C++ второй семестр

Купоросов Василий, Недиков Константин, Казначеев Дмитрий

26 марта 2017 г.

Содержание

1. Шаблоны

1.1. Шаблонные классы

Пусть мы написали свой класс, хранящий внутри себя объекты какого-то типа. Раньше эти типы фиксировались во врем написания кода.

```
1 class MyArray {
2 private:
3   int *array;
4 };
5
6 class Scoped_ptr {
7   GaussNumber *ptr;
8 };
```

Но иногда нам хочется завести вектор или умный указатель другого типа. Приходится вручную переписывать весь класс. Как такую проблему решать в разных стилях написания кода?

1.1.1. C-style

Используем препроцессор. Обратите внимание на слэши в конце строк. Они нужны для многострочного дефайна, `##` – для склеивания `MyArray_` и типа, подставленного на место `TYPE`.

`MyArray.h`:

```
1 #define MyArray(TYPE) class "MyArray_" ## TYPE { \
2 private:                                          \
3   size_t size;                                   \
4   TYPE *array;                                   \
5 public:                                          \
6   TYPE get(size_t index) {                       \
7     return array[index];                         \
8   }                                              \
9 };
```

`main.c`:

```
1 #include "MyArray.h"
2 MyArray(int);      // подставляем определение класса с интами
3 MyArray(double);   // подставляем определение класса с даблами
4 int main() {
5   MyArray_int a;
6   MyArray_double b;
7 }
```

Если хочется посмотреть, как изменился код после препроцессора, можно запустить gcc с ключем -E.

Рассмотрим особенности данного подхода.

1) Все методы класса должны быть `inline`, потому что, если класс объявлен в разных файлах, то на стадии линковки будет double definition.

2) Препроцессор человек простой. Видит строку – заменяет её. Поэтому возникают спецэффекты. В приведенном ниже коде название переменной `TYPE` заменится на `int`.

```
#define MyArray(TYPE) . . .
MyArray(int);
MyArray_int arr;
int TYPE;
```

3) Нельзя делать сложные объявления. Следующая строчка у препроцессора получится не лучшим образом.

```
MyArray(MyArray(int)) arr;
```

1.1.2. C++ style – шаблоны

Шаблоны позволяют делать всё то же самое, но на уровне компилятора. Этот стиль называется обобщенное (generic) программирование. Рассмотрим синтаксис.

`MyArray.h`:

```
template <typename T>
class MyArray {
private:
  size_t size;
  T *array;
public:
  T& get(size_t index) {
    return array[index];
  }
  T& MyArray<T>::operator[](size_t index);
  MyArray<T>& MyArray<T>::operator=(const MyArray<T>& obj);
};

template <typename T> // <class T> эквивалентно <typename T>
T& MyArray<T>::operator[](size_t index) {
  return array[index];
}
```

`main.cpp`:

```
#include "MyArray.h"
int main() {
  MyArray<int> a;
  MyArray<double> b;
  MyArray<MyArray<int> > arr; // работает всегда
  MyArray<MyArray<int>> arr;  // только в 11 стандарте
}
```

5 строчка отличается от 6 наличием пробела между закрывающими угловыми скобками. компиляторы c++ до 11 стандарта путают ">>" с оператором сдвига, поэтому лучше всегда ставить пробел.
`T` – тип шаблонных переменных класса, его принято обозначать именно так, но можно как угодно. Запись `template <typename T>` эквивалентна `template <class T>`.

Шаблон гораздо умнее препроцессорной подстановки: он разбирает то, что ему передаётся во время компиляции, поэтому проблем С-подхода не возникнет. Никаких double definition, все счастливы.
По сути происходит вот что: компилятор создает отдельный класс для каждого типа, который мы в программе поставляем в шаблон, получается такой "compile-time полиморфизм".
Важно, что здесь не применима идея раздельной компиляции, так как неизвестно заранее, какой тип захочет пользователь. Поэтому реализацию не выносят в отдельный .cpp файл. Таким образом время компиляции и объём файлов увеличиваются.
Если взять стандартную библиотеку языка с++, то не будет .so (.dll) файлов (в отличие от библиотек языка си). Это потому что в STL всё написано с шаблонами и не компилируется заранее.

