Лекция VIII

25 марта 2017

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++**.

1. Ошибки не нужны

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";
```

2. Ошибка - вон из программы

Вызов функций **exit(целое_значение)** или **abort()**, определённых в заголовочном файле <**cstdlib**>
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 собственных функций!

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 }
```

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
12 char character = 'f';
13 char capital_letter = get_uppercase( character );
14
15 if (capital_letter != 0) {
16   // что-нибудь полезное
17 }
```

3. Ошибке - своё значение

Пример 1: вернуть заглавную букву английского алфавита. Вместо непонятного нуля прописываем некоторую 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

Где подкралась пролема?
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**.

4. Ошибке - собственное состояние

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

4. Ошибке - собственное состояние

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 любого типа данных**, доступного программе (простые типы данных (**int, double, char**, пользовательские структуры и классы, перечисления)
- Если исключение не обработано - программа прекращает работу (аналогично функции **abort**)
- Как правило, исключения нужны в случаях, когда некоторая функция получила такой набор входных данных, при котором она не может выполнить возложенную на неё работу

Вызов (или бросок) исключения осуществляется с помощью ключевого слова **throw**

```
1  struct CustomError
2  {
3    int code;
4    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

Пример: функция задания целочисленнго вектора из файла следующего формата: (количество элементов, элемент 1, элемент 2, ...)

```
1  struct IntVector { unsigned size; int *data; };
2
3  IntVector get_vector_from_file(string file_name)
4  {
5    ifstream in_file{file_name};
6    IntVector vec;
7
8    if ( !in_file.is_open() ) {
9      // Что тут делать вскоре определим
10   } else {
11     in_file >> vec.size;
12     vec.data = new int[vec.size];
13
14     unsigned i = 0;
15     while (in_file) {
16       in_file >> vec.data[i]; ++i;
17     }
18   }
19   return vec;
20 }
```

5. Ошибка - исключительная ситуация

Пример: функция задания целочисленнго вектора из файла. Если файл не может быть открыт - бросаем исключение.
Попытка 1.

```
1  const int NO_FILE_ERROR_CODE = -1;
2  struct IntVector { unsigned size; int *data; };
3
4  IntVector get_vector_from_file(string file_name)
5  {
6    ifstream in_file{file_name};
7    IntVector vec;
8
9    if ( !in_file.is_open() ) {
10     throw NO_FILE_ERROR_CODE;
11   } else {
12     // чтение данных из файла
13   }
14   return vec;
15 }
```

5. Ошибка - исключительная ситуация

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

```
1  try {
2    /* Код, способный выбросить исключение */
3  }
4  catch ( exception_type1 & ex1) {
5    /*место обработки исключений типа exception_type1
6      само значение исключения – в переменной ex1*/
7  }
8  catch ( exception_type2 & ) {
9    /*место обработки исключений типа exception_type2
10     Значение исключения не получаем*/
11 }
12 catch ( 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

Пример: функция задания целочисленнго вектора из файла.
Если файл не может быть открыт - бросаем исключение.
Попытка 1.

```
1  IntVector get_vector_from_file(string file_name);
2
3  string f_name;
4  IntVector my_vec;
5
6  for (int attempts = 0; attempts < 3; ++attempts) {
7    try {
8      cout << "\nВведите имя файла: ";
9      cin >> f_name;
10     my_vec = get_vector(f_name);
11   }
12   catch (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 задания целочисленнго вектора из файла. Если файл не может быть открыт - бросаем исключение.
Попытка 2: вместо значения целого типа - пользовательские типы данных.

```
1  struct IntVector { unsigned size; int *data; };
2  struct NoFileError {};
3  struct NotEnoughElemetsError {};
4
5  IntVector get_vector_from_file(string file_name)
6  {
7    ifstream in_file{file_name};
8    IntVector vec;
9
10   if ( !in_file.is_open() ) {
11     throw NoFileError{};
12   } else {
13     // чтение данных из файла
14     // где-нибудь тут бросаем исключение типа ↩
           NotEnoughElemetsError
15   }
16   return vec;
17 }
```

5. Ошибка - исключительная ситуация

Пример: функция задания целочисленнго вектора из файла. Если файл не может быть открыт - бросаем исключение. Попытка 2.

```
1  IntVector get_vector_from_file(string file_name);
2
3  string f_name;
4  IntVector my_vec;
5
6  while (true) {
7    try {
8      cout << "\nИмя файла: ";
9      cin >> f_name;
10     my_vec = get_vector(f_name);
11   }
12   catch (NoFileError & ) {
13     cout << "Файл не найден. Попробуйте ещё раз...\n";
14   }
15   catch (NotEnoughElemetsError &) {
16     cout << "Файл некоректен: не хватает элементов."
17             "Введите другой...\n";
18   }
19 }
```

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**.

Базовая работа с ними одинакова:

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

5. Ошибка - исключительная ситуация

Пример: функция получение элемента массива по положительному и отрицательному индексу. Где проблема?

```
1 double get_val_from_array(double *arr,
2                 unsigned arr_size, int pos)
3 {
4   if ( pos < 0 )
5     return arr[arr_size + pos];
6   else
7     return arr[pos];
8 }
```

5. Ошибка - исключительная ситуация

Пример: функция получение элемента массива по положительному и отрицательному индексу.
Проверка границ диапазона.

```
1  #include <stdexcept>
2
3  double get_val_from_array(double *arr,
4                unsigned arr_size, int pos)
5  {
6    if ( (-pos) > int(arr_size) || pos >= int(arr_size) {
7      throw range_error{"Индекс выходит за границы массива"};
8    }
9
10   if ( pos < 0 ) {
11     return arr[arr_size + pos];
12   } else {
13     return arr[pos];
14   }
15 }
```

Шаблонное (обобщённое) программирование - как заставить компилятор писать код вместо себя

Шаблоны в C++

Шаблоны - механизм языка C++, позволяющий переложить конкретную реализацию функций / классов для разичных типов данных на компилятор.

Общий синтаксис объявления шаблонной функции:

```
1 template <typename параметр_типа1,
2           [typename параметр_типа2, ...]>
3 возвращаемое_значение имя_функции( список_аргументов )
4 {
5   тело_функции
6 }
```

Ранее вместо ключевого слова **typename** испольовалось слово **class**.

Пример: сложение двух аргументов

```
1  template <typename T>
2  T add_vars(const T first, const T second)
3  {
4    return first + second;
5  }
6
7  // Тип данных для шаблонной функции можно не указывать,
8  // если компилятор в состоянии самостоятельно выбрать
9  // правильный вариант
10 cout << "Сложение целых: "<< add_vars(1, 5) << "\n";
11 cout << "Сложение действительных: "<< add_vars(5.78, 11.04) <<↩
       "\n";
12
13 string s1 = "Начало +", s2 = " окончание";
14 // Явно указываем тип данных для шаблонной функции
15 cout << "Сложение строк: "<< add_vars<string>(s1, s2) << "\n";
16
17 char str1[] = "Не ", str2[] = "смогу";
18 // Последную строчку компилятор не пропустит, так как
19 // сложение не определено для массивов в стиле C
20 // cout << "Сложение строк2: "<< add_vars(str1, str2) << "\n";
```

Пример: получение элемента массива для произвольного типа данных.

```
1  #include <stdexcept>
2
3  template <typename T>
4  T get_val_from_array(T *arr,
5                       unsigned arr_size, int pos)
6  {
7    if ( (-pos) > int( arr_size ) || pos >= int( arr_size ) ) {
8      throw range_error{"Индекс выходит за границы массива"};
9    }
10
11   if ( pos < 0 ) {
12     return arr[arr_size + pos];
13   } else {
14     return arr[pos];
15   }
16 }
```

Пример: получение элемента массива для произвольного типа данных.

```
1  #include <stdexcept>
2
3  template <typename T>
4  T get_val_from_array(T *arr,
5                       unsigned arr_size, int pos);
6
7  int i_arr[5] = {1, 4, 6, 78, 9};
8  float f_arr[5] = {1.5, 2.4, 3.4, 8.9, 10.05};
9
10 cout << "1-ый элемент: " << get_val_from_array(i_arr, 5, 0) <<↩
       '\n';
11 cout << "-2-ой элемент : " << get_val_from_array(f_arr, 5, -2)↩
       << '\n';
```

Пример: построение стека через шаблон.

```
 1 template <typename DataType>
 2 struct StackNode
 3 { DataType data;  StackNode *next; };
 4
 5 template <typename DType>
 6 class MyStack
 7 {
 8 public:
 9   MyStack() { head = nullptr; stack_size = 0; }
10   ~MyStack() { remove_me(head); }
11
12   bool push(DType dt);      // Добавить новый элемент в стек
13   DType* pop();             // Забрать вершинный элемент стека
14   size_t current_size(); // узнать размер стека
15
16 private:
17   StackNode<DType> *head;
18   size_t stack_size;
19
20   void remove_me(StackNode<DType> *elem);
21 };
```

Пример: построение стека через шаблон - реализация метода добавления элемента (**push**).

```
1  template <typename DType>
2  bool MyStack<DType>::push(DType dt)
3  {
4    // Создаём новый элемент стека, проверяя
5    // выделение памяти под него
6    StackNode<DType> *new_elem = new (nothrow) StackNode<DType>{↩
         dt, head};
7    if ( new_elem == nullptr ) { return false; }
8
9    // Сдвигаем вершину стека на новый элмент
10   head = new_elem;
11
12   ++stack_size;
13   return true;
14 }
```

Пример: построение стека через шаблон - реализация метода удаления элемента (**pop**).

```
1  template <typename DType>
2  DType* MyStack<DType>::pop()
3  {
4    // Если стек — пуст, возращать нечего
5    if (head == nullptr) { return nullptr; }
6
7    // Запоминаем данные вершины стека
8    DType *head_val = new (nothrow) DType;
9    *head_val = head->data;
10
11   // Передвигаем вершину стек на следующий элемент
12   // и удаляем текущий
13   StackNode<DType> *elem = head;
14   head = elem->next;
15   delete elem;
16   --stack_size;
17
18   return head_val;
19 }
```

Пример: построение стека через шаблон - реализация **remove_me, current_size**

```
1  template <typename DType>
2  void MyStack<DType>::remove_me(StackNode<DType> *elem)
3  {
4    if (elem == nullptr) {
5      return;
6    }
7
8    StackNode<DType> *next_elem = elem->next;
9    delete elem;
10   remove_me(next_elem);
11 }
12
13 template <typename DType>
14 size_t MyStack<DType>::current_size()
15 {
16   return stack_size;
17 }
```

Пример: использование шаблонного класса

```
1  template <typename DType>
2  class MyStack
3  {
4  public:
5    bool push(DType dt); // Добавить новый элемент в стек
6    DType* pop();        // Забрать вершинный элемент стека
7    // Остальное — см. выше
8  };
9
10 // Создаём стек целых
11 MyStack<int> int_stack;
12 int_stack.push(1);
13 int_stack.push(2);
14 int_stack.push(3);
15
16 cout << "Элементы в стеке целых чисел:\n";
17 int *stack_val = int_stack.pop();
18 while (stack_val != nullptr) {
19   cout << *stack_val << '\n';
20   delete stack_val;
21
22   stack_val = int_stack.pop();
23 }
```

Пример: использование шаблонного класса

```
1  struct MaterialPoint
2  {
3    int x, y, z;
4    double weight;
5  };
6
7  // Создаём стек для элементов структур
8  MyStack<MaterialPoint> mp_stack;
9  mp_stack.push( MaterialPoint{4, 5, 6, 8.8} );
10 mp_stack.push( MaterialPoint{1, 2, 11, 15.455} );
11 mp_stack.push( MaterialPoint{-5, -4, 3, 3.42} );
12
13 cout << "Текущий первый элемент в стеке структур:\n";
14 MaterialPoint *point = mp_stack.pop();
15 // Напечатает: (-5, -4, 3)
16 cout << '(' << point->x << ' ' << point->y << ' ' << point->z ↩
       << ")\n";
17 // Не нужно значение - удаляем вручную
18 delete point;
19 cout << "Текущий размер стека: " << mp_stack.current_size() <<↩
       '\n';
```