Лекция IX

ООП и динамика

Динамические объекты в C++

- Все объекты в C++ могут быть динамическими.
- Главным свойством динамического объекта является его **время существования** в программе: объект существует до тех пор, пока не будет удалён вручную (либо программа не завершится).
- Создание и удаление динамических объектов происходит с помощью операторов **new** и **delete** соответственно (и их версий для массивов объектов).
- Действия с динамическими объектами выполняются с использованием **указателей** на нужный тип данных (класс, структура).

Динамические объекты в C++

Для повторения: пример использования **new** и **delete** для **фундаментальных** типов

```
1 double *p_real;
2 p_real = new double;
3
4 *p_real = 145.897;
5 std::cout << *p_real << '\n';
6
7 delete p_real;
```

Короткое приключение одинокого указателя-на-**double**. По сути:

строка 2: выделяем **одну** динамическую переменную типа **double**. Её адрес записываем в **p_real**;

строка 4: используя оператор разыменования (*****) в дин. переменную записываем значение **145.897**;

строка 7: удаляем динамическую переменную вручную.

Для подобных типов больше значения имеет создание динамических массивов, о них ещё вспомним позже.

Динамические объекты в C++

В предыдущем примере технически оператор **new** во **второй строке** отработал следующим образом:

1. посмотрел на запрашиваемый тип (в данном примере - **double**);
2. узнал, сколько нужно памяти под **одну** переменную этого типа (язык C++ всегда в курсе);
3. запросил требуемое количество памяти у ОС;
4. если выделение прошло успешно, вернул адрес выделенного блока (который сохранился в указателе **p_real**).

В дополнении к общей схеме при динамическом создании одной переменной **фундаментальных** типов, её можно присвоить конкретное значение. Делается это так:

```
2 p_real = new double{};
3 // создаётся динамическая перменная со значением 0.0
```

или так

```
2 p_real = new double{-1.4};
3 // создаётся динамическая перменная со значением -1.4
```

Динамические объекты в C++

Общая схема работы **new** для **динамических объектов** (переменных пользовательских типов данных) немного изменится:

1. оператор смотрит на запрашиваемый тип;
2. узнаёт, сколько нужно памяти под **одну** переменную этого типа (для пользовательских типов - сумма блоков памяти под каждое поле);
3. запрашивает требуемое количество памяти у ОС;
4. если выделение прошло успешно, **вызывает конструктор**;
5. возращает адрес динамического объекта.

Обратите внимание

Чётвёртый пункт является **ключевым** для создания динамических объектов: оператор **new** всегда попытается вызвать **конструктор** после выделения памяти под конкретный объект.

Динамические объекты в C++

Рассмотрим на примере из предыдущей лекции. Там был разработан класс:

```
 1 class SafetyArray
 2 {
 3 public:
 4     SafetyArray();
 5     SafetyArray(size_t sz);
 6     SafetyArray(size_t sz, double value);
 7     ~SafetyArray();
 8
 9     double elem_at(int index) const;
10     size_t length() const;
11     SafetyArray& push_to_end(double new_value);
12     SafetyArray& set_at(int index, double value);
13     SafetyArray& operator=(const SafetyArray& rhs);
14     friend std::ostream& operator<<(std::ostream&, const ↩
           SafetyArray&);
15 private:
16     double *_arr;
17     size_t _length;
18     size_t _capacity;
19
20     size_t compute_index(int index) const;
21 };
```

Динамические объекты в C++

И пробуем создать динамический объект класса **SafetyArray**

```
1 SafetyArray *p_arr = new SafetyArray;
```

Здесь:

- переменная **p_arr** имеет тип **указатель на SafetyArray**;
- оператор **new** после выделения памяти под объект типа **SafetyArray** пытается вызвать конструктор. Поскольку не указаны никакие параметры, он ищет либо **конструктор без параметров**, либо обращается к **конструктору по умолчанию**;
- так как класс **SafetyArray** содержит в себе **конструктор без параметров** (*седьмой слайд, строка 4*) - он и вызывается.

Итого: указатель **p_arr** ссылается на объект класса **SafetyArray**, который был создан с помощью **конструктора без параметров**.

