

C++ второй семестр

Купоросов Василий, Недиков Константин, Казначеев Дмитрий

25 марта 2017 г.

Содержание

1. Шаблоны	1
1.1 Шаблонные классы	1
1.1.1 C-style	1
1.1.2 C++ style – шаблоны	2
1.2 Классы с несколькими шаблонными параметрами	3
1.3 Шаблонные функции	3
1.4 Другие виды параметры шаблона	5
1.5 Неинстанцированный шаблон	6
1.6 Значение по умолчанию	6
1.7 Специализация шаблона	6
1.8 Частичная специализация	7
1.9 Экзотические случаи шаблонов	7
2. Исключения	8
2.1 Виды ошибок	8
2.2 Обработка ошибок	8
2.3 Обработка ошибок в C-style	9
2.3.1 Через возвращаемое значение	9
2.3.2 Через глобальную переменную	9
2.4 Недостатки C-style	10
2.5 C++-style, исключения	10
2.6 Stack unwinding	11
2.7 Типы исключений	11
2.8 Исключения любого типа	12
2.9 Подводные камни исключений	12
2.9.1 Утечка памяти	13
2.9.2 идиома RAII	13
2.9.3 Исключения в конструкторе	13
2.9.4 Исключения в деструкторе	14

2.10	Гарантии при работе с исключениями	14
2.10.1	Basic guarantee	15
2.10.2	Strong guarantee	15
2.10.3	No throw	15
3.	Стандартная библиотека шаблонов	17
4.	Стандарт C++11	18

1. Шаблоны

1.1. Шаблонные классы

Пусть мы написали свой класс, хранящий внутри себя объекты какого-то типа. Раньше эти типы фиксировались во время 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 дробными
4 int main() {
5     MyArray_int a;
6     MyArray_double b;
7 }
```

Если хочется посмотреть, как изменился код после препроцессора, можно запустить `g++` 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.

```
1 | #define MyArray(TYPE) . . .
2 | MyArray(int);
3 | MyArray_int arr;
4 | int TYPE;
```

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

```
1 | 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:

```
1 | template <typename T>
2 | class MyArray {
3 | private:
4 |     size_t size;
5 |     T *array;
6 | public:
7 |     T& get(size_t index) {
8 |         return array[index];
9 |     }
10 |     T& MyArray<T>::operator [] (size_t index);
11 |     MyArray<T>& MyArray<T>::operator=(const MyArray<T>& obj);
12 | };
13 |
14 | template <typename T> // <class T> эквивалентно <typename T>
15 | T& MyArray<T>::operator [] (size_t index) {
16 |     return array[index];
17 | }
```

main.cpp:

```
1 | #include "MyArray.h"
2 | int main() {
3 |     MyArray<int> a;
4 |     MyArray<double> b;
5 |     MyArray<MyArray<int> > arr; // работает всегда
6 |     MyArray<MyArray<int>> arr; // только в 11 стандарте
7 | }
```

5 строчка отличается от 6 наличием пробела между закрывающими угловыми скобками. компиляторы с++ 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 C-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 отдельной 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 `c++`, 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 напомним, что отдельная 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:

```

1 | template <typename T>
2 | class MyArray {
3 | private:
4 |     size_t size;
5 |     T *array;
6 | public:
7 |     MyArray(size_t s) {
8 |         array = new T[size];
9 |     }
10 | };
11
12 | MyArray arr(10); // Ошибка. Неизвестен тип хранимых объектов.
13 | MyArray<MyArray<int>> > arr(10); // Ошибка. Для внутреннего объекта отсутствует
14 |                               // конструктор по умолчанию.

```

Ещё пример:

```

1 | template <typename T>
2 | bool less(T& a, T& b) {
3 |     return a < b;
4 | }
5
6 | int a, b;
7 | double c;
8 | less(a, b); // ОК. Компилятор выведет тип int.
9 | less(a, c); // Ошибка. Не понятно int или double.

