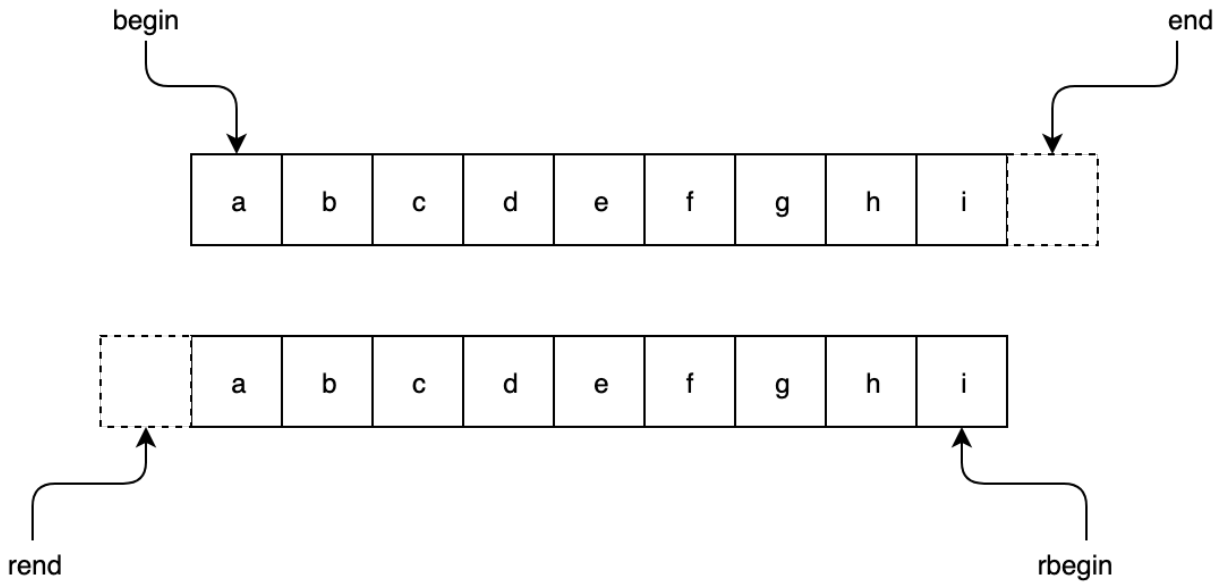
Примеры концепций

- Указатель на элемент массива – удовлетворяет требованиям итератора произвольного доступа
- Итераторы коллекций `array`, `vector`, `deque` – произвольный доступ
- Итераторы `list`, `set`, `map` – двунаправленный
- Итераторы `forward_list`, `unordered_set`, `unordered_map` – прямого обхода
- `back_insert_iterator` – итератор на запись, добавляет элементы в конец коллекции

Некоторые операции над итераторами

```
#include <iterator>

void std::advance(It & it, Distance distance);

Distance std::distance(It from, It to);

It std::next(It it, Distance distance);

It std::prev(It it, Distance distance);
```

Обход коллекций




```
std::begin()
std::end()
std::cbegin()
std::cend()

std::rbegin()
std::rend()
std::crbegin()
std::crend()
```

Некоторые адаптеры итераторов

```
RIt std::make_reverse_iterator(It it);

It std::front_inserter(Container & c);

It std::back_inserter(Container & c);

It std::inserter(Container & c, It it);
```

Примеры

```
std::vector<int> v {1, 2, 3, 4, 5};
std::deque<double> d;
auto it = std::front_inserter(d);
for (auto i : v) {
    *it++ = i;
}
std::copy(d.begin(), d.end(),
    std::ostream_iterator<double>(std::cout, ", "));
```

```
5, 4, 3, 2, 1,
```

Содержание

Алгоритмы стандартной библиотеки

Алгоритм – некоторая функция, принимающая пару итераторов, задающих начало и конец последовательности и, возможно, другие аргументы.
Оперирует элементами последовательности.

```
std::vector<int> v { 4, 5, 3, 1, 2 };
std::sort(v.begin(), v.end() - 1);
std::copy(v.begin(), v.end(),
    std::ostream_iterator<int>(std::cout, ",␣"));
```

```
1, 3, 4, 5, 2,
```

Типы алгоритмов

- Не модифицирующие (`any_of`, `for_each`, `count`, `find`)
- Разбивающие (`partition`)
- Сортирующие (`sort`, `nth_element`)
- Общие модифицирующие (`copy`, `transform`, `remove`, `unique`)
- Операции над сортированными последовательностями (`binary_search`, `merge`, `set_intersection`)
- И другие...

Пример модифицирующего алгоритма

```
std::vector<int> v { 2, 9, 10, 7, 5, 1, 3, 4, 8, 6 };

v.erase(std::remove_if(v.begin(), v.end(),
                       [] (const auto x) { return x < 5; }),
        v.end());

std::copy(v.begin(), v.end(),
        std::ostream_iterator<int>(std::cout, ",␣"));

9, 10, 7, 5, 8, 6,
```