VII

Возможно, не финальная версия 7-й лекции

Анонимные функции

C++: анонимные функции

Не то, чтобы слишком сложно

```
1  bool compare_abs_desc(double left, double right)
2  { return abs(left) > abs(right); }
3
4  double arr = {1.3, -12.5, 4.5,
5                -7.88, 99.125, 7.7};
6  std::sort(arr, arr + 6, compare_abs_desc);
7
8  for (double elem : arr) {
9    std::cout << elem << " ";
10 }
11 std::cout << std::endl;
```

Так ли необходима отдельная функция в глобальном пространстве имён?

C++: анонимные функции

Анонимная (безымянная) функция спешит на помощь

```
1  double arr = {1.3, -12.5, 4.5,
2                -7.88, 99.125, 7.7};
3
4  std::sort(arr, arr + 6,
5    [](double first, double second)
6    { return abs(first) > abs(second); });
7
8  for (double elem : arr) {
9    std::cout << elem << " ";
10 }
11 std::cout << std::endl;
```

C++: анонимные функции

Объекты первого порядка (first-class citizen (objects)) – значения любых типов в C++.

```
1 std::string greetings(std::string username)
2 { return "Привет, " + username + "!"; }
3
4 int my_number = 16;
5 double percent = 0.53;
6
7 std::string my_greet = greetings("Anonymous");
```

В широком смысле – все объекты языка программирования, которые могут быть присвоены переменным и выступать в качестве аргументов и возвращаемых значений функций/методов.

C++: анонимные функции

Свободные функции в C++ не являются объектами первого порядка.

C++: анонимные функции

Термин

Анонимная функция – определение функции, не привязанной к уникальному *идентификатору* и являющейся *объектом первого порядка* в языке программирования. Также известна как: *лямбда-функция*, *лямбда-выражение*.

C++: анонимные функции

Общий синтаксис:

```
1 [<captures>] (params) -> <return_type>
2 { function body };
```

- **<captures>** – блок «захвата» переменных, позволяющий анонимной функции получать досутп к переменным из **локальной** области видимости. Захват должен быть указан явно;
- **params** – параметры функции. Всё, как обычно;
- **<return_type>** – необязательная конструкция (включая стрелку до указания типа), которая явно задаёт тип возвращаемого значения из анонимной функции. В общем случае, тип возвращаемого значения выводится компилятором автоматически;
- **function body** – тело функции со всеми нужными инструкциями.

C++: анонимные функции

Анонимная функция может быть присвоена переменной

```
1 auto fn1 = [] (int num) { return 10 + num; };
2
3 cout << fn1(3) << ", " << fn1(-77) << endl;
4
5 auto fn2 = [] (double num = 4.5)
6 { return num + sqrt(num); };
7
8 cout << fn2() << ", " << fn2(16.16)
9      << ", " << fn2(-5.5) << endl;
```

Пример захвата

```
1  {
2    int state = 0;
3    string base_str = "Log: ";
4
5    auto fn_log = [base_str, &state] (string whats)
6    {
7      cout << base_str <<  "(state " << state
8           << ") " << whats << endl;
9    };
10
11   fn_log("starting computing");
12   state = 4;
13   fn_log("15% completed");
14 }
```

Переменная **base_str** передаётся в анонимную функцию *по значению* (то есть – копия строки), **state** – *по ссылке*.

C++: анонимные функции

Специальный синтаксис для доступа в анонимной функции ко всем локальным переменным. Первый вариант – захват всех по значению

```
1  {
2    int state = 0, counter = 12;
3    string base_str = "Log: ";
4
5    auto fn_log = [=] (string whats)
6    {
7      cout << base_str <<  "(state " << state
8           << ") " << whats << ", "
9           << counter << endl;
10   };
11
12   fn_log("starting computing");
13   state = 4; counter += 3;
14   fn_log("15% completed");
15 }
```

C++: анонимные функции

Термин

Анонимная функция, которая получает доступ к переменным окружающего контекста, в компьютерных науках называется **замыканием** (closure).

Второй вариант – захват всех по ссылке

```
1  {
2    int state = 0, counter = 12;
3    string base_str = "Log: ";
4
5    auto fn_log = [&] (string whats)
6    {
7      cout << base_str <<  "(state " << state
8           << ") " << whats << ", "
9           << counter << endl;
10   };
11
12   fn_log("starting computing");
13   state = 4; counter += 3;
14   fn_log("15% completed");
15 }
```

Вариант с исключениями конкретных переменных

```
1  {
2    int state = 0, counter = 12;
3    string base_str = "Log: ";
4
5    auto fn_log = [&, base_str] (string whats)
6    {
7      cout << base_str <<  "(state " << state
8           << ") " << whats << ", "
9           << counter << endl;
10   };
11
12   fn_log("starting computing");
13   state = 4; counter += 3;
14   fn_log("15% completed");
15 }
```

C++: анонимные функции

Особенности глоссария C++:

- Под *анонимной функцией* или *лямбда-выражением* понимается сама синтаксическая конструкция:

```
1 [] (int num) { return 10 + num; }
```

- Под *замыканием* (closure) понимается любой объект, созданный с помощью лямбда-выражения:

```
1 auto fn = [] (int num) { return 10 + num; };
```

  fn – замыкание в терминологии C++

- Под *типом замыкания* (closure type) понимается тип, который будет выведен для анонимнойй функции. Под капотом C++ этот тип является *безымянным* классом, который создаётся компилятором в той области видимости, где используется анонимная функция

Проектирование и обработка ошибок в программах

C++: какие ошибки бывают?

Классификация ошибок проектирования исполняемых блоков кода (а-ля *функции* и методы *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.

C++: 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.

C++: технические ошибки II

4. **Ошибке - собственное состояние**: из функции возвращается 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.

C++: 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.

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

C++: 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 возвращать вместо «**???**»?

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 }
```

C++: 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?

Пример 2: **printf** - стандартная функция печати в консоль из стандартной библиотеки языка C.

```
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.

C++: 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 }
```

C++: 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 консоль от неправильных символов
11   cin.ignore(std::numeric_limits<std::streamsize>::max↩
         (), '\n');
12 }
13
14 cout << "Введите снова: ";
15 cin >> rate;
```

C++: 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"};
```

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 }
```

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 }
```

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

```
1  FP003Array get_numbers_from_file(string file_name)
2  {
3    ifstream in_file{file_name};
4    FP003Array vec;
5
6    if ( !in_file ) {
7      // Что тут делать вскоре определим
8    } else {
9      int tmp;
10     while (in_file) {
11       in_file >> tmp;
12       vec.push(tmp);
13     }
14   }
15   return vec;
16 }
```

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

```
1  const int NO_FILE_ERROR_CODE = -1;
2
3  FP003Array get_numbers_from_file(string file_name↩
       )
4  {
5    ifstream in_file{file_name};
6    FP003Array vec;
7
8    if ( !in_file ) {
9      throw NO_FILE_ERROR_CODE;
10   } else {
11     // чтение данных из файла
12   }
13   return vec;
14 }
```

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

```
1  FP003Array get_numbers_from_file(string file_name);
2
3  string f_name;
4  FP003Array 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 }
```

C++: 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.

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  FP003Array get_enough_numbers(string file_name, size_t ↩
       at_least = 1)
7  {
8    ifstream in_file{file_name};
9    FP003Array 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 }
```

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

```
1  FP003Array get_enough_numbers(string file_name, size_t ↩
       at_least = 1);
2
3  std::string f_name;
4  FP003Array 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 }
```

C++: 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() – возращающий 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.