Динамические объекты в C++

Нет никаких препятствий для вызова других конструкторов класса **SafetyArray**. Синтаксис следующий:

```
1 SafetyArray *p_arr1 = new SafetyArray{16};
2
3 SafetyArray *p_arr2 = new SafetyArray{8, 555.555};
4
5 p_arr1->push_to_end(-1.4);
6 cout << p_arr1->elem_at(1);
7 // ...
```

строка 1: создаём динамический объект с помощью конструктора с одним параметром (седьмой слайд, строка 5).
строка 3: создаём динамический объект с помощью конструктора с двумя параметрами (седьмой слайд, строка 6).
строки 5-6: демонстрация работы с объектом через указатель.
Стоит помнить: синтаксис C++ позволяет вместо *фигурных* скобок, использовать *круглые*. Но первый вариант в настоящее время *предпочтительней*.

Динамические объекты в C++

Когда динамический объект становится ненужным, от него следует избавиться. С технической точки зрения - высвободить используемую под него память. Для этого применяется оператор **delete**. Общая схема его работы для динамических объектов:

1. оператор получает объект для удаления;
2. определяет тип объекта;
3. вызывает **деструктор** объекта;
4. высвобождает память, используемую под объект.

В виде примера:

```
1 SafetyArray *p_arr1 = new SafetyArray{16};
2 SafetyArray *p_arr2 = new SafetyArray{8, 555.555};
3 // ... что-то делаем
4
5 delete p_arr1; // удалили первый объект
6 delete p_arr2; // а теперь и второй
```

Динамические объекты в C++

Аналогично **фундаментальным** типам, можно создавать массивы динамических объектов с помощью операторов **new[]** и **delete[]**. Для примера:

```
1 SafetyArray *p_arrays = new SafetyArray[5];
2
3 p_arrays[2].push_to_end(9.999);
4
5 delete[] p_arrays;
```

строка 1: создаём пятиэлементный динамический массив объектов **SafetyArray**. Для каждого объекта в данном случае была выделена память под него и был вызван **конструктор без параметров**.

строка 3: после работы с динамическим массивом, удалаляем все его элементы. Для каждого из пяти объектов будет вызван деструктор и выделенная под объект память возращена ОС. **Обратите внимание**, для массивов должен быть вызван оператор **delete[]**, а не **delete**

Динамические объекты в C++

А что, если для каждого объекта массива хочется вызывать конструктор с параметрами. «Нет каких преград» - отвечает Вам C++:

```
1 SafetyArray *p_arrays = new SafetyArray[3] {3, 4, 5};
2
3 p_arrays[1].push_to_end(9.999);
4 cout << p_arrays[1].elem_at(0) << "\n";
5
6 delete[] p_arrays;
```

В **первой строке** создаём массив на три элемента, для каждого из которых вызывается **конструктор с одним параметром** (слайд 7, строка 5). Причём для первого объекта в конструктор передаётся число **три**, во второй - **четыре**, в третий - **пять**.

Динамические объекты в C++

Более того, для **SafetyArray** есть ещё конструктор с двумя параметрами. И его можно вызвать при создании динамического массива объектов:

```
1  SafetyArray *p_arrays =
2              new SafetyArray[3] { {3, 1.5},
3                                   {4, -0.4},
4                                   {5, 900.03} };
5
6  cout << p_arrays[0].elem_at(0) << "\n";
7  cout << p_arrays[1].elem_at(1) << "\n";
8  cout << p_arrays[2].elem_at(2) << "\n";
9
10 delete[] p_arrays;
```

Первый объект массива будет создан с вызовом конструктора **SafetyArray(3, 1.5)**, второй - **SafetyArray(4, -0.4)**, третий - **SafetyArray(5, 900.03)**.
Итого: параметры в конструктор каждого объекта динамического массива можно передать с использованием пары *фигурных* скобок.

Пространства имён (**namespaces**)

Пространства имён в C++

В C++ любое **объявление** и **определение** переменной, функции, псевдонимов, пользовательских типов данных (структуры, классы, перечисления, объединения) может быть помещено в **пространство имён**.

Технически, **пространство имён** - это **лексическая** область видимости для группы *идентификаторов*.

По смыслу, **пространство имён** - это именованное множество, название которого необходимо для точного указания некоторой переменной, функции или типа данных.

Пространства имён в C++ - **открыты для расширения**: в любое из них (хоть из стандратной библиотеки, хоть из собственной, хоть из внешней) каждая программа может добавить свой набор констант, функций, типов, объектов и прочего.

Пространства имён в C++

Пространство имён создаётся с помощью ключевого слова **namespace** и выбора названия. Например,

```
1  namespace fp603
2  {
3  const size_t players = 7;
4
5  enum class StudyState { READY, SOMETIMES_LATER, NEVER };
6
7  bool ready_to_pass(StudyState status)
8  {
9    return status == StudyState::READY;
10 }
11
12 struct Participant
13 {
14   std::string name;
15   StudyState status;
16 };
17 }
```

Имя пространства имён - **fp603**, внутри него содержатся - константа, перечисление, функция, структура - каждой сущности по одной штуке.

Пространства имён в C++

Само пространство имён ограничено блоком из фигурных скобок (строки 2 и 17 с предыдущего слайда).
Ко всему содержимому пространства имён можно обратиться с помощью его имени и **двойного двоеточия**. В виде кода:

```
18
19 cout << "Константа равна: " << fp603::players << '\n';
20
21 fp603::Participant p1 = {"Nikita", fp603::StudyState::READY};
22
23 if ( fp603::ready_to_pass(p1.status) ) {
24   cout << "Студент " << p1.name
25        << ", наиболее вероятно, проги сдаст!\n";
26 }
```

Всё как обычно, разве что для всех сущностей появилась постоянная приставка «**fp603::**». Как можно заметить из кода выше, при частом использовании идентификаторов из пространства имён, каждый раз добавлять его имя - утомительно.

Пространства имён в C++

Для упрощения доступа к идентификаторам используется оператор **using**. И большинство смотрящих слайды его видело и не раз:

```
1 using namespace std;
```

- данное использование **using** делает **все идентификаторы** из пространства имён **std** доступными в текущей **лексической области видимости** (другими словами, происходит *импорт названий*).

Лексическая область видимости определяется просто - это или блок внутри пары *фигурных* скобок (и любые вложенные в него подблоки), или весь файл с исходным кодом, начиная со следующей строки. Так, в примере выше, **using namespace** используется вне любых скобок, то всё из пространства имён **std** доступно по прямым именам, начиная со второй строки.

Для собственного пространтсва имён **fp603** правила аналогичны. Для примера ограничения лексической области видимости, рассмотрим код

```
1  int main()
2  {
3    // здесь нужно обращаться с добавлением приставки
4    fp603::Participant man;
5
6    { // создаёт дополнительную область видимости
7      std::cout << "Введите имя: ";
8      std::cin >> man.name;
9
10     using namespace fp603;
11     // далее всё из пространства имён доступно без приставки
12
13     if (man.find("Ю") == 0 || man.find("Н") == 0) {
14       man.status = StudyState::READY;
15     } else {
16       man.status = StudyState::SOMETIMES_LATER;
17     }
18   }
19   // тут снова имена из *fp603* доступны только с приставкой
20   std::cout << "Студент " << p1.name
21   if ( fp603::ready_to_pass(p1.status) ) {
22     std::cout << ", наиболее вероятно, проги сдаст!\n";
23   } else {
24     std::cout << " в течении двух лет проги сдаст!\n"
25   }
26 }
```

Пространства имён в C++

С помощью **using** можно делать доступным только часть идентификаторов из пространства имён

```
1  using std::cout;
2  using std::endl;
3
4  using fp603::ready_to_pass;
5  using fp603::Participant;
6
7  Participant p1 = {
8    "Maxim", fp603::StudyState::SOMETIMES_LATER};
9
10 if ( !ready_to_pass(p1.status) ) {
11   cout << "не смог, так не смог" << endl;
12 }
```

В примере после действия операторов **using** название функии (**ready_to_pass**) и структуры (**Participant**) можно использовать напрямую, а вот имя перечисления (**StudyState**) - только с явным указанием пространства имён. **Минус данного подхода** только один - нужно явно каждый идентификатор добавить с оператором **using**. Кроме того, отдельные идентификаторы также могут быть добавлены в ограниченную область видимости.

