C++ второй семестр

Купоросов Василий, Недиков Константин, Казначеев Дмитрий

31 мая 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); // OK. 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.

```
class PersonDatabase {
  MyVector<Person> array;
  void process() {
    auto_ptr <Person> p(new Person(...));
    for (int i = 0; i < array.length(); i++) {
      int a = divide(rand(), rand()); // может бросить исключение
      array[i]->setAge(a);
      std::cout << p << endl;
    }
  }
};
```

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.

```
class PersonDatabase {
  MyVector <Person> array;
  void process() {
    auto_ptr<Person> p (new Person (...));
    MyVector<Person> copy (array);
    for (int i = 0; i < array.length(); i++) {
      int a = divide(rand(), rand());
      copy[i] -> setAge(a);
    }
    array = copy;
  }
};
```

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

4.1. Немного о стандартах

Какие есть стандарты языка C++:

Major – это основные стандарты, содержащие существенные изменения.

- C++98 – стандарт языка в том виде, как мы его изучали с начала семестра.
- C++11 – добавлено `auto`, `move`, `lambda`, `thread`, `unordered_map` и многое другое.
- C++17 – находится в состоянии черновика. Должен выйти в финальной версии в конце года.

Minor – промежуточные стандарты. Содержат, в том числе, исправления ошибок основных стандартов. C++03, C++14.

`gcc` по умолчанию использует 98 стандарт. Чтобы включить поддержку 11 стандарта, необходимо компилировать с флагом `-std=c++11`.

4.2. `default & delete`

Раньше, чтобы запретить использовать какой-либо метод класса (например оператор копирования), его делали приватным. Такой трюк использовался, например, в `Singletone` (объект, существующий в единственном экземпляре) и в `scoped_ptr` (обёртка над указателем).
Кроме того, когда мы объявляли конструктор с параметрами, конструктор по умолчанию не генерировался, приходилось его тоже объявлять и определять. Выглядело это так:

```
1 class A {
2 public:
3   A(int);
4   A() {}; // т.к. есть конструктор от инта - не сгенерируется дефолтный
5           // он может пригодиться, например, для динамического массива (A *arr = new A[100])
6 private:
7   A(const A&); // теперь извне копировать нельзя
8   A& operator= (const A&); // и так тоже нельзя
9 }
```

Такой способ имеет недостаток – приватные методы можно вызывать внутри самого класса и в дружественных классах.
В C++11 появилась возможность делать всё явно:

```
1 class A {
2 public:
3   A(int);
4   A() = default; // конструктор по умолчанию
5   A(const A&) = delete; // удаленный метод (использовать нельзя)
6   A& operator= (const A&) = delete; // удаленный метод (использовать нельзя)
7 }
```

4.3. override & final

Раньше писали так:

```
class Base {
public:
  virtual void f(int);
  virtual int g() const;
  void h(int);
};
```

```
class Derived : public Base {
public:
  void f(int);
  int g();      // не перекроется (забыли const)
  void h(int); // не перекроется (метод не виртуальный)
};
```

Чтобы понять, являются ли методы `f(), h()` виртуальными, надо посмотреть базовый класс. У метода `g()` забыли `const`, и он получилась перегрузка вместо перекрытия. (Дебажить очень сложно!)
В C++11 можно сказать - перекрывай мне функции явно!

```
class Derived : public Base {
public:
  void f(int) override;  // ok
  int g() override;      // compilation error
  void h(int) override;  // compilation error
};
```

Инструкция `final` связана, в основном, с оптимизацией. Вызовы виртуальных функций связаны с накладными расходами. `final` говорит, что с этого этапа наследования функция не виртуальна. Т.е. у всех потомков она не будет виртуальной. Тогда вызов будет быстрее потому что известно, что её перекрыть нельзя.

```
struct Base {
  virtual void f();
};
struct Derived : public Base {
  void f() final; // функция виртуальная, но у потомки её перекрыть не смогут
  void g() { f(); } // f() вызовется без просмотра виртуальной таблицы
};
struct DerivedDerived : public Derived {
  void f(); // ошибка: перекрывать нельзя
};
```

