Объектно-ориентированное программирование

Стандартная библиотека C++

Вопросы

- Что такое абстрактный класс?
- Что такое шаблон класса?
- Что такое развертывание стека?
- Можно ли унаследовать один класс исключения от другого?
- Что означает конструкция catch (…)?

Стандартная библиотека C++

Состав стандартной библиотеки C++

STL

- Потоковые классы
- Строковый класс
- Контейнерные классы – хранение данных
- Итераторы – доступ к элементам контейнерных классов
- Математические классы-
  обработка массивов и комплексных чисел
- Диагностические классы – идентификация типов данных

Контейнерные классы стандартной библиотеки

- Стандартная библиотека шаблонов (standard template library – STL)
- **Контейнерные классы —** это классы, предназначенные для хранения данных, организованных определенным образом.
- STL содержит контейнеры, реализующие основные структуры данных, используемые при написании программ — ***векторы, двусторонние очереди, списки и их разновидности, словари и множества.***

Классификация контейнеров

- Контейнеры можно разделить на два типа; **последовательные** и **ассоциативные**.
- ***Последовательные контейнеры*** обеспечивают хранение конечного количества однотипных величин в виде непрерывной последовательности.
  - векторы – класс **vector**
  - двусторонние очереди – класс **deque**
  - списки - класс **list**
  - адаптеры:
    - стеки (**stack)**
    - очереди (**queue)**
    - очереди с приоритетами (**priority_queue)**.

Классификация контейнеров

- ***Ассоциативные контейнеры*** **обеспечивают быстрый доступ к данным по ключу**.

- Построены на основе сбалансированных деревьев.

  - словари (**map**),
  - словари с дубликатами (**multimap**),
  - множества (**set**),
  - множества с дубликатами (**multiset**)
  - битовые множества (**bitset**).

Общие операции с контейнерами

Общие методы для работы контейнеров

- Контейнеры обеспечивают единый набор методов для работы:
  - Конструкторы, деструкторы
  - Присваивание, операторы сравнения
  - swap – обмен данными между разными контейнерами
  - empty
  - max_size
  - size

Стандартные контейнерные методы

typedef	Пояснение
value_type	Тип элементов контейнера
size_type	Тип индексов, счетчиков элементов
iterator	Итератор
const_iterator	Константный итератор
reverse_iterator	Обратный итератор
const_reverse_iterator	Константный обратный итератор
reference	Ссылка на элемент
const_reference	Константная ссылка на элемент
key_type	Тип ключа (для ассоциативных контейнеров)
key_compare	Тип критерия сравнения (для ассоциативных контейнеров)

Итераторы

- Для обработки информации в контейнерах используются итераторы
- ***Итератор*** **–** это специальный класс, который является аналогом указателя на элемент. Он используется для просмотра контейнера в прямом или обратном направлении:
  - ***** возвращает значение,
  - **++** переводит на следующий элемент контейнера
  - **`container.begin()`** возвращает итератор, указывающий на начало контейнера
  - **`container.end()`** возвращает итератор, указывающий на конец контейнера
- Константный итератор не дает изменять значения контейнера

```
vector<int>::const_iterator cit = v.begin();
*cit = 24; // does not compile
```

Пример работы с итератором

```
using namespace std;

string str1 ( "No way out." ) ;
basic_string <char>::iterator str_Iter, str1_Iter;
basic_string <char>::const_iterator str1_cIter;

str1_Iter = str1.end ( );
str1_Iter--;
str1_Iter--;
cout << "Last character of str1:" << *str1_Iter << endl;

// end used to test when an iterator has reached the end of its
string
cout << "The string is now: ";

for ( str_Iter = str1.begin( );
      str_Iter != str1.end( );
      str_Iter++ )
      cout << *str_Iter;

cout << endl;
```

Методы работы с итераторами

Метод	Пояснение
iterator begin() const_iterator begin() const	Указывает на первый элемент
iterator end() const_iterator end() const	Указывает на элемент, следующий за последним
reverse_iterator rbegin() const_reverse_iterator rbegin() const	Указывает на первый элемент в обратной последовательности
reverse_iterator rend() const_reverse_iterator rend() const	Указывает на элемент, следующий за последним в обратной последовательности