Пространства имён в C++

Правило хорошего тона

В современном C++ рекомендуется по возможности использовать полный импорт идентификаторов (**using namespace fp603** и ему подобные) только внутри отдельных блоков кода, ограниченных парой *фигурных скобок* (функции, классы, другие пространства имён).

Зачем вообще нужны?

Пространства имён позволяют использовать разные библиотеки, содержащие одинаковые по именованию сущности. Например, если есть класс **Vector** в библиотеках **lib1** и **lib2**, то в своей программе можно использовать обе реализации, если внутри библиотек используются разные пространства имён. И у компилятора не будет никаких претензий.

К слову, про претензии компилятора: следующий код не скомпилируется:

```
1  namespace sp1
2  {
3  struct MyRecord {};
4  }
5
6  namespace sp2
7  {
8  struct MyRecord {};
9  }
10
11 int main()
12 {
13   using namespace sp1;
14   using namespace sp2;
15   // не компилируется
16   MyRecord mr;
17 }
```

Для исправления ситуации отлично подходят **псевдонимы**

```
1  namespace sp1
2  {
3  struct MyRecord {};
4  }
5
6  namespace sp2
7  {
8  struct MyRecord {};
9  }
10
11
12 int main()
13 {
14   using MyRecordV1 = sp1::MyRecord;
15   using MyRecordV2 = sp2::MyRecord;
16   // Всё работает
17   MyRecordV2 mr;
18 }
```

Стандартная библиотека C++

В настоящий момент все типы данных, константы, перечисления и объекты стандартной библиотеки помещаются в пространство имён **std**.

Расширение пространтсва имён

В завершении слайдов о **namespaces**, пример на расширение:

```
1 namespace fp603
2 {
3   const size_t real_players = 5;
4 }
```

такой код работает без проблем, с учётом того, что пространство имён **fp603** уже определено.

Проектирование и обработка ошибок в программах

Какие ошибки бывают?

Классификация ошибок проектирования исполняемых блоков кода (а-ля *функции*)

↙

Логические ошибки - неправильная реализация выбранных/придуманных алгоритмов. Выявление подобных проблем возможно только через *тестирование кода*. Не рассматривается в данной лекции.

↘

Технические ошибки - проблемы возникающие при работе с входными параметрами и возвращаемыми значениями. Требуют **продумывания** при написании функций и **внимания** при их использовании.

Технические ошибки I

Основными подходами к проектированию и обработке ошибок данного типа являются:

1. **Ошибки не нужны**: написание функций, которые не содержат ошибочных ситуаций: для случая любых входных параметров можно вернуть значение со смыслом.
2. **Ошибка - вон из программы**: вызов внутри блока кода комманд, немедлено завершающих выполнение программы.
3. **Ошибке - своё значение**: для возращаемого значения функции задаются **специальное** значение (или несколько), которые свидетельствуют о какой-то внештатной ситуации. Подобные "особые"значения должны обязательно сопровождаться комментариями, раскрывающими суть ошибочной ситуации.

Технические ошибки II

4. **Ошибке - собственное состояние**: из функции возращается произвольное значение нужного типа, которое не имеет смысла при нормальном ходе программы. Одновременно устанавливается некоторое **глобальное** состояние, служащее индикатором проблем в программе.
5. **Ошибка - исключительная ситуация**: используется механизм исключений, предоставляемый языком программирования. Присутствует **только в C++**.

Пример: символ Кронекера $\delta_{ij} = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases}$

```
1 int kroneckers_delta(int i, int j)
2 {
3   return (i == j) ? 1 : 0;
4 }
5
6 ...
7 cout << kroneckers_delta(2, 3) << "\n";
8 cout << kroneckers_delta(5, 5) << "\n";
9 cout << kroneckers_delta(1542, 3) << "\n";
```

Никаких побочных эффектов для **любых** аргументов функции.

2. Ошибка - вон из программы

Вызов функций **exit(целое_значение)** или **abort()**, определённых в заголовочном файле <**cstdlib**> (функции доступны в C++ из стандартной библиотеки языка C)
Пример: проверка файла на успешное открытие

```
1 #include <cstdlib>
2
3 ...
4
5 ifstream data_file{"some_data_file.dat"};
6 if ( !data_file.open() ) {
7   cerr << "Не удалось открыть файл\n";
8   exit(1);
9 }
```

Данный подход **не стоит применять** при написании многократно используемых функций. Для собственных программ - вполне себе рабочий подход.

2. Ошибка - вон из программы

Разница между **exit** и **abort** состоит в следующем:
функция **exit** вызовет деструкторы всех **глобальных** объектов программы и закроет все файловые потоки из стандартной библиотеки языка C (**но не C++!**). В тоже время, для локальных переменных функции, в которой была вызвана **exit**, деструкторы не вызываются.
функция **abort** вообще не вызывает никаких деструкторов (ни глобальных, ни локальных объектов), а её действия с файловыми потоками языка C зависит от реализации (не определено стандартом).
Примеры можно посмотреть тут:
http://en.cppreference.com/w/cpp/utility/program/exit
и тут:
http://en.cppreference.com/w/cpp/utility/program/abort

3. Ошибке - своё значение

Пример 1: вернуть заглавную букву английского алфавита (в предположении, что таблица кодов ASCII соблюдается)

```
1 char get_uppercase(char letter)
2 {
3   if ( (letter >= 'a') && (letter <= 'z') ) {
4     return letter - 32;
5   } else {
6     return ???;
7   }
8 }
```

Что возращать вместо «**???**»?

3. Ошибке - своё значение

Пример 1: вернуть заглавную букву английского алфавита.
Добавляем конкретный код в случае "неправильного"символа

```
1  char get_uppercase(char letter)
2  {
3    if ( (letter >= 'a') && (letter <= 'z') ) {
4      return letter - 32;
5    } else {
6      return 0;
7    }
8  }
9
10 //...
11 char character = 'f';
12 char capital_letter = get_uppercase( character );
13
14 if (capital_letter != 0) {
15   // что-нибудь полезное
16 }
```

3. Ошибке - своё значение

Пример 1: вместо «магического» нуля добавляем константу

```
1  const char UNCORRECT_LETTER = 0;
2
3  /* Возращается UNCORRECT_LETTER если передана не буква */
4  char get_uppercase(char letter)
5  {
6    if ( (letter >= 'a') && (letter <= 'z') ) {
7      return letter - 32;
8    } else {
9      return UNCORRECT_LETTER;
10   }
11 }
12
13 ...
14
15 char character = 'f';
16 char capital_letter = get_uppercase( character );
17
18 if (capital_letter != UNCORRECT_LETTER) {
19   // что-нибудь полезное
20 }
```

Где подкралась проблема: а кто гарантирует, что возращаемые значения будут проверяться?

3. Ошибке - своё значение

Пример 2: **printf** - стандартная функция печати в консоль в языке C (аналог **cout**).

```
1 #include <cstdio>
2
3 int status = printf("Просто слова\n");
4
5 if ( status < 0 ) {
6   // Что-то случилось с выводом
7   // печать строки не удалась
8 }
```

В практически любом учебнике по языкам C/C++ ни разу не проверяется возвращаемое значение от функции **printf**. **Возможность** «закрыть глаза» на проверку возвращаемого значения - существенный недостаток данного подхода.

4. Ошибке - собственное состояние

- В стандартной библиотеке языка C существует специальная мета-переменная **errno**, которая является глобальной по отношению к любой программе и хранит в себе код произошедшей ошибки
- В C++ входит стандартная библиотека C, поэтому при её использовании **errno** также существует в программе
- Сама она определена в заголовочном файле **<cerrno>**
- По умолчанию **errno** равна 0 (ошибка функционирования программы отсутствует)
- Получить текстовое описание ошибки можно с помощью функции **strerror(код_ошибки)**, определённой в **<cstring>**
- Таблицу с возможными значениями **errno** можно посмотреть тут: http://en.cppreference.com/w/cpp/error/errno_macros

Пример 1: функция **sqrt** из математической библиотеки

```
1  #include <cerrno>
2  #include <cstring>
3  #include <cmath>
4
5  double root = sqrt(-1.0);
6  if ( errno != 0 ) {
7    cout << root << "\n"; // Напечатает: -nan
8    cout << strerror(errno) << "\n";
9
10   root = 0;
11   errno = 0;  // Сбрасываем ошибку
12 }
13
14 // Напечатает: success
15 cout << strerror(errno) << "\n";
```

Пример 2: проверка открытия файла через **ifstream**. При неудаче, также устанавливается значение errno, отличное от нуля.

```
1  #include <cerrno>
2  #include <cstring>
3  #include <fstream>
4
5  ifstream in_file("some_unexisted.dat");
6  if ( !in_file.is_open() ) {
7    cout << "Файл не был открыт. Возможная причина:\n";
8    // Напечатаем: No such file or directory
9    cout << strerror(errno) << "\n";
10 }
```

4. Ошибке - собственное состояние

В C++ для некоторых классов используется аналогичная глобальному состоянию идея - объект некоторого класса тоже может быть в ошибочном состоянии. Например, ввод некоректного значения в консоли.

```
1  double rate;
2
3  cout << "Введите число: ";
4  cin >> rate; // Введём: avr
5
6  if ( cin.fail() ) {
7    rate = 0.0;
8    cin.clear();  // Убираем ошибочное состояние
9    cin.ignore(1024, '\n'); // Отчищаем консоль от ↩
         неправильных символов
10 }
11
12 cout << "Введите снова: ";
13 cin >> rate;
```

5. Ошибка - исключительная ситуация

Исключения и их обратботка - специальный механизм языка C++, позволяющий **вызывать** ошибку в произвольном месте программы и **обработать** её вне вызвашего блока кода.
Ключевые моменты:

- Исключения сами по себе представляют **значения (объекты)** любого типа данных, доступного программе (фундаментальные типы данных (**int, double, char** и прочие), пользовательские структуры и классы, перечисления)
- Если исключение не обработано - программа прекращает работу (где-то внутри механизма обработки исключений вызывается **abort**)
- Как правило, исключения нужны в случаях, когда некоторая функция получила такой набор входных данных, при котором она не может продолжить своё выполнение

Вызов(он же - выброс, возбуждение, бросок) исключения осуществляется с помощью ключевого слова **throw**

```
1  struct CustomError
2  {
3    int code;
4    std::string message;
5  };
6
7  // Примеры использования throw
8  throw 5;
9  throw '*';
10 throw "Строка - значение исключения";
11 throw CustomError{};
12 throw CustomError{25, "Объяснение"};
```

5. Ошибка - исключительная ситуация

Перехват исключения осуществляется с помощью комбинации блоков кода **try / catch**

```
1  try {
2    /* Код, способный выбросить исключение */
3  }
4  catch (const exception_type1& ex1) {
5    /*место обработки исключений типа exception_type1
6      само значение исключения – в переменной ex1*/
7  }
8  catch (const exception_type2& ) {
9    /*место обработки исключений типа exception_type2
10     Значение исключения не получаем*/
11 }
12 catch (const exception_type3& ex3) {
13   /*место обработки исключений типа exception_type3
14     само значение исключения – в переменной ex3*/
15 }
16 catch ( ... ) {
17   /*место обработки исключений ЛЮБОГО другого типа*/
18 }
```

5. Ошибка - исключительная ситуация

Базовый пример перехвата:

```
1 try {
2   throw CustomError{5, "так надо"}
3 }
4 catch (const CustomError& err) {
5   std::cout << "Исключение перехвачено с кодом "
6             << err.code << " и сообщением: "
7             << err.message << "\n";
8 }
```

5. Ошибка - исключительная ситуация

Пример: функция чтения действительных чисел из файла (числа располагаются через пробел в текстовом файле)

```
1  SafetyArray get_numbers_from_file(string file_name)
2  {
3    ifstream in_file{file_name};
4    SafetyArray vec;
5
6    if ( !in_file ) {
7      // Что тут делать вскоре определим
8    } else {
9      double tmp;
10     while (in_file) {
11       in_file >> tmp;
12       vec.push_to_end(tmp);
13     }
14   }
15   return vec;
16 }
```

5. Ошибка - исключительная ситуация

Пример: функция чтения чисел из файла. Если файл не может быть открыт - бросаем исключение.

```
1  const int NO_FILE_ERROR_CODE = -1;
2
3  SafetyArray get_numbers_from_file(string ↩
       file_name)
4  {
5    ifstream in_file{file_name};
6    SafetyArray vec;
7
8    if ( !in_file ) {
9      throw NO_FILE_ERROR_CODE;
10   } else {
11     // чтение данных из файла
12   }
13   return vec;
14 }
```

5. Ошибка - исключительная ситуация

Пример: функция чтения чисел из файла. Если файл не может быть открыт - бросаем исключение.

```
1  SafetyArray get_numbers_from_file(string file_name);
2
3  string f_name;
4  SafetyArray my_arr;
5
6  for (size_t attempts = 0; attempts < 3; ++attempts) {
7    try {
8      cout << "\nВведите имя файла: ";
9      cin >> f_name;
10     my_arr = get_numbers_from_file(f_name);
11   }
12   catch (const int& ex_code) {
13     cout << "Ошибка: " << ex_code <<
14             " Попробуйте ещё раз...\n";
15
16     if (attempts == 2) {
17       cout << "Попытки закончились, до свидания...\n"
18     }
19   }
20 }
```

5. Ошибка - исключительная ситуация

Пример: функция чтения чисел из файла, но не меньше заданного количества. Вместо исключений целого типа - пробуем определить собственные типы данных.
В стандартной библиотеке C++ все типы исключений построены по следующему шаблону:

```
1  class SomeError
2  {
3  public:
4      SomeError(const char* message);
5      SomeError(const std::string& message);
6
7      const char* what() const;
8  private:
9      std::string _msg;
10 };
```

а именно: объекты класса создаются с сообщением об ошибке, которую можно затем получить через метод **what** объекта, брошенного в качестве исключения.

5. Ошибка - исключительная ситуация

Пример: функция чтения чисел из файла, но не меньше заданного количества.

```
1  // про реализацию — задавайте вопросы, напишем.
2  // Здесь не приводится
3  class NoFileError;
4  class NotEnoughElemsError;
5
6  SafetyArray get_enough_numbers(string file_name, size_t ↩
       at_least = 1)
7  {
8    ifstream in_file{file_name};
9    SafetyArray vec;
10
11   if ( !in_file ) {
12     throw NoFileError{"Файл не найден"};
13   } else {
14     // чтение данных из файла
15     if (vec.length() < at_least) {
16         throw NotEnoughElemsError{"Недостаточно элементов!"}
17     }
18   }
19   return vec;
20 }
```

5. Ошибка - исключительная ситуация

Пример: функция чтения чисел из файла, но не меньше заданного количества.

```
1  SafetyArray get_enough_numbers(string file_name, size_t ↩
       at_least = 1);
2
3  std::string f_name;
4  SafetyArray my_arr;
5
6  while (true) {
7    try {
8      std::cout << "\nИмя файла: ";
9      std::cin >> f_name;
10     my_arr = get_enough_numbers(f_name, 10);
11   }
12   catch (const NoFileError& err1) {
13     std::cout << "Проблема с файлом: " << err1.what() << "\n"
14          << "\nВведите другой...\n";
15   }
16   catch (const NotEnoughElemsError& err2) {
17     std::cout << "Файл некоректен: " << err2.what()
18          << "\nВведите другой...\n";
19   }
20 }
```

5. Ошибка - исключительная ситуация

Стандартная библиотека языка C++ содержит некоторое количество классов для типовых ошибок. Они определены в библиотеке <**stdexcept**>: **logic_error, domain_error, invalid_argument, length_error, out_of_range, runtime_error, range_error, overflow_error**. Справка о них: http://www.cplusplus.com/reference/stdexcept/
Базовая работа с ними одинакова:

```
1 try {
2   throw invalid_argument{"передано что-то не то"};
3 }
4 catch (const invalid_argument& ia_err) {
5   // Каждый класс из <stdexcept> опеределяет
6   // метод what() — возвращающий строку с описанием,
7   // которое может быть установлено при выбросе исключения
8   std::cout << "Неправильные аргументы: " << ia_err.what();
9 }
```

Данные готовые классы исключений можно использовать в *логически* подходящих ситуациях.