Классы тоже могут быть `final`. Такие классы не могут иметь детей :(

```
struct Base1 final { };
struct Derived1 : Base1 { }; // compilation error
```

4.4. Новые фишки конструкторов

Раньше из конструктора нельзя было вызвать другой конструктор этого же класса. Теперь можно.

```
class A {
  int avg = 1; // можно задать начальное значение
  A(int a1, a2) { avg = (a1 + a2) / 2; }
  // можно вызвать один конструктор из другого
  A (int *array) : A (array[0], array[1]) { }
};
```

Появились списки инициализации для векторов.

```
std::vector<std::string> v = {''AA'', ''AB'', ''AC''};
std::vector<std::string> v({''AA'', ''AB'', ''AC''});
std::vector<std::string> v{''AA'', ''AB'', ''AC''};
```

Пример использования здесь:

```
template <class T>
struct S {
    std::vector<T> v;
    S(std::initializer_list<T> l) : v(l) {
         std::cout << ''constructed with a '' << l.size() << ''-element list\n'';
    }
    void append(std::initializer_list<T> l) {
        v.insert(v.end(), l.begin(), l.end());
    }
};
```

Обратим внимание, что итератор в нём именно `const T*`, но это нам не сильно помешает, можно итерироваться и по ним, то есть если хочется достать какие-то значения - используем следующую конструкцию:

```
template<class T>
class vector {
  T* data;
  vector(std::initializer_list<T> list) {
    data = new T[list.size()];
    T* dest = data;
    for (auto &val : list) {
      *dest++ = val;
    }
  }
}
```

Это должно работать, но стоило бы и проверить.

4.5. `lvalue` и `rvalue`

Кто хочет получить перелом мозга - могут почитать формально в стандарте. Дадим менее формальное, но не менее корректное описание.

`lvalue` – может быть и в левой, и в правой части присваивания(переменные, включая константы)

- продолжает существовать за пределами своего `scope`
- есть имя
- можно взять адрес

`rvalue` – выражение, которое может быть только в правой части присваивания

- не существует за пределами своего scope
- временное значение

Например:

```
1 int a = 42;
2 int b = 43;
3 int c = a * b; //это rvalue и всё окей
4 a * b = 42; //всё не окей, потому что это rvalue, ловите error
```

```
1 vector<Person> people;
2 people.push_back(Person("Antient evil", 9999)); //Person("Antient evil", 9999) - это
      rvalue
```

```
1 int square(int x) {return x * x;}
2 int a = square(5); // Это тоже rvalue
```

Логичный вопрос: зачем мне это всё?
Ответ: во имя оптимизации, конечно же. c++ - это быстрый язык для умных дядечек, которым не лень сделать отдельные конструкторы для разных видов значений, чтобы программа работала быстрее. Почему без разделения на подобные типы программа работает медленнее? Сейчас разберёмся.
Пусть `X` - это какой-то весомый класс, который хранит ресурсы: указатель на динамически выделенную память, файлы и т.п.

```
1 X foo() {/*реализация*/)}
2 X a;
3 a = foo();
```

Что только что произошло? Очень много чего. Функция вернёт какое-то значение, его нужно будет склонировать во временную память, потом нужно отчистить то, что уже валялось в `a`, после присвоить в `a` всё из временной памяти и освободить ресурсы во временной памяти. А если foo возвращает конспект Омельченко? Не порядок, нам нет дела до временной памяти, она уходит и приходит, поэтому мы хотим сразу всё скопировать в переменную без лишних затрат. Добавим классу ещё один конструктор, так называемый move constructor

```
1 X(X&& x) {std::swap(this->data, x.data);}
```

`&&` – это rvalue reference. Заметим вот ещё что, у нас был реализован `operator=`, который тоже занимался нудным копированием, нам хочется и его сделать быстрее, но есть грязные хаки, а именно - `swap idiom`. Оператор присваивания можно не менять, если в него передаётся не ссылка, а копия объекта. Почему? У нас передаётся в оператор копия объекта, он в нём копируется