1.2. Классы с несколькими шаблонными параметрами

Класс может иметь несколько шаблонных параметров.

```
1 template <typename K, typename V>
2 class TreeItem {
3   K key;
4   V value;
5 }
```

Все методы в шаблонных классах по умолчанию `inline`.

1.3. Шаблонные функции

По традиции реализацию функций и методов класса всё же выносят в отдельный .h файл. Но еще раз напомню, что раздельная компиляция здесь неприменима.

`swap.h`:

```
1 template <typename T>
2 void swap(T& a, T& b);
```

`swap_ind.h`:

```
1 template <typename T>
2 void swap(T& a, T& b) {
3   T t(a);
4   a = b;
5   b = t;
6 }
```

Если можно однозначно вывести типы аргументов, то тип для функции можно не указывать.

```
1 int main() {
2   int a = 2;
3   int b = 3;
4   swap<int> (a, b);
5   swap(a, b) // swap<int> выведется автоматически;
6 }
```

Но это не всегда возможно.

Примеры, когда компилятор не сможет вывести шаблонный параметр функции:

```
template <typename T>
class MyArray {
private:
  size_t size;
  T *array;
public:
  MyArray(size_t s) {
    array = new T[size];
  }
};

MyArray arr(10); // Ошибка. Неизвестен тип хранимых объектов.
MyArray<MyArray<int> > arr(10); // Ошибка. Для внутреннего объекта отсутствует
                                // конструктор по умолчанию.
```

Ещё пример:

```
template <typename T>
bool less(T& a, T& b) {
  return a < b;
}

int a, b;
double c;
less(a, b); // ОК. Компилятор выведет тип int.
less(a, c); // Ошибка. Не понятно int или double.
```

Рассмотрим пример функции с несколькими шаблонными типами.

```
template <typename T, typename V>
void copy(MyArray<T>& a, MyArray<V>& b) { }

MyArray<int> a;
MyArray<double> b;
copy<int, int>(a, b); // int приведется к double.
copy(a, b); // Типы однозначно выводятся.
copy<int, string>(a, b); // Ошибка. int не приведется к string.
```

Характерный пример шаблонной функции – функция сортировки. Рассмотрим три реализации.

1) Процедурное программирование (язык си).

```
typedef int (*compare)(const void *, const void *) func_ptr;
void sort(void *array, size_t n, size_t elem_size, func_ptr cmp);
```

2) ООП.

```
class comparable { //базовый класс для всех сравниваемых типов
  virtual int compare(const comparable *o) const = 0;
};
void sort(comparable **array, size_t n);
```

Чтобы объекты можно было отсортировать, они должны наследоваться от comparable и у них должна быть реализована виртуальная функция `compare()`.
По сути мы сортируем массив указателей, чтобы внутри сортировки независимо от типа можно было всегда смещаться на одно и то же число байтов. Отсюда берется двойной указатель `(comparable **array)`.

Недостатки ООП подхода:

- Виртуальные функции. Динамическое связывание работает дольше.
- Нельзя на халяву отсортировать инты. Приходится городить новый класс.

3) generic programming (шаблоны):

```
template <typename T>
void sort(T *array, size_t n);

GaussNumber a[100];
Sort<GaussNumber>(a);
```

Здесь нет проблем со смещением, потому что код компилируется с уже известными типами. Каждый класс должен иметь оператор <.

Недостатки шаблонного подхода:

- увеличение времени компиляции и работы.
- увеличение размера исполняемого файла (компилятор генерирует функции и классы для каждого типа).
- нельзя заранее скомпилировать.

1.4. Другие виды параметры шаблона

В качестве параметра может быть не тип, а, например, число. Допустим я не хочу выделять массив в куче, поэтому я выделяю массив на стадии компиляции. Но на стадии компиляции размер массива должен быть фиксирован. Пример – битовое множество (Bitset).

```
template <size_t Size>
class Bitset{
private:
  char m[(Size - 1) / 8 + 1];
public:
  bool get(size_t index) { }
};

Bitset <128> b1;
Bitset <7> b2;
```

Когда ещё применяется такая штука? В языке си надо было помнить и передавать размер массива (например для функции сортировки). Чтобы этого избежать, можно создать обёртку над массивом с помощью шаблонов.