Последовательные контейнеры

Вектор

- **Вектор** – это структура, основанная на массивах, реализующая произвольный доступ к элементам, добавление в конец, удаление из конца

  - #include <vector>
  - структура данных с последовательным размещением в памяти
  - для доступа используется оператор **[ ]**
  - Используется в случае, если данные необходимо сортировать и быстро получать доступ к любому элементу
  - Когда блок выделенной памяти заканчивается
    - выделяет увеличенный блок памяти
    - копирует себя в него
    - очищает устаревшую память

Описание вектора

- Определение
  - `vector <type> v;`
  - `vector <type> v(7812);`
  - ```
    template <class It>
    vector<type> v(It begin, It end);
    ```
  - *type* - **int, float, Point, и д.п.**
  - Опасайтесь вектора **vector<bool>**
- Iterators:

```
vector<type>::const_iterator it;
vector<type>::iterator it = v.begin();

*(it1 + 5) = 34;
```
```

Методы работы с вектором

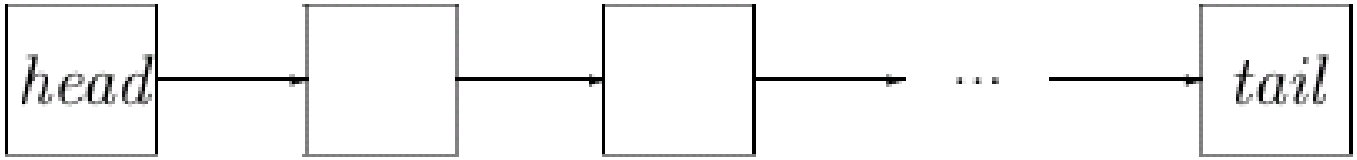
- Функции **vector** для объекта **v**

v.push_back(value) – добавить элемент в конец
v.size() – размер вектора
v.capacity() – какой объем доступен для хранения
v.reserve(n) – зарезервировать объем для хранения
v.insert(*pointer*, *value* **)** – добавить *value* до *pointer.*
v.erase(*pointer* **)** – удалить элемент из контейнера
v.erase(*pointer1*, *pointer2* **)** – удалить элементы начиная с *pointer1* и до (не включая) *pointer2*.
v.clear()
v[*elementNumber***] = value;**
v.at[*elementNumber***] = value;**

- Присвоить значение элементу с проверкой диапазона
- throws **out_of_bounds** exception

Список

- ***Список*** реализует вставку и удаление элементов в произвольное место, но не имеет произвольного доступа к своим элементам.



- Эффективная вставка/удаление элементов в любом месте контейнера
- Двусвязный список
- Двунаправленные итераторы

Методы работы со списками

`listObject.sort()`
- сортировка в возрастающем порядке

`listObject.splice(iterator, otherObject);`
- добавить значения **otherObject** до **iterator**

`listObject.merge(otherObject)`
- удаляет `otherObject` вставляет его в `listObject`

`listObject.unique()`
- удаляет дубликаты элементов в списке

`listObject.swap(otherObject);`
- обмен содержимым списков

`listObject.remove(value)`
- удаляет все вхождения `value`

Двусторонняя учередь

- **Двусторонняя очередь** реализует произвольный доступ к элементам, добавление в оба конца и удаление из обоих концов.

  - #include **<deque>**
  - доступ посредством **[ ]**
  - эффективное добавление в конец и начало
  - не последовательная область памяти: “умные” указатели
  - такие же операции как **vector** + **push_front** / **pop_front**

Последовательные контейнеры

Операция	**Метод**	**vector**	**deque**	**list**
Вставка в начало	push_front	–	+	+
Удаление из начала	pop_front	–	+	+
Вставка в конец	push_back	+	+	+
Удаление из конца	pop_back	+	+	+
Вставка в произвольное место	insert	+	+	+
Удаление из произвольного места	erase	+	+	+
Произвольный доступ к элементу	[].at	+	+	–