с помощью `rvalue`, потому что мы написали такой конструктор для этой ситуации, а потом внутри данные этой копии подменяются. Т.е. по сути он работает так же быстро... ну может самую малость медленнее, зато точно не ошибётся.
Теперь давайте разберём побольше краеугольных случаев. Напишем свой своп.

```
template <class T>
void swap(T& a, T&b) {
  T tmp(a);
  a = b;
  b = tmp;
}
```

Внимание, вопрос. Если реализован `move` конструктор у класса `T`, будет ли это работать быстрее? Нет. Так как `a` и `b` - это `lvalue`, у них есть имена. Если программист c++ может выстрелить себе в ногу – нужно ни в коем случае не лишать его этой возможности, поэтому добавлен специальный каст `lvalue reference` к `rvalue`, а имя ему – `std::move`.

```
template <class T>
void swap(T& a, T&b) {
  T tmp(std::move(a));
  a = std::move(b);
  b = std::move(tmp);
}
```

Теперь всё быстренько.

Есть более подробный материал с практики, но мне лень его сюда писать в 4 часа утра, поэтому оставлю ссылку на черновик, любопытным сюда.
Пните, чтобы я это потом оформил.
TODO

4.6. `auto` & `decltype`

`auto` - удобный способ для самых ленивых людей, даже слишком удобный.

```
auto x = expression;
//выведет сам по expression.

decltype(expression) y = x;
//тип будет такой же, как у expression.
//т.е. он сделает у с типом, как у expression, но запишет туда x;
```

Немного более наглядных примеров, чтобы почувствовать разницу

```
const std::vector<int> v(1);
auto a = v[0];         // a - int
decltype(v[0]) b = 1; // b - const int&

//auto можно уточнить для констант, ссылок и указателей
std::string s = "hello";
auto \&s1 = s;          // s1 - std::string&
const auto \&c = v[0]; // c - const int&

//Также можно использовать decltype в typedef
typedef decltype(v[0]) new_type;
```

Увлекаться авто не надо, если ставите авто везде - идея плохая, если у вас есть собственный класс, другим будет непонятно, что выведется, в отличии от STL, который все более менее

знают.

4.7. nullptr

Два слова о nullptr
обычно в c++ пишут что-то вроде

```
#define NULL 0
```

А это не есть хорошо, например такой код:

```
void f(int *ptr) {...}
void f(int n) {...}

f(NULL); // не поймёт
f(nullptr); // поймёт
```

Порой некоторые компиляторы определяют NULL немного сложнее:

```
#define NULL (void *)0
```

Почему так тоже не всё хорошо? Дело в преобразованиях типов.

4.8. Вычисления в compile time

1. static_assert

```
static_assert(sizeof(unsigned int) * 8 == 32, ''16bit CPU is not supported'');
```

   По сути то же самое, что и обычный assert, но работает во время компиляции.

2. constexpr

```
constexpr unsigned fibonacci(unsigned i) {
  return (i <= 1u) ? i : (fibonacci(i-1) + fibonacci(i-2));
}
```

   Этот монстр будет считаться во время компиляции, если на вход будут подаваться константы.

4.9. begin(), end(), for(:)

Появились шаблонные функции begin() и end(), они не делают ничего умнее, кроме как подставляют .begin() и .end() соответственно, если они есть, а если там массив - работают как должны работать с массивом.
Ещё появился for по контейнерам и массивам.

```
for (auto& x : v) {
  ++x;
}
//Он просто раскроет это во что-то такое:
for (auto x = begin(v); x != end(v); x++) {
  ++(*x);
}
```

Здесь важно, что перед `x` имеет тип `auto&`, то есть ссылку. В противном случае в `x` будут копироваться значения из `v`, и `++x` будет изменять только его, а содержимое контейнера останется неизменными. Более того, версия без ссылки будет работать медленнее, так как будет тратиться время на копирование. Такая конструкция цикла `for` не будет работать для массивов, память под которые была выделена динамически, ведь неизвестно, где у их конец. Если же размер задан на этапе компиляции, всё норм.