1.5. Неинстанциированный шаблон

Можно задавать, например, на чем реализовать стек – на векторе или на списке, передав ему в качестве шаблона имя соответствующего класса.

```
template <typename T, class Container>
class Stack{
private:
  Container c;
public:
  void push(const T& v) {}
};
```

При этом могут возникать проблемы.

```
Stack <int, List<int> > s1; //OK
Stack <double, Vector<int> > s2; //Будет работать, но будет потеря точности
```

Во втором случае мы якобы храним даблы, но на самом деле числа будут складываться в интовый вектор и обрезаться.
Чтобы этого избежать, можно не задавать тип вектора при объявлении, а сделать это внутри класса.

```
template <typename T, template <typename> Container>
class Stack{
private:
  Container<T> c ;
};

Stack <int, List> s1 ; // List<int>
Stack <double, Vector> s2 ; // Vector<double>
```

1.6. Значение по умолчанию

Можно задавать значения шаблона по умолчанию.

```
template <typename T, template <typename> Container = Deque>
class Stack{ ... };
Stack <int> s1; // Deque<int>
```

1.7. Специализация шаблона

Допустим хотим обёртку над массивом любого типа. Но для хранения `bool` можно сделать более оптимальную по памяти реализацию. В этом нам поможет специализация.
В следующем примере массив динамический, поэтому размер в качестве шаблонного параметра не передаётся.
Код для общего случая:

```
template <typename T>
class Array {
private:
  T* a;
...
public:
  Array(size_t size) {
    a = new T[size];
  }
...
};
```

Код специально для `bool`:

```
template <>
class Array <bool> {
private:
  char* a ;
...
public:
  Array(size_t size) {
    a = new char[(size - 1) / 8 + 1];
  }
};
```

Обратите внимание, что в специальной реализации надо заново писать все методы (даже если некоторые их них совпадают с методами для общего случая).

1.8. Частичная специализация

В специализации можно оставить часть параметров шаблонными.
Например, хотим, чтобы массив массивов хранился одним последовательным куском, а не массивом объектов типа `Array`:

```
template <class T>
class Array <Array<T> > {
  T** a;
};
```

Или мы хотим реализовать специализацию `bool` у массива с шаблонным размером.

```
template <size_t N>
class Array<bool, N> {
...
};
```

1.9. Экзотические случаи шаблонов

На досуге можете почитать про использование

- указателя в качестве шаблонной переменной,
- глобальной переменной в качестве шаблонной переменной.

2. Исключения

2.1. Виды ошибок

Виды ошибок:

1. Ошибки по вине программиста. Примеры:

```
char *s = NULL;
size_t l = strlen(s);
Array a(-1);
```

Что с ними делать:

- Лучше выявить на стадии тестирования (`assert`, `unit test`, etc).
- При выполнении идеальной программы их не происходит.
- Библиотека C подобные ошибки не обрабатывает.
- Библиотека C++ – по-разному в разных местах:
  `vector.at(i)` делает, а `vector[i]` не делает.
- Обрабатывать или нет – на усмотрение программиста.

2. Ошибки по вине окружения программы. Примеры:

- Файл не существует.
- Сервер разорвал сетевое соединение.
- Пользователь вместо числа ввел букву.

Что с ними делать:

- Могут произойти и при выполнении идеальной программы.
- Обязательно надо обрабатывать!

2.2. Обработка ошибок

Как же обрабатывать ошибки?

1. Проверить наличие ошибки в потенциально опасных местах (`if`).
2. Освободить ресурсы.

```
delete [] array;
fclose(file);
```

3. Сообщить пользователю и/или вызывающей функции.

```
FILE* f = fopen(''a.txt'', ''r'');
if (f == NULL) {
  printf(''File a.txt not found\n'');
} //или
if (f == NULL) {
  return-1;
}
```

4. Предпринять действия по устранению ошибки (например, не смогли соединиться – попробовали ещё пять раз).

Но как же сообщать пользователю об ошибках?
Рассмотрим наивную реализацию.

```
class GUI_VIew {
  load_config() {
    f = fopen();
    if (f == NULL)
      printf(''Error:...'');
      // или
      fprintf(stderr, ''Error:...'');
  }
}
```

Здесь есть проблема – функция `load_config` взяла в себя слишком много, ибо неизвестно, с каким интерфейсом она работает. Она может быть вызвана в консоли, может в браузере и т.д. Также возможно, если файл не открылся - не возможно продолжать работу и что-либо напечатать.
Как сделать более грамотно?