Ассоциативные контейнеры

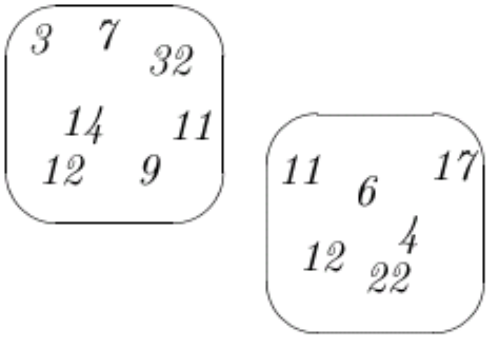
Хранение элементов «ключ-значение»

- Ассоциативные контейнеры обеспечивают прямой доступ к значению элемента по ключу
- 4 типа: **multiset**, **set**, **multimap**, **map**
  - **multiset** и **multimap** допускают хранение дублирующих ключей
  - **multimap**, **map** позволяют хранить ключи и ассоциированные значения
- Для хранения пары “ключ—элемент” используется шаблон **pair**, описанный в заголовочном файле <utility>:

```
template <class T1, class T2> struct pair{
     typedef T1 first_type;
     typedef T2 second_type;
     T1 first;
     T2 second;
     pair(const T1& x, const T2& y);
     template <class U, class V> pair(const pair<U, V> &p);
};
```

Множества

- **`set, multiset`** – быстрое хранение, получение ключей



- Логарифмические операции добавления, удаления, проверки вхождения
- Эффективные операции конъюнкции, дизъюнкции, вычитания множеств
- Хранение в порядке по возрастанию

Операции со множествами

- Методы для объекта типа **multiset**
  - **msObject.insert(***value***)** – добавить элемент
  - **msObject.find(***value***)** – возвращает итератор на первое вхождение *value*, или *msObject.end()*
  - **msObject.lower_bound(***value***)** – возвращает итератор на первое вхождение *value*
  - **msObject.upper_bound(***value***)** – возвращает итератор на место после последнего вхождения *value*
  - для объекта **p** типа **pair**

    p = msObject.equal_range(*value***)**

    - устанавливает элементы **first** и **second** в паре на **lower_bound** и **upper_bound** для определенной *value*

Словари

- Коллекция пар «ключ – значение»
- Ключи могут быть любого упорядоченного типа (строки, числа и т.п.)
- Значения могут быть любого типа
- Эффективные операции вставки, удаления, тестирования на вхождение
- **`#include <map>`**
- one-to-one mapping (duplicates ignored)
  - используется `[]` для доступа к значениям

$$
\begin{aligned}
key_1 &\rightarrow value_1 \\
key_2 &\rightarrow value_2 \\
key_3 &\rightarrow value_3 \\
&\dots \\
key_n &\rightarrow value_n
\end{aligned}
$$

```
map<string, double> M;
M[“test”] = 4000.21;
```

Выбор контейнера

Время исполнения типовых операций

Контейнер	Добавление элемента	Удаление элемента из середины	Проверка вхождения
Вектор	O(1) или O(n)	O(1) или O(n)	O(n) или O(log n)
Список	O(1)	O(1)	O(n)
Дек	O(1)	O(n)	O(n) или O(log n)
Стек	O(1)	NA	NA
Очередь	O(1)	NA	NA
Множество	O(log n)	O(log n)	O(log n)
Словарь	O(log n)	O(log n)	O(log n)

Какой контейнер выбрать?

- Какой требуется доступ к элементам?
  - Случайный – vector или deque
  - Ассоциативный– set или map
  - Последовательный – list
- Важен ли порядок хранения элементов в контейнере?
  - Ассоциативный – set or map
  - Могут быть отсортированы – vector or deque
  - Важно время добавления – stack or queue

Какой контейнер выбрать?

- Будет ли размер структуры значительно варьироваться в процессе работы?
  - Да – list или set
  - Нет – vector or deque
- Возможно ли оценить размер коллекции?
  - Да – vector

Какой контейнер выбрать?

- Часто ли требуется узнать, есть ли требуемый элемент в коллекции?
  - Да – set
- Требуется ли индексированный доступ?
  - Целочисленный индекс – vector or deque
  - Индекс любого упорядоченного типа– map

Какой контейнер выбрать?

- Требуется ли, чтобы элементы были сравнимы?
  - Set требует чтобы элементы были сравнимы
  - Vector и list не требуют поддержки операций сравнения
- Куда чаще всего вставляются элементы коллекции?
  - Середина – list
  - Конец – stack или queue