Лекция X

Ещё пара фактов про исключения в C++
(не вошедших в предудующую лекцию)

1. 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 } catch (std::bad_alloc& ex) {
11   std::cerr << ex.what() << std::endl;
12   std::cerr << "не нашлось достаточно памяти"
13             << std::endl;
14 }
```

1. new и исключения

Если вместо исключения в случае проблем выделения памяти хочется получать **нулевой указатель**, используется специальная версия оператора **new**:

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

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 std::cout << "Длина вектора равна " << v1.length();
```

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!).

Шаблонное (обобщённое) программирование в C++ или как заставить компилятор писать код вместо себя

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

И не только реализацию функций/классов, но и провести часть вычислений, проверок типов на различные условия, и получение, по необходимости, информации, исходя из переданных в момент компиляции объектов.

На первое - *шаблонные функции*

Шаблонные функции

Для начала надо понять проблематику, а именно, зачем появилась необходимость перекладывать реализацию конкретных функций на компилятор. Рассмотрим обмен значениями двух переменных для типов **int**, **double** и **string**:

```
1  void fp_swap(int& lhs, int& rhs)
2  {// префикс "fp" добавлен к имени функции для красоты
3    int tmp = lhs;
4    lhs = rhs;
5    rhs = tmp;
6  }
7
8  void fp_swap(double& lhs, double& rhs)
9  {
10   double tmp = lhs;
11   lhs = rhs;
12   rhs = tmp;
13 }
14
15 void fp_swap(std::string& lhs, std::string& rhs)
16 {
17   std::string tmp = lhs;
18   lhs = rhs;
19   rhs = tmp;
20 }
```

Шаблонные функции

Что видно из кода на предыдущем слайде:

- объявлены три функции; они перегружены для соответствующих типов;
- все они выполняют одни и теже *действия* (в примере - три присваивания), для переменных **разных типов**;
- все они принимают два параметра нужного типа данных;
- строки «**3, 4, 5**», «**10, 11, 12**» и «**17, 18, 19**» - ничем не отличаются друг от друга за исключением **типа временной переменной**

В итоге, три функции можно обобщить псевдокодом:

```
1 void fp_swap(Type& lhs, Type& rhs)
2 {
3   Type tmp = lhs;
4   lhs = rhs;
5   rhs = tmp;
6 }
```

и найти того, кто будет создавать функции для нужных нам типов.

Шаблонные функции

Как следует из заглавного слайда данной темы, создавать функции нам будет компилятор.

Суть шаблонного программирования в C++

Мы определяем общие **действия**, которые должны быть сделаны для некоторых объектов, а вот **могут ли эти действия быть сделаны** для объектов конкретных типов - проверяет уже компилятор.

Общий синтаксис объявления шаблонной функции (псевдокод):

```
1 template <typename Type1, [typename Type2, ...]>
2 return_value func_name( arguments )
3 {
4   func_body
5 }
```

- функция начинается с ключевого слова **template** и **блока** в треугольных скобках;
- в **блоке** указываются **типы как параметры**, для этого используется слово **typename** и псевдоним для типа;

Шаблонные функции

- далее следует обычное определение функии. Только теперь, в аргументах и теле функции можно создавать переменные перечисленных в **блоке** типов;
- количество типов для шаблона - можно считать неограниченым (определяется задачей). Квадратные скобки в первой строке говорят о том, что второй и последующие параметры шаблонной функции - **опциональны**;
- ранее вместо ключевого слова **typename** использовалось слово **class**. Его и сейчас можно использовать, но всё-таки в современном C++ рекомендуется выбирать только **typename**.

Шаблонные функции

И теперь реализуем шаблонную функцию обмена значениями двух переменных:

```
1 template <typename Type>
2 void fp_swap(Type& lhs, Type& rhs)
3 {
4   Type tmp = lhs;
5   lhs = rhs;
6   rhs = tmp;
7 }
```

Одна шаблонная функция, которой, для превращения в реальную функцию в программе, достаточно знать два аспекта:

1. **Тип** передаваемых аргументов (аргументы должны быть одинакого типа).
2. Возможность выполнения **операции присваивания** для объектов этого типа.

Второй пункт - ключевой: если действия (вызов операторов и/или методов), описанные в теле шаблонной функции, не определены для объектов - произойдёт ошибка компиляции.

Шаблонные функции

Использование шаблонной функции:

```
1  int i1 = 5, i2 = 10;
2  fp_swap(i1, i2);
3  cout << i1 << ", " << i1 << '\n';
4
5  double r1 = 1.5, r2 = 8.8;
6  fp_swap(r1, r2);
7  cout << r1 << ", " << r1 << '\n';
8
9  std::string s1 = "str1", s2 = "2str";
10 fp_swap(s1, s2);
11 cout << s1 << ", " << s1 << '\n';
```

Кратко: компилятор видит вызовы шаблонной функции, видит аргументы и их типы, и создаёт реализацию конкретной функции, если действия в теле функции подходят для аргументов.
Тип можно указать явно с помощью следующего синтаксиса:

```
12
13 char sym1 = 'e', sym2 = 'w';
14 fp_swap<char>(sym1, sym2);
15 cout << sym1 << ", " << sym1 << '\n';
```

Шаблонные функции

Использование шаблонной функции:

```
1  int i1 = 5, i2 = 10;
2  fp_swap(i1, i2);
3  cout << i1 << ", " << i1 << '\n';
4
5  double r1 = 1.5, r2 = 8.8;
6  fp_swap(r1, r2);
7  cout << r1 << ", " << r1 << '\n';
8
9  std::string s1 = "str1", s2 = "2str";
10 fp_swap(s1, s2);
11 cout << s1 << ", " << s1 << '\n';
```

Кратко: компилятор видит вызовы шаблонной функции, видит аргументы и их типы, и создаёт реализацию конкретной функции, если действия в теле функции подходят для аргументов.
Тип можно указать явно с помощью следующего синтаксиса:

```
12
13 char sym1 = 'e', sym2 = 'w';
14 fp_swap<char>(sym1, sym2);
15 cout << sym1 << ", " << sym1 << '\n';
```

Шаблонные функции

Использованная простая реализация функции обмена будет работать и для пользовательских классов

```
1 class Point
2 {
3 public:
4   double x, y, z;
5 };
6
7 Point p1 = {3.4, 5.5, 1.2}, p1 = {-1.1, -2.2, -3.3}
8 fp_swap(p1, p2);
9 cout << p1.x << ", " << p2.y << '\n';
```

Всё работает из-за того, что классу предоставляется **оператор присваивания** по умолчанию.

Шаблонные функции

Но оператор присваивания можно и запретить для класса. Демо:

```
1  class Point
2  {
3  public:
4    double x, y, z;
5
6    Point& operator=(const Point&) = delete;
7  };
8
9  // Пример не скомпилируется
10 Point p1 = {3.4, 5.5, 1.2}, p1 = {-1.1, -2.2, -3.3}
11 fp_swap(p1, p2);
12 cout << p1.x << ", " << p2.y << '\n';
```

Будет ошибка компиляции с сообщением об невозможности превратить *шаблонную функцию* в *конкретную* для типа, у которого отсутствует *оператор присваивания*.

Шаблонные функции

Кроме того, компилятору можно дать команду на создание конкретной версии шаблонной функции. Это называется **явным инстанцированием**. Пример:

```
1 template<>
2 void fp_swap(size_t&, size_t&);
```

Функция обмена для типа **size_t** будет создана, хотя ни разу не вызвана для конкретных переменных.

Шаблонные функции

Параметры шаблона могут быть не только **псевданимами** для типа, но и значениями конкретных типов. Единственное условие на значения - они должны быть известны на этапе компиляции:

```
1  template<typename Type, size_t how_many>
2  void repeat_to_cout(const Type& obj)
3  {
4    for (size_t i = 0; i < how_many; ++i) {
5      std::cout << obj << "\n";
6    }
7  }
8
9  int i = 18;
10 repeat_to_cout<int, 20>(i); // Всё хорошо
11 // repeat_to_cout<int, i>(i); // Не получится
```

Пример не особо полезен, но показывает использование нетипового параметра шаблона.

Шаблонные функции

Более интересный пример - автоматический вывод размера *массива в стиле C* при передаче в функцию. Напишем функцию, которая будет складывать все элементы массива и возвращать сумму.

```
1  template<typename Type, size_t N>
2  Type reduce_sum(Type (&array)[N])
3  {
4    Type sum{};
5    for (size_t i = 0; i < N; ++i) {
6      sum += array[i];
7    }
8    return sum;
9  }
10
11 int arr1[] = {1, 4, 5, 6, 7, 8, 9};
12 double arr2[] = {5.5, 4.4, 3.3, 2.2, 1.1, 0.999};
13
14 cout << "sum of arr1 is " << reduce_sum(arr1) << "\n";
15 cout << "sum of arr2 is " << reduce_sum(arr2) << "\n";
```

Хинт: работает за счёт передачи массива в функцию по ссылке.