2.3. Обработка ошибок в C-style

2.3.1. Через возвращаемое значение

В случае ошибки функция может вернуть значение, зарезервированное под тип данной ошибки.

```
int load_config() {
  f = fopen();
  if (f == NULL)
    return -1;
}

r = load_config();
if (r == -1) {
  //обработка ошибки
}
```

Но таким способом мы узнаем слишком мало информации об ошибке.

2.3.2. Через глобальную переменную

Функция может записывать в глобальную переменную информацию о произошедшей ошибке.

```
#include <errno.h>
FILE* fopen(...) {
  if (file not found) {
    errno = 666;
    return NULL;
  }
  if (permission denied) {
    errno = 777;
    return NULL;
  }
}
```

```
12
13 file = fopen(''f.txt'');
14 if (errno == 666) {
15   //обработка ошибки file not found
16 }
17 else if (errno == 777) {
18   //обработка ошибки permission denied
19 }
```

2.4. Недостатки C-style

- Не всегда хватает диапазона возвращаемых значений.
  Например, функция `strtol(''a'')` вернет ноль, как и `strtol(''0'')`.

- Код логики и обработка ошибок перемешаны.

```
r = fread(...);
if (r < ...) {
  //error
}
r = fseek(...);
if ( r != 0 ) {
  //error
}
```

2.5. C++-style, исключения

Суть такая – в любой непонятной ситуации бросай исключение и выходи из функции. Более формально. Если внутри функции произошла ситуация, в которой функция не знает, что её делать, она бросает ”исключение” – объект особого класса. При этом её выполнение прекращается, а исключение передаётся функции, которая её вызвала. Пример:

```
1  class MyException {
2  private:
3    char message[256];
4    // possible fields: filename, line, function name
5  public:
6    const char* get() {
7      return message;
8    }
9  };
10
11 double divide(int a, int b) {
12   if (b == 0) {
13     throw MyException(''Devision by zero'');
14   }
15   return a / b;
16 }
```

Функция divide не знает, как делить на ноль. Когда её просят это сделать, она кидает исключение и со словами ”ой всё!” прекращает выполняться. Как же поймать исключение?

```
1 try {
2   x = divide(c, d);
3 }
4 catch(MyException& e) {
5   std::cout << e.get();
6   //обработка ошибки
```

```
7 }
```

Если внутри блока `try` функция `divide` бросила исключение, оно сразу же ловится, ссылка на него попадает в переменную e и начинается выполнение блока `catch`. Если исключение не было брошено, блок `catch` выполняться не будет.
Бывает такое, что функция, поймавшая исключение может сделать только часть работы по обработке ошибки. В таком случае эта функция может бросить его дальше:

```
1 try {
2   x = divide(c, d);
3 }
4 catch(MyException& e) {
5   std::cout << e.get();
6   throw e;
7 }
```

Мы решили проблему Си-стиля, теперь есть отдельный механизм для возврата значений, а есть механизм для ошибок.

2.6. Stack unwinding

Пусть функция мы из функции main вызвали функцию `g`, а из неё – функцию `f`. Функция `f` бросила исключение. Проследим за его полётом.
Во время выполнения функции `f` стек выглядел так:

f()
g()
main()

Далее происходит следующее:

1. Нормальный процесс выполнения программы заканчивается, т.е. поток управления до `printf` внутри `f` не дойдет.
2. Начинается stack unwinding: последовательный просмотр стека до тех пор, пока не будет найден подходящий по типу исключения (в нашем примере тип `MyException`) блок `try-catch`. Функции без `try-catch` убиваются автоматически.
3. Если подходящий блок не был найден, и исключение вылетело за `main()` , то программа аварийно завершается.

2.7. Типы исключений

Если в программе несколько подсистем (GUI, Network, Model), то можно у каждой подсистемы сделать свой тип исключения (`GuiException, NetworkException, ModelException`) и обрабатывать их по-разному.

```
1  main() {
2    try {
3      doGame();
4    }
5    catch(GuiException& e) {
6      showMessageBox (...);
7    }
8    catch(NetworkException& e) {
9      showMessageBox(...);
10     logger.log(...);
```

```
11    }
12
13    catch(ModelException& e ) {
14      logger.log(...);
15    }
16  }
```