4.10. `lambda`

Что-то полезное и интересное.
Раньше мы писали компараторы, чтобы, например, передать их в функцию сортировки. Сейчас появился более короткий способ записать функции, которым не нужно какое-то особое имя.
Лямбда функции имеют три секции:

1. `capture list`
   Это переменные в квадратных скобочках, которые используются в теле функции, но не являются её аргументами. Они ”захватываются” из той области видимости, где лямбда создается.

2. аргументы
   В круглых скобках записаны аргументы, как и у обычной функции

3. тело функции
   Тут тоже всё как и у обычной функции.

Примеры:

```
1 //Отсортировать по абсолютным значениям
2 std::sort(v.begin(), v.end(), [](int a, int b){return abs(a) < abs(b);} );
3
4 //Изменяем total, прибавив к нему все значения из вектора
5 for_each(v.begin(), v.end(), [&total](int x) { total += x; });
6
7 //Не изменяем total, передав его без ссылки.
8 for_each(v.begin(), v.end(), [&total](int x) { x -= total; });
```

Можно в `capture list` передавать особенные символы: `[=]` – взять всё по значению, `[&]` – взять всё по ссылке, `[]` – ничего не брать. Можно также комбинировать это, например: `[=, &x]`, `[&x, y]`, `[&, y]`.
Также можно явно указать возвращаемое значение `lambda` функции, как в примере ниже, а также можно дать ей имя.

```
1 auto func = [](int a, int b)->bool {return a > b;};
2 bool res = func(4, 5);
3 sort(v.begin(), v.end(), func);
```

4.11. Variadic templates

Сначала немного в целом про шаблоны в C++11.
Исправлена проблема с двойными треугольными скобками, они больше не воспринимаются, как оператор.
Появилась чудесная возможность делать `typedef` для шаблонов, используя `using`:

```
1 template<typename T>
2 using 2dvector = std::vector<std::vector<T>>;
```

TODO

4.12. Threads

TODO

4.13. Множественное наследование

TODO

4.14. Метапрограммирование

Метапрограммирование – это когда программа генерирует/модифицирует другую или сама себя.
Метапрограммирование в C++ – это когда какие-то операции (вычисления) производятся во время компиляции.
Метафункция во время компиляции:

- вычисляет значение/константу (`constexpr`) или тип
- выбирает тот или иной вариант алгоритма (`#ifdef`)
- генерирует алгоритм (`variadic templates`)

Рассмотрим следующий код:

```
1  template <int N>
2  struct Factorial {
3    static int value = N * Factorial<N-1>::value;
4  };
5
6  template <>
7  struct Factorial<0> {
8    static int value = 1;
9  };
10
11 std::cout << Factorial<10>::value << std::endl;
```

При компиляции последней строчки создастся класс `Factorial<10>`, для которого создастся класс `Factorial<9>`, и.т.д.
То есть компилятор будет рекурсивно генерировать классы, пока не дойдет до `Factorial<0>`, который является базой рекурсии.
После этого в runtime факториалы чисел от 0 до 10 будут вычисляться за $\mathcal{O}(1)$ без предподсчета!

Рассмотрим другой пример. Функция, вычисляющая, евклидово расстояние между двумя точками "втупую":

```
float euclidean_baseline(int n, float* x, float* y) {
  for (int i = 0; i < n; ++i) {
    float num = x[i] - y[i];
    result += num * num;
  }
  . . .
}
```

Компиляторы имеют разные расширения, позволяющие заставить процессор что-то делать эффективнее. Например, следующий код решает ту же задачу, но с помощью супер-крутых эффективных команд, встроенных в процессор (`intrinsic functions`):

```
float euclidean_intrinsic(int n, float* x, float* y) {
  __m128 euclidean = _mm_setzero_ps();
  for(; n > 3; n -= 4) {
    __m128 a = _mm_loadu_ps(x);
    __m128 b = _mm_loadu_ps(y);
    __m128 a_minus_b = _mm_sub_ps(a, b);
    __m128 a_minus_b_sq = _mm_mul_ps(a_minus_b, a_minus_b);
    euclidean = _mm_add_ps(euclidean, a_minus_b_sq);
    x += 4;
    y += 4;
  }
  . . .
}
```