Шаблонные функции

Или функцию, которая возращает размер статитеческого *массива в стиле C*.

```
1  template<typename Type, size_t N>
2  size_t array_size(Type (&arr)[N])
3  {
4    return N;
5  }
6
7  int arr1[] = {1, 4, 5, 6, 7, 8, 9, 11, 13, 15, 17};
8  double arr2[] = {5.5, 4.4, 3.3, 2.2, 1.1, 0.999};
9
10 cout << "size of arr1 is " << array_size(arr1) << "\n";
11 cout << "size of arr2 is " << array_size(arr2) << "\n";
```

Шаблонные классы

И конечно же C++ позволяет создавать шаблонные классы.
Принцип - тот же, что и с функциями: мы можем написать «прототип» класса, который будет делать одинаковые **действия** для **объектов** разных типов.
Для начала, класс, который хранит два значения **разных** типов.

```
1  template<typename T1, typename T2>
2  class Pair
3  {
4  public:
5    T1 first;
6    T2 second;
7  }
8
9  Pair<int, double> p1 = {5, 789.123};
10 Pair<char, std::string> p2 = {'f', "текст"};
11
12 cout << p2.second << '\n';
```

Стоит заметить, что подобный класс есть в стандартной библиотеке C++; называется **pair** и обитает в заголовочном файле **<utility>**.

Шаблонные классы. Кубическая решётка

Теперь класс, реализующий кубическую решётку чего-нибудь (любых конкретных узлов).

```
1  template<typename Cell>
2  class CubicLattice
3  {
4  public:
5    CubicLattice(size_t lx, size_t ly, size_t lz);
6    ~CubicLattice();
7
8    Cell& operator()(size_t i, size_t j, size_t k) const;
9
10   size_t count_x() const;
11   size_t count_y() const;
12   size_t count_z() const;
13   size_t total_count() const;
14
15 private:
16   Cell *_arr;
17   size_t _lx, _ly, _lz;
18 };
```

Шаблонные классы. Кубическая решётка

Методы шаблонного класса также могут быть определены вне его объявления с помощью следующего синтаксиса:

```
1  template<typename Cell>
2  CubicLattice<Cell>::CubicLattice(size_t lx, size_t ly, size_t ↩
       lz)
3      : _lx{lx}, _ly{ly}, _lz{lz}
4  {
5    _arr = new Cell[_lx * _ly * _lz];
6  }
7
8  template<typename Cell>
9  CubicLattice<Cell>::~CubicLattice()
10 {
11   delete[] _arr;
12 }
13
14 template<typename Cell>
15 Cell& CubicLattice<Cell>::operator()(size_t i, size_t j, ↩
       size_t k) const
16 {
17   return _arr[i + j * _lx + k * (_lx * _ly)];
18 }
```

Шаблонные классы. Кубическая решётка

Методы шаблонного класса:

```
1  template<typename Cell>
2  size_t CubicLattice<Cell>::count_x() const
3  {  return _lx; }
4
5  template<typename Cell>
6  size_t CubicLattice<Cell>::count_y() const
7  {  return _ly; }
8
9  template<typename Cell>
10 size_t CubicLattice<Cell>::count_z() const
11 {  return _lz; }
12
13 template<typename Cell>
14 size_t CubicLattice<Cell>::total_count() const
15 {  return _lx * _ly * _lz; }
```

Шаблонные классы. Кубическая решётка

И его использование:

```
1  struct Point
2  {
3    double x, y, z;
4  };
5
6  // решётка 4x5x10
7  CubicLattice<double> real_latt{4, 5, 10};
8  real_latt(0, 0, 0) = 456;
9  real_latt(1, 2, 10) = -11.89;
10
11 cout << real_latt(1, 2, 10);
12
13 // 7x7x19
14 CubicLattice<Point> pt_latt{7, 7, 19};
15 pt_latt(1, 1, 1) = {0.5, 0.5, 0.5};
```

Константы компиляции (constexpr)

C++ в дополнении к константам времени выполнения (переменные, объявленные вместе с **const**) позволяет определить конктанты, которые всегда будут вычислены **на этапе компиляции**. Они объявляются с помощью ключевого слова **constexpr**.

```
1 constexpr size_t SZ = 100;
2 constexpr double R_CONST = 3.14;
```

В определённых случаях, такие константы времени компиляции полезны для шаблонного программирования.
В дополнени C++ пошёл ещё дальше, и позволяет объявлять **constexpr**-функции:

```
1 constexpr int factorial(int n)
2 {
3     return n <= 1 ? 1 : (n * factorial(n - 1));
4 }
```

Такие функции достаточно ограничены в C++11 стандарте, но в последующих (C++14 / C++17) расширены. Подробнее: https://en.cppreference.com/w/cpp/language/constexpr