Лекция XII

Проектирование и обработка ошибок в программах

Какие ошибки бывают?

Классификация ошибок проектирования исполняемых блоков кода (*функции/методы*)

↙

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 специальных функций, 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, как правило, сопровождаются комментариями, поясняющими случаи их возникновения.

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 специальный механизм, называемый *исключениями*, предоставляемый языком программирования.

1. Ошибки не нужны

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

1. Ошибки не нужны

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

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

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

2. Ошибка - вон из программы

Для немедленного прерывания работы программы C++ через библиотеку <**cstdlib**> предоставляет 3 функции:

```
1 void abort();

2 void exit(int status);

3 void quick_exit(int status);
```

и пару констант:

1. **EXIT_SUCCESS** - целочиселенное значение, хранящее код для ОС который показывает, что программа завершилась успешно. Как правило, значение равно нулю;
2. **EXIT_FAILURE** - код для ОС показывающий, что программа прекратила своё выполнение по причине каких-либо проблемам.

2. Ошибка - вон из программы

```
2 void exit(int status);
```

- Прекращет работу программы, вызывая деструкторов **только** для глобальных объектов;
- возращает операционной системе код, переданный в параметре **status**;
- * закрывает все файловые потоки, но не из стандартной библиотеки C++, а из стандартной библиотеки языка C (доступной через <**cstdio**>;
- позволяет установить произвольное число функций-обработчиков, которые будут вызваны перед выполнением самой функции **exit**. Обработчики устанавливаются с помощью функции **atexit**.

Сигнатура функции-обработчика

Обработчиком может быть только функция, которая не принимает ни одного параметра и её возвращаемым значением является тип **void**

2. Ошибка - вон из программы

Пример на **exit**:

```
1  void handler1()
2  { std::cout << "Я первый обработик.\n"; }
3
4  void handler2()
5  {std::cout << "Я второй. Но буду вызван раньше :)\n";}
6
7  int main()
8  {
9    atexit(handler1);
10   atexit(handler2);
11
12   int num;
13   cout << "Введите число от 5 до 10: ";
14   cin >> num;
15
16   if (num < 5 or num > 10) {
17     exit(EXIT_FAILURE);
18   }
19 }
```

2. Ошибка - вон из программы

Особенности завершения программы на C++

После завершения выполнения функции **main** (из неё явно или неявно был сделан возврат с помощью ключевого слова **return**) всегда вызывается функция **exit**. Её аргументом становится значение, возвращённое функцией **main**.

Схематично, порядок вызова функций таков:

```
1 int st = main();
2 exit( st );
```

Конено же, этот порядок нарушается, если функция **exit** была вызвана до завершения работы **main**.

2. Ошибка - вон из программы

```
3 void quick_exit(int status);
```

- Прекращет работу программы, не вызывая **никаких** деструкторов для глобальных/локальных объектов;
- возращает операционной системе код, переданный в параметре **status**;
- позволяет установить произвольное число функций-обработчиков, которые будут вызваны перед выполнением самой функции **quick_exit**. Обработчики устанавливаются с помощью функции **at_quick_exit**;
- сигнатура обработчика аналогична случаю для **exit**.

2. Ошибка - вон из программы

Пример на **at_quick_exit**:

```
1  class Msg
2  {
3  public:
4    Msg(string message) : _msg{message}
5    {}
6
7    ~Msg()
8    { std::cout << _msg << ": dtor called\n"; }
9  private:
10   string _msg;
11 };
12
13 Msg gl_obj{"Глобальный объект"};
14
15 int main()
16 {
17   Msg l_obj{"Локальный объект"};
18   quick_exit(-5); // Замените на *exit* для просмотра
19                   // различий
20 }
```

2. Ошибка - вон из программы

```
1 void abort();
```

- Прекращет работу программы не вызывая никаких деструкторов глобальных/локальных переменных;
- возращает операционной системе код **EXIT_FAILURE**;
- что происходит с файловыми потоками ввода-вывода - зависит от реализации этой функции в стандартной библиотеке;
- не вызывает никаких обработчиков **atexit**/**at_quick_exit**.

2. Ошибка - вон из программы

Пример: проверка файла на успешное открытие

```
1
2 ifstream data_file{"some_data_file.dat"};
3 if ( !data_file.open() ) {
4   cerr << "Не удалось открыть файл\n";
5   abort();
6 }
```

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

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 }
```

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 проблема: а кто гарантирует, что возвращаемые 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.

3. Ошибке - своё значение

Обобщение с использованием составного типа:

```
1  struct ResultOrError
2  {
3    string value;
4
5    bool was_error;
6    string err_msg;
7  };
8
9  ResultOrError res = some_valuable_func(...);
10 if ( !res.was_error ) {
11   cout << "Результат получен: " << res.value << endl;
12 } else {
13   cout << "Ошибка: " << res.err_msg << endl;
14 }
```

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. Причина:\n";
8    // Напечатает: 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    // Убираем ошибочное состояние
9    cin.clear();
10   // Отчищаем поток ввода от группы неправильных ↩
         символов
11   cin.ignore(numeric_limits<streamsize>::max(), '\n');
12 }
13
14 cout << "Введите снова: ";
15 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 (технически, по умолчанию 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   Msg test_obj{"Тестовый объект"};
3   throw CustomError{5, "так надо"}
4 }
5 catch (const CustomError& err) {
6   std::cout << "Исключение перехвачено с кодом "
7             << err.code << " и сообщением: "
8             << err.message << "\n";
9 }
```

Что с локальными объектами

При выбросе исключения, неважно будет оно обработано, или нет, будут вызваны деструкторы всех локальных объектов.

5. Ошибка - исключительная ситуация

C++ определяет две библиотеки для работы с исключениями: **<exception>** и **<stdexcept>**.
Первая из них определяет:

- класс **exception** - его рекомендуется использовать в качетсве **базового** класса для пользовательских исключений;
- функцию **terminate** - функция для немедленного завершения программы;
- функцию **set_terminate** - установить обработчик для необработанных исключений;
- и что-нибудь ещё: https://en.cppreference.com/w/cpp/header/exception

5. Ошибка - исключительная ситуация

Базовый класс используется следующим образом:

```
1  class MyError : public std::exception
2  {
3  public:
4    const char* what()
5    { return "Логичное сообщение об ошибке"; }
6  };
7
8  class CustomError : public std::exception
9  {
10 public:
11   CustomError(std::string message) : _str{message}
12   {}
13
14   const char* what()
15   { return _msg.c_str(); }
16
17 private:
18   std::string _msg;
19 };
```

5. Ошибка - исключительная ситуация

Перехват исключений, определённых способом выше, должен происходить в порядке от **производных** до **базовых**

```
1  try {
2    /* Код, способный выбросить исключение */
3  }
4  catch (const MyError&) {
5    // произошло исключение типа MyError
6  }
7  catch (const CustomError& ex1) {
8    /*место обработки исключений типа CustomError
9      само значение исключения – в переменной ex1*/
10 }
11 catch (const std::exception) {
12   //место обработки исключений общего типа exception
13 }
14 catch ( ... ) {
15   /*место обработки исключений ЛЮБОГО другого типа*/
16 }
```

5. Ошибка - исключительная ситуация

<stdexcept> определяет набор классов-исключений для некоторых типичных ошибок: **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.

5. Ошибка - исключительная ситуация

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

```
1  DynArray1D get_numbers_from_file(string file_name)
2  {
3    ifstream in_file{file_name};
4    DynArray1D 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 << 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  DynArray1D get_numbers_from_file(string file_name)
2  {
3    ifstream in_file{file_name};
4    DynArray1D vec;
5
6    if ( !in_file ) {
7      throw std::logic_error;
8    } else {
9      // чтение данных из файла
10   }
11
12   return vec;
13 }
```

5. Ошибка - исключительная ситуация

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

```
1  DynArray1D get_numbers_from_file(string file_name);
2
3  string f_name;
4  DynArray1D 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 std::logic_error&) {
13     cout << "Файл не существует. "
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.

```
1  // про реализацию — задавайте вопросы, напишем.
2  // Здесь не приводится
3  class NoFileError : public std::exception;
4  class NotEnoughElemsError : public std::exception;
5
6  DynArray1D get_enough_numbers(string file_name,
7                                size_t at_least = 1)
8  {
9    ifstream in_file{file_name};
10   DynArray1D vec;
11
12   if ( !in_file ) {
13     throw NoFileError{"Файл не найден"};
14   } else {
15     // чтение данных из файла
16     if (vec.length() < at_least) {
17         throw NotEnoughElemsError{"Недостаточно элементов!"}
18     }
19   }
20   return vec;
21 }
```

5. Ошибка - исключительная ситуация

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

```
1  DynArray1D get_enough_numbers(string file_name,
2                                size_t at_least = 1);
3
4  std::string f_name;
5  DynArray1D my_arr;
6
7  while (true) {
8    try {
9      std::cout << "\nИмя файла: ";
10     std::cin >> f_name;
11     my_arr = get_enough_numbers(f_name, 10);
12   }
13   catch (const NoFileError& err1) {
14     std::cout << "Проблема с файлом: " << err1.what() << "\n"
15               << "\nВведите другой...\n";
16   }
17   catch (const NotEnoughElemsError& err2) {
18     std::cout << "Файл некоректен: " << err2.what()
19               << "\nВведите другой...\n";
20   }
21 }
```

5. Оператор new и исключения

Начиная со стандарта C++11, оператор **new** по умолчанию выбрасывает исключение **std::bad_alloc**, если выделение памяти по каким-либо причинам невозможно. На примере:

```
1  const unsigned long long arr_sz = 1024 * 1024 * 1024 + ↩
       1000 * 1024 * 1024;
2
3  double *real_array = nullptr;
4  try {
5    real_array = new double[arr_sz];
6
7    for (size_t i = 0; i < arr_sz; ++i) {
8      real_array[i] = 0.75 * (i + 1);
9    }
10 }
11 catch (std::bad_alloc& ex) {
12   std::cerr << ex.what() << std::endl;
13   std::cerr << "не нашлось достаточно памяти"
14             << std::endl;
15 }
```

Методы пользовательских классов можно помечать специальным индикатором - **noexcept**, говорящим о том, что он(метод) никаких исключений при вызове не бросает. На примере простого 3D вектора (в математическом смысле)

```
1  class Vector3D
2  {
3  public:
4    double x, y, z;
5
6    Vector3D() : x{0.0}, y{0.0}, z{0.0}
7    {}
8
9    double length() const noexcept
10   {
11     return std::sqrt(x*x + y*y + z*z);
12   }
13 };
```

```
1  class Vector3D
2  {
3  public:
4    double x, y, z;
5
6    Vector3D() : x{0.0}, y{0.0}, z{0.0}
7    {}
8
9    double length() const noexcept
10   {
11     return std::sqrt(x*x + y*y + z*z);
12   }
13 };
14 //...
15 Vector3D v1;
16 v1.x = 10.5; v1.y = -1.4; v1.z = 5.4;
17 cout << "Длина вектора равна " << v1.length() << endl;
```

Метод **length** определён как *константный* (не меняет никаких полей объекта) и как не выбрасывающий исключений. **noexcept** может помочь компилятору оптимизировать код, содержащий вызов подобных методов (может, но не гарантирует!).