```

Рассмотрим пример функции с несколькими шаблонными типами.

```

1 | template <typename T, typename V>
2 | void copy(MyArray<T>& a, MyArray<V>& b) { }
3
4 | MyArray<int> a;
5 | MyArray<double> b;
6 | copy<int, int>(a, b); // int приведется к double.
7 | copy(a, b); // Типы однозначно выводятся.
8 | 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).

```

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

```

2) ООП.

```

1 | class comparable { //базовый класс для всех сравниваемых типов
2 |     virtual int compare(const comparable *o) const = 0;
3 | };
4 | 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 породить новый класс.

3) generic programming (шаблоны):

```
1 | template <typename T>
2 | void sort(T *array, size_t n);
3 |
4 | GaussNumber a[100];
5 | 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).

```
1 | template <size_t Size>
2 | class Bitset{
3 | private:
4 |     char m[(Size - 1) / 8 + 1];
5 | public:
6 |     bool get(size_t index) { }
7 | };
8 |
9 | Bitset <128> b1;
10 | 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. Неинстанцированный шаблон

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

```
1 | template <typename T, class Container>
2 | class Stack{
3 | private:
4 |     Container c;
5 | public:
6 |     void push(const T& v) {}
7 | };
```

При этом могут возникать проблемы.

```
1 | Stack <int, List<int> > s1; //OK
2 | 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.

```
1 | template <typename T, template <typename> Container>
2 | class Stack{
3 | private:
4 |     Container<T> c ;
5 | };
6 |
7 | Stack <int, List> s1 ; // List<int>
8 | Stack <double, Vector> s2 ; // Vector<double>
```

1.6. Значение по умолчанию

Можно задавать значения шаблона по умолчанию.

```
1 | template <typename T, template <typename> Container = Deque>
2 | class Stack{ ... };
3 | 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:

```
1 | template <typename T>
2 | class Array {
3 | private:
4 |     T* a;
5 |     ...
6 | public:
7 |     Array(size_t size) {
8 |         a = new T[size];
9 |     }
10 |     ...
11 | };
```


Код специально для bool:

```
1 | template <>
2 | class Array <bool> {
3 | private:
4 |     char* a ;
5 |     ...
6 | public:
7 |     Array(size_t size) {
8 |         a = new char[(size - 1) / 8 + 1];
9 |     }
10|};
```

Обратите внимание, что в специальной реализации надо заново писать все методы (даже если некоторые из 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:

```
1 | template <class T>
2 | class Array <Array<T> > {
3 |     T** a;
4 | };
```

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

```
1 | template <size_t N>
2 | class Array<bool, N> {
3 |     ...
4 |};
```

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.

```
1 | int load_config() {
2 |     f = fopen();
3 |     if (f == NULL)
4 |         return -1;
5 | }
6 |
7 | r = load_config();
8 | if (r == -1) {
9 |     //обработка ошибки
10| }
```

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.

```
1 | #include <errno.h>
2 | FILE* fopen(...) {
3 |     if (file not found) {
4 |         errno = 666;
5 |         return NULL;
6 |     }
7 |     if (permission denied) {
8 |         errno = 777;
9 |         return NULL;
10|     }
11| }
```

```

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` в `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`, подходящим 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.

```

1 | f () {
2 |     int *buffer = new int[n];
3 |     if ( ... ) throw MyException(...);
4 |     delete[] buffer; // не выполнится.
5 | }
```

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.

```

1 | void f() {
2 |     auto_ptr p(new Person("Jenya", 36, true)); // или другой умный указатель
3 |     divide(c, e); // может бросить исключение
4 | }
```

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.

```

1 | class PhoneBookItem {
2 |     PhoneBookItem(const char* audio, const char* pic) {
3 |         af = fopen(audio, "r");
4 |         pf = fopen(pic, "r");
5 |         divide(c, e); // может бросить исключение
6 |         f();
7 |     }
8 |     ~PhoneBookItem() {
9 |         fclose(af);
10 |        fclose(pf);
11 |    }
12 |    ...
13 | };
```

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

```

1 | class PhoneBookItem {
2 |     PhoneBookItem(const char* audio, const char* pic) {
3 |         try {
4 |             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 деструкторы. 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 class 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 вернуться 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

TODO:

4. Стандарт C++11

TODO: