Обработка исключений

- Исключение - это ошибка которая возникает во время выполнения программы.

- Давайте разделим о**бнаружение ошибок** от **обработки ошибок**.

Обнаружение ошибки

```
// Открываем файл и возвращаем содержимое
string readFileToString(const string &filename) {

  // Попытка открыть файл
  ifstream fin(filename);
  if (!fin.is_open()) {
    // ОШИБКА! невозможно открыть
    // файл.
    // Что с этим делать?
  }
  ...
}
```

Игнорировать и продолжить работу?

Вывести сообщение?

Обработка исключений

- Какими есть способы сообщить об ошибке в программе во внешний мир:
  - Глобальные коды ошибок
  - Возврат кода ошибки
  - Throw/Catch Exceptions (механизм исключений)

Глобальные коды ошибки

- Механизм:
  1. Хранить код ошибки глобально, затем вернуть.
  2. Вызывающая функция/программа получают код ошибки.

Такой подход плохо работает для сложных программ, должно гарантироваться что код ошибки проверяется и программа продолжает работать корректно.

Ошибки состояния объекта

- Если метод класса не отработал корректно, объект переходит в состояние ошибки
- Необходимо проверять состояние объекта после каждой операции с ним.

```
...
// Попытка открыть файл
ifstream fin(filename);
if (!fin.is_open()) {
  ...
}
...
```

Возврат кода ошибки

- Механизм:
  1. Вернуть код ошибки.
  2. Вызывающая функция/программа должна проверить код возврата.

- Лучше чем глобальный код ошибки, так как происходит локально и не может быть изменена извне.
- Однако, необходимо следить за кодом возврата…

Возврат кода ошибки

```
// Возвращает n! для положительных
// и -1 в противном случае.
int factorial(int n) {

  // Проверка
  if (n < 0) {
    return -1;
  }
  ...
}
```

Возврат кода ошибки

```
// Парсит число из строки.
// Возвращает int. Возвращает ??? в
// случае ошибки.
int parseInt(const string &str) {
  // Проверка
  if (/*Недопустимый символ*/) {
    return ???;
  }
  ...
}
```

Что вернуть в случае ошибки?

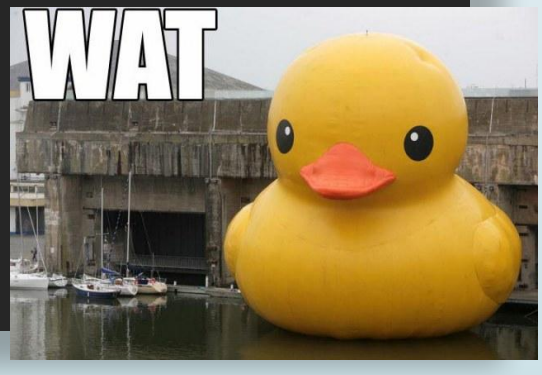
Возврат кода ошибки

```
// Создает Duck. Если что-то пошло не
// так, возвращает duck WAT

Duck makeDuck(/*Duck Parameters*/) {

  if (/*ОШИБКА*/) {
    return Duck("WAT");
  }
  ...
}
```

Как-то странно.



Возврат кода ошибки

- Вызывающая сторона может забыть проверить код возврата.

```
// Возвращает n! для положительных
// и -1 в противном случае.
int factorial(int n);

int main(int n) {

  int x = askUser();
  int f = factorial(x);

  // Использование кода ошибки в вычислениях.
  // Что может случиться??
}
```

Возврат кода ошибки

- Проверка кода ошибки чередуется с проверкой нормально значения.

```
int main() {
  int x = askUser();
  int f = factorial(x);
  if (f < 0) {
    cout << "ERROR" << endl;
  }
  else if (f < 100) {
    cout << "Small factorial" << endl;
  }
  else {
    cout << "Larger factorial" << endl;
  }
}
```

Обработка исключений

- Системные средства, с помощью которых программа может справится с ошибками времени выполнения

Использование исключений

```
int main() {
  int x = askUser();
  try {
    int f = factorial(x);
    if (f < 100) {
      cout << "Small" << endl;
    }
    else {
      cout << "Larger" << endl;
    }
  }
  catch (FactorialError &e) {
    cout << "ERROR" << endl;
  }
}
```

```
class FactorialError { };

// Возвращает n! для
// положительных
// Генерирует исключение для
// отрицательных чисел
int factorial(int n) {

  // Check for error
  if (n < 0) {
    throw FactorialError();
  }
  ...
}
```

Код которым может обнаружить ошибку помещается в try блок.

Код который, выполняется в случае если произошла ошибка.

Если что-то пошло не так генерируется исключения

Использование исключений

- Обработка исключений основывается на трех понятиях:

1. Обнаружение ошибки. Код который необходимо контролировать на предмет ошибки помещается в блок **try**
2. Выброс исключения **throw**. Сообщение о возникшей ошибке определенного типа
3. Обработка ошибки. Перехват кода ошибки и обработка ошибки соответствующим образом. Блок **catch**

Выброс исключения throw

```
class FactorialError { };

// Возвращает n! для
// положительных
// Генерирует исключение для
// отрицательных чисел
int factorial(int n) {


  if (n < 0) {
    throw FactorialError();
  }
  ...
}
```

- Когда встречается **throw** дальнейшее выполнение кода в блоке прекращается.
- Программа пытается найти соответствующий блок **catch**.
- Тип кода ошибки может быть любым, включая пользовательские типы.
  - например `FactorialError`
- Только один объект может выбросить исключение в один момент времени

try-catch блок

```
int main() {
  int x = askUser();
  try {
    int f = factorial(x);
    if (f < 100) {
      cout << "Small" << endl;
    }
    else {
      cout << "Larger" << endl;
    }
  }
  catch (FactorialError &e) {
    cout << "ERROR" << endl;
  }
}
```

- Каждому **try** блоку должен соответствовать один или несколько блоков **catch**.
- Если исключение выбрасывается внутри блока **try**, проверяются соответствующие блоки **catch**
- Если блок **catch** соответствует типу исключения, то выполняется код внутри этого блока
- Если соответствий не находится, исключение выбрасывается выше
- Необработанное исключение == крах программы.

Exercise: average

- Напишите функцию, которая усредняет числа в последовательности, заданной двумя итераторами. Если последовательность пуста, выбросите исключение.

```
class AverageException { };

// РЕЗУЛЬТАТ: Возвращает усредненное значение в
// последовательности [begin, end). Если
// последовательность  пустая выбросить исключение типа
// AverageException

template <typename Iter_type>
double average(Iter_type begin, Iter_type end) {

  // РЕАЛИЗАЦИЯ

}
```

Solution: average

```
class AverageException { };

// РЕЗУЛЬТАТ: Возвращает усредненное значение в
// последовательности [begin, end). Если
// последовательность  пустая выбросить исключение типа
// AverageException
template <typename Iter_type>
double average(Iter_type begin, Iter_type end) {
  int count = 0;
  double total = 0;
  if (begin == end) {
    throw AverageException();
  }
  while (begin != end) {
    ++count;
    total += *begin;
    ++begin;
  }
  return total / count;
}
```

catch параметры

- `catch` блок выполняется когда тип исключения совпадает с типом параметра блока **catch**
- **catch-by-value** или **catch-by-reference**.
- Неявные преобразования недопустимы, но полиморфизм работает

```
int main() {
  try {
    throw 2;
  }
  catch (int e) { ... }
}
```

Типы int.

```
int main() {
  try {
    throw 2;
  }
  catch (double e) { ... }
}
```

Неявное преобразование недопустимо.

```
int main() {
  try {
    throw Gorilla();
  }
  catch (Gorilla &e) {
    ...
  }
}
```

Норм, передача по ссылке.

```
int main() {
  try {
    throw Duck();
  }
  catch (Bird &e) {
    ...
  }
}
```

Полиморфизм!

Несколько catch блоков

- try блок может иметь несколько catch блоков.
- Будет использоваться тот блок, для которого тип совпадает
- Используйте “ …” для того чтобы обработать любой тип.

```
int main() {
  try {
    if (/*something*/) {
      throw 4;
    }
    if (/*something*/) {
      throw 2.0;
    }
    if (/*something*/) {
      throw 'a';
    }
    if (/*something*/) {
      throw false;
    }
  }
  catch (int x) { }
  catch (double d) { }
  catch (char c) { }
  catch (...) { }
}
```

Exercise: Exceptions 1

- Что выведет этот код?

```
class GoodbyeError { };
void goodbye() {
  cout << "goodbye called\n";
  GoodbyeError e; throw e;
  cout << "goodbye returns\n";
}
```

```
class HelloError { };
void hello() {
  cout << "hello called\n";
  goodbye();
  cout << "hello returns\n";
}
```

```
int main() {
  try {
    hello();
    cout << "done\n";
  }
  catch (HelloError &he) {
    cout << "caught hello\n";
  }
  catch (GoodbyeError &ge) {
    cout << "caught goodbye\n";
  }
  cout << "main returns\n";
}
```

Solution: Exceptions 1

- Что выведет этот код?

```
class GoodbyeError { };
void goodbye() {
  cout << "goodbye called\n";
  GoodbyeError e; throw e;
  cout << "goodbye returns\n";
}
```

```
class HelloError { };
void hello() {
  cout << "hello called\n";
  goodbye();
  cout << "hello returns\n";
}
```

```
int main() {
  try {
    hello();
    cout << "done\n";
  }
  catch (HelloError &he) {
    cout << "caught hello\n";
  }
  catch (GoodbyeError &ge) {
    cout << "caught goodbye\n";
  }
  cout << "main returns\n";
}
```

```
hello called
goodbye called
caught goodbye
main returns
```

Exercise: Exceptions 2

- Что выведет этот код?

```
class GoodbyeError { };
void goodbye() {
  cout << "goodbye called\n";
  GoodbyeError e; throw e;
  cout << "goodbye returns\n";
}
```

```
class HelloError { };
void hello() {
  cout << "hello called\n";
  try { goodbye(); }
  catch (GoodbyeError &ge) {
    throw HelloError();
  }
  cout << "hello returns\n";
}
```

```
int main() {
  try {
    hello();
    cout << "done\n";
  }
  catch (HelloError &he) {
    cout << "caught hello\n";
  }
  catch (GoodbyeError &ge) {
    cout << "caught goodbye\n";
  }
  cout << "main returns\n";
}
```

Solution: Exceptions 2

- Что выведет этот код?

```
class GoodbyeError { };
void goodbye() {
  cout << "goodbye called\n";
  GoodbyeError e; throw e;
  cout << "goodbye returns\n";
}
```

```
class HelloError { };
void hello() {
  cout << "hello called\n";
  try { goodbye(); }
  catch (GoodbyeError &ge) {
    throw HelloError();
  }
  cout << "hello returns\n";
}
```

```
int main() {
  try {
    hello();
    cout << "done\n";
  }
  catch (HelloError &he) {
    cout << "caught hello\n";
  }
  catch (GoodbyeError &ge) {
    cout << "caught goodbye\n";
  }
  cout << "main returns\n";
}
```

```
hello called
goodbye called
caught hello
main returns
```

Exercise: Exceptions 3

- Что выведет этот код?

```
class Error {
  string msg;
public:
  Error(const string &s) : msg(s) { }
  const string &get_msg() { return msg; }
};
```

```
void goodbye() {
  cout << "goodbye called\n";
  throw Error("bye");
  cout << "goodbye returns\n";
}
```

```
void hello() {
  cout << "hello called\n";
  try { goodbye(); }
  catch (Error &e) { throw Error("hey");}
  cout << "hello returns\n";
}
```

```
int main() {
  try {
    hello();
    cout << "done\n";
  }
  catch (Error &e) {
    cout << e.get_msg();
    cout << endl;
  }
  catch (...) {
    cout << "unknown error\n";
  }
  cout << "main returns\n";
}
```

Solution: Exceptions 3

- Что выведет этот код?

```
class Error {
  string msg;
public:
  Error(const string &s) : msg(s) { }
  const string &get_msg() { return msg; }
};
```

```
void goodbye() {
  cout << "goodbye called\n";
  throw Error("bye");
  cout << "goodbye returns\n";
}
```

```
void hello() {
  cout << "hello called\n";
  try { goodbye(); }
  catch (Error &e) { throw Error("hey");}
  cout << "hello returns\n";
}
```

```
int main() {
  try {
    hello();
    cout << "done\n";
  }
  catch (Error &e) {
    cout << e.get_msg();
    cout << endl;
  }
  catch (...) {
    cout << "unknown error\n";
  }
  cout << "main returns\n";
}
```

```
hello called
goodbye called
hey
main returns
```

Exercise: Exceptions 4

- Что выведет этот код?

```
class Error {
  string msg;
public:
  Error(const string &s) : msg(s) { }
  const string &get_msg() { return msg; }
};
```

```
void goodbye() {
  cout << "goodbye called\n";
  throw GoodbyeError();
  cout << "goodbye returns\n";
}
```

```
void hello() {
  cout << "hello called\n";
  try { goodbye(); }
  catch (Error &e) { throw Error("hey");}
  cout << "hello returns\n";
}
```

```
int main() {
  try {
    hello();
    cout << "done\n";
  }
  catch (Error &e) {
    cout << e.get_msg();
    cout << endl;
  }
  catch (...) {
    cout << "unknown error\n";
  }
  cout << "main returns\n";
}
```

Solution: Exceptions 4

- Что выведет этот код?

```
class Error {
  string msg;
public:
  Error(const string &s) : msg(s) { }
  const string &get_msg() { return msg; }
};
```

```
void goodbye() {
  cout << "goodbye called\n";
  throw GoodbyeError();
  cout << "goodbye returns\n";
}
```

```
void hello() {
  cout << "hello called\n";
  try { goodbye(); }
  catch (Error &e) { throw Error("hey");}
  cout << "hello returns\n";
}
```

```
int main() {
  try {
    hello();
    cout << "done\n";
  }
  catch (Error &e) {
    cout << e.get_msg();
    cout << endl;
  }
  catch (...) {
    cout << "unknown error\n";
  }
  cout << "main returns\n";
}
```

```
hello called
goodbye called
unknown error
main returns
```

“В настоящее время не существует стандартных технологий, применение которых обеспечивает предсказуемость и надежность обработки исключений”

“Использующие механизм исключений и при этом корректно работающие программы появляются не случайно, а требуют тщательного проектирования”

С. Мейерс

Зачем нужны исключения?

- Исключение нельзя проигнорировать
- Если в исключительной ситуации функция возвращает код ошибки или флаг статута, то нет никакой гарантии что программа не продолжит свое выполнение и код будет обработан
- В отличие от кода возврата или флага статуса программа в случае исключения немедленно прекратит свое выполнение

Исключения в конструкторе

```
class Image {
public:
    Image (const std::string &image_name);

    ...
};

class AudioClip {
public:
    AudioClip(const std::string audio_name);

    ...
};

class PhoneNumber {
    ...
};
```

Исключения в конструкторе

```
class BookEntry {
public:
    BookEntry(const std::string &name,
              const std::string &address = "",
              const std::string &imageFile = "",
              const std::string &audioFile = "");

    ~BookEntry();
    void addPhoneNumber(const PhoneNumber &number);
private:
    std::string name;
    std::string address;
    std::list<PhoneNumber> numbers;
    Image *image;
    AudioClip *audio;
};
```

Исключения в конструкторе

```
BookEntry(const std::string &name,
          const std::string &address = "",
          const std::string &imageFile = "",
          const std::string &audioFile = ""):
    name(name), address(address),
    image(nullptr), audio(nullptr) {

    if (imageFile != "") {
        image = new Image(name);
    }

    if (audioFile != "") {
        audio = new AudioClip(audioFile);
    }
}
```

Исключения в конструкторе

Что произойдет если исключение случиться, например, здесь?

```
if (audioFile != "") {
    audio = new AudioClip(audioFile);
}
```

Если исключение возникает во время создания объекта, на который должен указывать audio, что удалит объект, на который уже указывает image?

Очевидно что деструктор. Но нет

Исключения в конструкторе

Если исключение возникает во время создания объекта, на который должен указывать audio, что удалит объект, на который уже указывает image?

Деструктор BookEntry никогда не будет вызван! Так как деструктор вызывается только для полностью сконструированных объектов.

```
void testBookEntry() {
    BookEntry *entry = nullptr;
    try {
        entry = new BookEntry("Vasya", "Spb",
"imageFile", "audioFile");
    }
    catch(...) {
        delete entry;
        throw BookEntryException;
    }

    delete entry;
}
```

Исключения в конструкторе

Объект **entry** все равно будет потерян, так как присвоение **entry** произойдет только после успешного завершения оператора **new**. Если исключение произойдет раньше, то **entry** останется **nullprt** и поэтому его удаление в блоке catch не вызовет никаких действий!

```
BookEntry(const std::string &name,
          const std::string &address = "",
          const std::string &imageFile = "",
          const std::string &audioFile = ""):
    name(name), address(address),
    image(nullptr), audio(nullptr) {

    try {
        if (imageFile != "") {
            image = new Image(name);
        }

        if (audioFile != "") {
           audio = new AudioClip(audioFile);
        }
    }
    catch(...) {
        delete image;
        delete audio;
    }
}
```

Дублирование кода

```
BookEntry(...):
      name(name), address(address),
      image(nullptr), audio(nullptr) {

      try {
      ...
      }
      catch(...) {
          delete image;
          delete audio;

          throw BookEntryException;
      }
 }
```

```
~BookEntry()
 {
      delete image;
      delete audio;
 }
```

```
class BookEntry {
...
private:

    void cleanUp()
    {
        delete image;
        delete audio;
    }
...
}
```

Дублирование кода

```
BookEntry(...):
      name(name), address(address),
      image(nullptr), audio(nullptr) {

      try {
      ...
      }
      catch(...) {
          cleanUp();
          throw BookEntryException;
      }
}
```

```
~BookEntry()
 {
      cleanUp();
}
```

const *

```
class BookEntry {
public:
  ...
private:

...
    Image * const image;
    AudioClip * const audio;
};
```

Что не так?

```
class BookEntry {
public:
    BookEntry(const std::string &name,
              const std::string &address = "",
              const std::string &imageFile = "",
              const std::string &audioFile = ""):
        name(name), address(address),
        image(imageFile != "" ? new Image(imageFile) :
nullptr),
        audio(audioFile != "" ? new AudioClip(audioFile) :
nullptr)

    { }

...
};
```

```
BookEntry(const std::string &name,
          const std::string &address = "",
          const std::string &imageFile = "",
          const std::string &audioFile = ""):
        name(name), address(address),
        image(initImage(imageFile)),
        audio(initAudioClip(audioFile))

    { }
```

Фух, вроде работает...

```
Image * initImage(const std::string &
imageName)
{


if (imageName != "")
     return new Image(imageName);
return nullptr;

}

AudioClip * initImage(const
std::string & audioName)
{
try {
     if (audioName != "") {

     return new AudioClip(audioName);

     }

return nullptr;
}
catch (...) {
     delete image;
     throw BookEntryException;
}

}
```

Умные указатели

Smart pointer — это объект, работать с которым можно как с обычным указателем, но при этом, в отличии от последнего, он предоставляет некоторый дополнительный функционал (например, автоматическое освобождение закрепленной за указателем области памяти).

Умные указатели

- Умные указатели призваны для борьбы с утечками памяти, которые сложно избежать в больших проектах.
- Они особенно удобны в местах, где возникают исключения, так как при последних происходит процесс раскрутки стека и уничтожаются локальные объекты.
- В случае обычного указателя — уничтожится переменная-указатель, при этом ресурс останется не освобожденным.
- В случае умного указателя — вызовется деструктор, который и освободит выделенный ресурс.

shared_ptr

shared_ptr реализует подсчет ссылок на ресурс. Ресурс освободится тогда, когда счетчик ссылок на него будет равен 0.

Как видно, система реализует одно из основных правил сборщика мусора.

shared_ptr

```
std::shared_ptr<int> x_ptr(new int(42));
std::shared_ptr<int> y_ptr(new int(13));

// после выполнения данной строчки, ресурс
// на который указывал ранее y_ptr (int(13))
освободится,
// а на int(42) будут ссылаться оба указателя
y_ptr = x_ptr;

std::cout << *x_ptr << "\t" << *y_ptr << std::endl;

// int(42) освободится лишь при уничтожении
последнего ссылающегося
// на него указателя
```

shared_ptr

```
someFunction(std::shared_ptr<Foo>(new Foo), getRandomKey());
```

Этот код может привести к учетке памяти. Потому, что стандарт C++ не определяет порядок вычисления аргументов. Может случиться так, что сначала выполнится new Foo, затем getRandomKey() и лишь затем конструктор shared_ptr. Если же функция getRandomKey() бросит исключение, до конструктора shared_ptr дело не дойдет, хотя ресурс (объект Foo) был уже выделен.

shared_ptr

```
someFunction(std::make_shared<Foo>(), getRandomKey());
```

make_shared возвращает shared_ptr. Этот результат является временным объектом, а стандарт C++ четко декларирует, что временные объекты уничтожаются, в случае появления исключения.

```
class BookEntry {
public:
    BookEntry(const std::string &name,
              const std::string &address = "",
              const std::string &imageFile = "",
              const std::string &audioFile = ""):
        name(name), address(address),
        image(imageFile != "" ? std::make_shared<Image>() :
nullptr),
        audio(audioFile != "" ? std::make_shared<AudioClip>() :
nullptr)
    { }


void addPhoneNumber(const PhoneNumber &number);

private:

    std::string name;
    std::string address;
    std::list<PhoneNumber> numbers;
    shared_ptr<Image> const image;
    shared_ptr<AudioClip>  const audio;
};
```