Не будем разбираться, как это работает, главное что это работает быстро, но не на всех процессорах. Поэтому иногда при компиляции нужно выбирать "тупую" реализацию. Здесь нам на помощь снова придет метапрограммирование:

```
float euclidean(int dim, float* x ,float* y) {
  float (*euclidean)(int, float*, float*) = euclidean_baseline;
#ifdef __SSE__
  euclidean = euclidean_intrinsic;
#endif
  return euclidean(dim, x, y);
}
```

Функция `euclidean` по-умолчанию имеет внутри себя указатель на обычную реализацию, однако, если процессор поддерживает расширение команд SSE (в таком случае в стандартных заголовках есть дефайн `__SSE__`), то указатель становится на продвинутую реализацию.

4.14.1. SFINAE

SFINAE расшифровывается "Substitution Failure Is Not An Error" – неудачная подстановка – не ошибка.
Что же это означает?
Пусть есть **шаблонная** функция, имеющая несколько перегрузок. Некоторые перегрузки вызывают синтаксическую при попытке подставить туда наши шаблонные параметры. Но при этом не происходит ошибка компиляции, а перегрузка просто исключается из списка кандидатов, и компилятор ищет более подходящую версию.

Рассмотрим пример.

```
template <typename T>
void show (typename T::iterator x, typename T::iterator y) {
  for(; x != y; ++x)
    cout<< *x << " ";
  cout << endl;
}

show<int>(16, 18);
```

Здесь будет ошибка компиляции, потому что внутри `int`'а нет класса `iterator`.
Компилятор скажет:

```
error: no matching function for call to 'show(int, int)'
note: candidate  is:
note: template void show(typename T::iterator, typename T::iterator)
```

А теперь добавим ещё одну реализацию этой же функции

```
template <typename T>
void show (typename T::iterator x, typename T::iterator y) {
  for(; x != y; ++x)
    cout << *x << " ";
  cout << endl;
}

template <typename T>
void show(T a, T b) {
  cout << a << "; " << b << endl;
}

show<int>(16, 18);
vector<int> arr = {16, 17, 18};
show<int>(arr.begin(), arr.end());
```

Теперь всё успешно скомпилируется, причем при первом вызове выведется

$$16; 18$$

а при втором

$$16\ 17\ 18$$

То есть если для какой-то шаблонной подстановки подходят несколько вариантов, выбирается **первый**.
Теперь хотим написать класс, в который можно передать какой-то тип и во время компиляции понять, есть ли у него итератор. После чего на основании этой проверки можно будет выбирать тот или иной алгоритм.

```
template <typename T>
struct has_iterator {
  template <typename U>
  static char test(typename U::iterator* x);
  template <typename U>
  static long test (U* x);
  static bool value = sizeof(test<T>(0)) == 1;
};

bool first = has_iterator<int>::value;
bool second = has_iterator<std::vector<int>>::value;
```

ШТА здесь происходит? Давайте разбираться.
В качестве шаблонного параметра в класс передается какой-то тип. В первом случае (строка 10) это `int`.
Сразу же вычисляется статическая переменная `has_iterator<T>::value`. Она равна `true` если размер значения, возвращаемого функцией `has_iterator<T>::test<T>` равен 1 байту и `false` – иначе.
Функция `has_iterator<T>::test<T>` получает указатель на `T`. Мы передаем в неё ноль, который неявно приводится к нужному типу (нулевой указатель – тоже указатель).
При подстановке `int*` в первую реализацию возникает ошибка, так как нет итератора. Поэтому выбирается вторая. Тип её возвращаемого значения `'long'` занимает 4 байта. `4 != 1`, поэтому в `value` присваивается `false`.
Если же мы передадим классу тип `std::vector<int>`, у него будет итератор. Значит для подстановки подходят оба варианта функции `test<T>`, из них выберется первая. Она возвращет `char`, который имеет размер 1 байт. `value` будет равно `true`.
Таким образом в переменной `first` будет `false` (действительно, `int` не имеет итератора), а в переменной `second` будет `true`. И всё это посчиталось в `compile time`.