Можно наследовать исключения друг от друга, чтобы где-то ловить исключения сразу нескольких типов, а где-то каждого типа отдельно.

```
1  class MyException {};
2  class GuiException:public MyException {};
3  class NetworkException:public MyException {};
4  class ModelException:public MyException {};
```

Но надо быть аккуратным и помнить, что исключение поймается первым блоком `catch`, подходящий типу.
В следующем фрагменте кода будет всегда срабатывать только первый `catch`:

```
1  try { ... }
2  catch(MyException& e) {...}
3  catch(GuiException& e) {...}
```

Чтобы всё работало правильно, надо поменять порядок:

```
1  try { ... }
2  catch(GuiException& e) {...}
3  catch(MyException& e) {...}
```

В STL все исключения – наследники `std::exception`.

2.8. Исключения любого типа

Если я не знаю, какого типа исключение может броситься внутри блока `try`, я могу написать `catch`, который поймает исключение любого типа. При этом мы не будем знать структуры пойманного объекта и не сможем из него вытянуть информацию (даже если работаем в ФСБ). Зато мы, как и раньше, можем кинуть его дальше.

```
1  try {
2    doMainWork();
3  }
4  catch (...) {
5    throw;
6  }
```

Здесь `catch(...)` это именно синтаксис, там должно быть многоточие.
throw в данном случае пишется без параметра.
На самом деле кидать можно переменные и объекты любого типа (`int, char*,` ...), но я вам этого не говорил! ;)

2.9. Подводные камни исключений

При использовании исключений увеличивается выделяемое место на стеке и увеличивается время работы. Ну тут понятно, за удобства надо платить.
Однако механизм исключений таит в себе более серьезные опасности, с которыми может столкнуться неопытный программист.

Первое, о чем надо помнить, это что при Stack unwinding происходят вызовы деструкторов локальных переменных, которые снимаются со стека. Чем это грозит, рассмотрим позже.

2.9.1. Утечка памяти

При этом если какие-то объекты внутри функции были выделены динамически, до их удаления может и не дойти. Тогда произойдет утечка ~~мозгов~~ памяти.

```
f () {
  int *buffer = new int[n];
  if ( ... ) throw MyException(...);
  delete[] buffer; // не выполнится.
}
```

2.9.2. идиома RAII

RAII – Resource Acquisition Is Initialization (”Взятие Ресурса Должно Происходить через Инициализацию”, или как-то так).
Взятие ресурса нужно инкапсулировать в класс, чтобы в случае исключения вызвался деструктор и освободил ресурс.

```
void f() {
  auto_ptr p(new Person(''Jenya'', 36, true)); // или другой умный указатель
  divide(c, e); // может бросить исключение
}
```

В коде выше, если `divide` бросит исключение, то у указателя вызовется деструктор и сделает с Женей всё что нужно.

2.9.3. Исключения в конструкторе

В следующем примере во время конструирования объекта функция divide может бросить исключение, и объект останется недостроенным.

```
class PhoneBookItem {
  PhoneBookItem(const char* audio, const char* pic) {
    af = fopen(audio, ''r'');
    pf = fopen(pic, ''r'');
    divide(c, e); // может бросить исключение
    f();
  }
  ~PhoneBookItem() {
    fclose(af);
    fclose(pf);
  }
  ...
};
```

Когда вызовется деструктор, он может натворить бед, например, попытаться освободить память, которая должна была выделиться в функции `f`, которая не выполнилась из-за исключения. Поэтому в конструкторе нужно обязательно обрабатывать все исключения.

```
class PhoneBookItem {
  PhoneBookItem(const char* audio, const char* pic) {
    try {
      af = fopen(audio, ''r'');
```

```
5         pf = fopen(pic, ''r'');
6         divide(c, e); // может бросить исключение
7         f();
8       }
9
10      catch(MyException& e) {
11        fclose(af);
12        fclose(pf);
13        throw e; // отправляем e лететь дальше
14      }
15    }
16    ...
17  };
```

2.9.4. Исключения в деструкторе

Как уже говорилось, в процессе Stack unwinding вызываются деструторы. По этой причине деструкторы никогда ни за что **не должны кидать исключения!** Дело в том, что механизм исключений поддерживает не более одного одновременно летящего исключения. Как только появляется два одновременно летящих исключения, программа аварийно завершается.
Пример – В функции `f` бросилось исключение и вызвался деструктор базы данных. Этот деструктор хочет послать на сервер сообщение от том, что база закрыта. Сервер может оказаться недоступен, и бросится второе исключение, и всё упадет.

```
1  c
2  lass PersonDatabase {
3    ~PersonDatabase() {
4      // бросает исключение, если сервер недоступен.
5      networkLogger.log(''Database is closed.'');
6      ...
7    }
8  };
9
10 f() {
11   PersonDatabase db;
12   if(...) throw MyException(''Error: disk is full.'');
13 }
```

Чтобы этого не произошло:

```
1  PersonDatabase::~PersonDatabase() {
2      try {
3        networkLogger.log(''Database is closed.'');
4      }
5      catch(...) { } // поймать всё
6    }
```

2.10. Гарантии при работе с исключениями

Гарантии – это своего рода документация для программиста, который работает с функцией-/методом. Мы гарантируем, что в случае ошибки функция обязана будет вести себя каким-то более-менее вменяемым образом.
Гарантии бывают трёх видов, разберемся с ними подробнее.

2.10.1. Basic guarantee

Базовая гарантия обещает, что при возникновении исключительной ситуации не будет утечек памяти, но при этом, если в ходе выполнения функции какие-то объекты поменяли свои значения, они, скорее всего, не вернутся в исходное состояние.
Например, здесь в цикле for может броситься исключение, у указателя на Person вызовется деструктор и утечки не произойдет. Однако какие-то элементы массива, которые успешно поменяли свои значения и вывелись в `stdout`, так и останутся с новыми значениями, и из потока вывода их будет уже не вынуть, тогда как остальные будут со старыми значениями и не выведутся.

```
1  class PersonDatabase {
2    MyVector<Person> array;
3    void process() {
4      auto_ptr <Person> p(new Person(...));
5      for (int i = 0; i < array.length(); i++) {
6        int a = divide(rand(), rand()); // может бросить исключение
7        array[i]->setAge(a);
8        std::cout << p << endl;
9      }
10   }
11 };
```

Функции, в которых применяется RAII, обеспечивают как минимум basic guarantee.

2.10.2. Strong guarantee

Гарантирует не только отсутствие утечек памяти, но и то, что в случае ошибки все переменные сохранят свои значения, которые они имели до начала выполнения функции.
Самый частый пример – это банковский перевод. Допустим, с одного счёта списалась круглая сумма, а во время начисления на второй счёт произошла ошибка. Вся сумма обязана вернуться на первый счёт.
Сильной гарантии можно добиться, например, используя идиому copy-and-swap. Когда мы хотим поменять значение объекта, мы создаём его копию, применяем к ней необходимые изменения и потом свопаем с нашим объектом (или просто присваиваем), если работа с копией прошла успешно.

```
1  class PersonDatabase {
2    MyVector <Person> array;
3    void process() {
4      auto_ptr<Person> p (new Person (...));
5      MyVector<Person> copy (array);
6      for (int i = 0; i < array.length(); i++) {
7        int a = divide(rand(), rand());
8        copy[i] -> setAge(a);
9      }
10     array = copy;
11   }
12 };
```

2.10.3. No throw

Такие функции гарантируют, что они не будут бросать исключений ни при каких обстоятельствах. Если такая функция вызовет другую, которая бросает исключения, то она должна его поймать и обработать внутри себя.

```
1 void f () {
2   try {
3     divide(a , b );
4   }
5   catch(...){ // ловим всё
6   }
7 }
```

При написании кода рекомендуется по возможности стремиться к Strong guarantee. Однако это не всегда возможно, но хотя бы базовая гарантия должна быть.
Исключения – это довольно удобный способ отлавливать ошибки по вине окружения, однако не все их любят и признают. Например в Google ими пользоваться запрещено.

3. Стандартная библиотека шаблонов

В этом разделе мы немного рассмотрим основные аспекты стандартной библиотеки шаблонов (STL) языка C++ стандарта 2001 года. Стандарт C++11 будет рассмотрен отдельно позже.

3.1. Контейнеры

Подробнее про разные контейнеры можно почитать здесь и здесь.
Контейнеры бывают:

1. Последовательные
   - `array` (C++11) – обёртка над статическим массивом.
   - `vector` – динамический саморасширяющийся последовательный массив.
   - `deque` – двусторонний массив (можно добавлять в начало и конец). Поддерживает Random Access (быстрый доступ к любому элементу).
   - `forward_list` (C++11) – односвязный список.
   - `list` – двухсвязный список.
2. Ассоциативные
   - `set` – множество, хранит упорядоченные уникальные элементы в сбалансированном дереве поиска.
   - `map` – множество пар <ключ, значение>, упорядоченных по ключу. Ключи уникальны.
   - `multiset` – упорядоченное мультимножество.
   - `multimap` – упорядоченное мультимножество пар.
3. Неупорядоченные ассоциативные
   - `unordered_set` (C++11) – хеш таблица уникальных элементов.
   - `unordered_map` (C++11) – хеш таблица пар с уникальными ключами.
   - `unordered_multiset` (C++11) – хеш таблица повторяющихся элементов.
   - `unordered_multimap` (C++11) – хеш таблица пар с повторяющимися ключами.
4. Адаптеры последовательных контейнеров
   - `stack` – стек.
   - `queue` – очередь.
   - `priority_queue` – очередь с приоритетами (бинарная куча).

Адаптеры – это обёртки над последовательными контейнерами (их можно построить на любом типе послед. контейнера, подсунув этот тип в качестве шаблонного параметра).

3.2. Итераторы

Мы хотим добиться полиморфизма работы с контейнерами. Хотим уметь перебирать/добавлять/удалять элементы контейнера любого типа, не задумываясь о том, как они хранятся и в каком порядке их перебирать. Для этого есть итераторы. Это что-то вроде указателей, только с более сложной структурой. Они поддерживают операции ++, *(разыменование) и ->.
Таким образом любая последовательность задаётся двумя итераторами – начало и конец (конец не включается в последовательность). Рассмотрим пример.

```
vector<int> arr;
set<int> st;

for (vector<int>::iterator it = arr.begin(); it != arr.end(); ++it) {...}
for (set<int>::iterator it = st.begin(); it != st.end(); ++it) {...}
```

Для итераторов принято использовать префиксный оператор ++, потому что он работает быстрее постфиксного в силу того что не сохраняет промежуточное значение.
А вот как оператор ++ может быть реализован у разных типов итераторов.
Для вектора:

```
template <class T>
class vector {
  T* array;
  class iterator {
    T *pos;
    iterator& operator++() {
      pos++;
      return *this;
    }
  }
}
```

Для списка:

```
template <class T>
class list {
  Node* head;
  class iterator {
    Node pos;
    iterator& operator++() {
      pos = pos->next;
      return *this;
    }
  }
}
```

3.3. Инвалидация итераторов

Итераторы внутри себя хранят указатель или ссылку на элемент контейнера. Иногда при работе с контейнерами может очищаться/перевыделяться память, элементы могут переезжать с места на место.
Невалидный итератор – тот, который указывает на элемент, который был перемещен/удалён.
Самый простой пример инвалидации – когда создают итератор на элемент, а потом элемент удаляют. После этого итератор указывает непонятно куда. Более интересные случае рассмотрим на примере вектора.

`push_back()` Вектор периодически расширяется и перевыделяет память, после этого все итераторы инвалидируются, так как все элементы переезжают на новые места.

`insert()` При добавлении элемента в середину вектора все элементы справа от добавленного сдвигаются на одну ячейку памяти вправо, итераторы на них становятся неправильными.

`erase()` То же, что и при `insert()`, только элементы сдвигаются влево.

Обратите внимание, что в списке никакая из этих трёх проблем не встречается, так как добавление/удаление одних элементов никак не влияет на положение других.

3.4. Исключения в STL

В STL есть свои классы для исключений.
Базовый:

```
1 std::exception {
2   const char* what(); // сообщение об ошибке
3 }
```

И унаследованные:

`logic_error`:

`invalid_argument`;

`out_of_range` (`vector::at(-1)`)

`runtime_error`:

`bad_alloc` (`new int[10000000000]`)

`ios_base::failure`.

Мы можем унаследовать свой класс исключений от стандартного. Для этого он обязательно должен иметь конструктор, принимающий строчку с сообщением об ошибке.

```
1 matrix_exception : public std::logic_error {
2   matrix_exception(char* s) : logic_error(s) {}
3 };
```

При работе с потоками ввода/вывода можно настраивать, в каких случаях они будут кидать исключения. Подробнее об этом здесь.