4.14.2. Отступление про `typename`

В нашем коде была строчка

```
static char test(typename U::iterator* x);
```

Зачем здесь ключевое слово `typename`?
Потому что когда компилятор видит выражение `U::iterator* x`, он не знает, что представляет из себя `U::iterator`. Это может быть тип, тогда выражение задает указатель на этот тип. А может быть статической переменной, тогда `'*'` в этом выражении будет оператором умножения. Чтобы явно сказать компилятору, что это тип, надо написать ключевое слово `typename`.

4.14.3. `enable_if`

`enable_if` – это метакласс в STL, который позволяет включать в исходник функцию только если выполнено условие
Рассмотрим ещё одну пару функций `show`.

```
1 template <typename T>
2 typename enable_if<!has_iterator<T>::value, void>::type show(const T& x) {
3   cout << x << endl ;
4 }
5 template <typename T>
6 typename enable_if<has_iterator<T>::value, void>::type show(const T& x) {
7   for (auto& i : x)
8     cout << i << endl;
9 }
```

Эта пара функций отличается от предыдущей тем, что у них одинаковая сигнатура, поэтому компилятор не может понять, какую реализацию надо включать. `enable_if` делает так, что первая функция компилируется только для тех типов, у которых нет итератора (для которых `has_iterator<T>::value == false`), а вторая только для тех, у которых есть. Далее в игру вступает SFINAE и выбирает то что нужно.
У `enable_if` есть два шаблонных параметра. Первый – это условие, туда можно передавать любое выражение типа `bool`. Второй параметр – это тип, который подставится вместо этой страшной строчки в случае успеха.

Как реализован `enable_if`:

```
template <bool B, class T>
struct enable_if {};

template <class T>
struct enable_if<true, T> {
 typedef T type;
};
```

Всё очень просто. У этого класса есть общая реализация, у которой нет никакого `enable_if::type`. И есть специализация `enable_if<true, T>` для случая, когда условие истинно. У этой специализации внутри есть тайпдеф, который делает `enable_if::type` синонимом `T`, то есть того типа, который мы передали вторым параметром.
Когда мы в коде пишем `enable_if<EXPR, TYPE>::type`, то в случае когда `EXPR == false`, ничего не скомпилируется, а в случае когда `EXPR == true`, скомпилируется, и на это место подставится `TYPE`.
Ещё примерчик применения `enable_if`. Все мы знаем, что процессоры и системы имеют разные битности. В зависимости от этого бывает эффективно делать операции с тем или иным количеством битов одновременно. В следующем примере в зависимости от битности системы происходит копирование памяти по 32 или 64 бита:

```
struct is_64_bit {
  static const bool value = sizeof(void*) == 8; // true если указатель занимает 64 бита
};

template <typename T = void>
typename enable_if<is_64_bit::value, T>::type
         my_memcpy(void* target, const void* source, size_t n) {
  cout << ''64 bit mem cpy'' << endl;
}

template <typename T = void>
typename enable_if<!is_64_bit::value, T>::type
         my_memcpy(void* target, const void* source, size_t n) {
  cout << ''32 bit mem cpy'' << endl;
}
```

Внимательный читатель удивиться, зачем мы искусственно вводим шаблонный параметр `T = void`, если функция ничего возвращать не может? Дело в том, что `enable_if` работает только с шаблонными функциями, поэтому приходится вставлять костыль.

4.14.4. `type_traits`

В заголовочном файле `<type_traits>` реализовано множество классов в стиле нашего `has_iterator`. Они называются трейты (`traits`). Их можно использовать в `enable_if`. Например это `is_integral, is_pointer, is_copy_assignable, ...`
Без них пришлось бы писать реализацию функции для каждого примитивного типа или один общий случай и несколько специальных реализация при помощи шаблонов.
`enable_if` и `type_traits` можно сделать реализацию для целой пачки типов, удовлетворяющих определенному условию (например отдельно для целых типов и отдельно для вещественных).

Дополнительную информацию в менее удобном виде можно посмотреть в
презентациях с лекций;
Костиных черновиках;
Васиных черновиках.