3.5. Несколько лирических отступлений

3.5.1. Почему обязательно закрывать файл после использования

С файлами, открытыми для записи всё более-менее понятно. Если мы открыли файл, а потом его захотела открыть другая программа, то у неё или у нас будут проблемы. Файл либо второй раз не откроется, либо откроется и перезапишется. Поэтому открытый для записи файл лучше как можно быстрее закрыть.
Почему же так важно закрывать файл для чтения? На это есть ряд причин.

1. Когда мы открываем файл, в программе выделяется память под структуру для этого файла и буфер. Для экономии памяти её лучше освободить.
2. В различных операционных системах есть ограничения на количество одновременно открытых файлов.
3. Как и в случае с файлом для записи, если мы открываем файл для чтения, доступ к нему других приложений ограничивается (в винде, например, его нельзя удалить).

3.5.2. Анонимные переменные

Пусть у нас есть класс `MyArray` и функция, которая принимает объект этого класса. Мы можем написать это так:

```
1 void f(MyVector& v) {...}
2
3 MyVector arr(10);
4 f(arr);
```

А что если мы не хотим отдельно заводить переменную, потом хранить её? Мы можем передать в функцию сразу то, что вернёт конструктор.

```
1 void f(MyVector v) {...}
2 //или
3 void f(const MyVector& v) {...}
4 //но не
5 void f(MyVector& v) {...}
6
7
8 f(MyVector(10));
```

Обратите внимание, что теперь функция не может принимать неконстантную ссылку на объект, потому что временный объект нельзя изменять (он является `rvalue`). Можно либо константную ссылку, либо копию.
Какие бывают применения в реальной жизни? Пусть мы в `vector<int> arr` сделали много раз `push_back()`, а потом много раз `pop_back()`. При этом `capacity` осталась большой. В C++11 появился метод `shrink_to_fit()`, который сжимает память до достаточного объёма. В более ранних стандартах это можно было сделать так:

```
1 vector<int>(arr).swap(arr);
```

Мы создали временный объект – копию `arr` (при этом `capacity` уменьшилась) и вызвали у него метод, который свопает все поля с полями `arr`. После этого у временного объекта вызовется деструктор, а `arr` продолжит существовать с уменьшенным объемом.

3.5.3. Операторы сравнения

Мы хотим уметь сравнивать объекты любых классов. Но вот не зачада, нам подсунули объект, у которого определен только оператор <. На самом деле нам этого достаточно чтобы выразить все остальные.
Итак, мы точно знаем, что у класса `T` определен оператор <.

```
template<class T>
bool operator > (T& a, T& b) {
  return b < a;
}
template<class T>
bool operator >= (T& a, T& b) {
  return !(a < b);
}
template<class T>
bool operator <= (T& a, T& b) {
  return b >= a;
}
template<class T>
bool operator == (T& a, T& b) {
  return (a <= b) && (b <= a);
}
template<class T>
bool operator != (T& a, T& b) {
  return !(a == b);
}
```

Можно было выражать и в другом порядке. В качестве упражнения можете выразить каждый оператор только через <.

3.6. Функторы

Допустим у класса Person оператор < сравнивает объекты по именам, а мы хотим по возрасту. Мы не можем переопределить оператор < для `Person`, потому что он уже есть. Можно создать функтор.

```
class by_age {
  bool operator()(const Person &p1, const Person &p2) {
    return p1.age < p2.age;
  }

  set<Person, by_age>;
};
```

Функтор – это класс, для которого перегружен оператор (). Объект такого класса чем-то похож на функцию, но с некоторыми фичами. Например, он может хранить внутри себя какую-то информацию, а ещё его можно передать куда-нибудь в качестве параметра, например попросить, чтобы `set` сравнивал пёрсанов нашим функтором.
Функтор, возвращающий `bool`, принято называть предикатом.
Ещё пример.

```
struct accum {
  int acc;
  accum() : acc(0) { }
  void operator()(int a) {
    acc += a;
  }
};
```

```
 8 accum f;
 9 f(13);
10 f(16);
11 cout << f.acc; // 29
```

`f` аккумулирует (суммирует) в себе все значения, которые мы ему передаём.

3.7. Алгоритмы

В STL реализовано более 100 алгоритмов. Здесь мы разберем только некоторые из них. Больше информации о библиотеке алгоритмов языка C++ ищите здесь и здесь. **TODO:**

4. Стандарт C++11

TODO: