VIII

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

Не термин

Шаблоны (templates) - механизм языка C++, позволяющий описать прототип функции или составного объекта (структуры, классы, **union** и т.п.), реализующих одинаковый функционал для различных типов данных. Причём, конкретная реализация прототипа в компилируемый блок C++ делегирована компилятору.

В случае (свободных) функций этот механизм даёт возможность описать прототип для потенциально бесконечного числа перегрузок конкретной функции. С шаблонных функций и начнём знакомство с этой стороной C++.

Шаблонные функции

Для начала надо понять проблематику, а именно, зачем появилась необходимость перекладывать реализацию конкретных функций на компилятор. Для демонстрации идеи, используем дополнительный тип:

```
1  class Sector
2  {
3  public:
4    Sector operator*(int factor)
5    {
6      double new_val = abs(angles * factor);
7      if (new_val > 360.0) { new_val = 360.0; }
8
9      Sector sec{new_val};
10     return sec;
11   }
12
13   double angles = 0.0;
14 };
```

Совсем простой класс, представляющий *сектор круга на плоскости* и умеющий увеличивать угол сектора путём умножения на произвольное целое число. **Не предназначен** для использования в реальных программах.

Шаблонные функции

И для примера запишем функции, которые будут умножать переданные ей значения типов **int**, **double** и **Sector** на произвольный множитель :

```
1 int multiply_by(int value, int factor)
2 { return value * factor; }
3
4 double multiply_by(double value, int factor)
5 { return value * factor; }
6
7 Sector multiply_by(Sector value, int factor)
8 { return value * factor; }
```

Что видно из кода:

- определены три разных *перегрузки* одной функции **multiply_by**;
- все они выполняют одни и теже *действия* (в примере – перемножение со значением типа **int**), для переменных **разных типов**;
- все они принимают два параметра задуманных типов;
- тела функций друг от друга совсем не отличаются (с точностью до названия *параметров* функций.

Шаблонные функции

В итоге, три функции можно обобщить псевдокодом:

```
1 Type multiply_by(Type value, int factor)
2 {
3   return value * factor;
4 }
```

и переложить задачу создания перегрузок для конкретных типов на кого-то другого.

Суть шаблонного программирования

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

Шаблонные функции I

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;
- далее следует обычное определение функии. Только теперь, в аргументах и теле функции можно создавать переменные перечисленных в **блоке** типов;

Шаблонные функции II

- 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**;
- до стандарта **C++11** для задания параметра шаблона использовалось ключевое слово **class**. Можно использовать и сейчас, но первый вариант больше выражает семантику (смысл) задания параметр-а(-ов) для тип-а(-ов).

Шаблонные функции

И вместо трёх перегрузок выше появляется одна шаблонная функция:

```
1 template <typename Type>
2 Type multiply_by(const Type& value, int factor)
3 {
4   return value * factor;
5 }
```

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

1. **тип** первого аргумента, который совпадает с типом возвращаемого значения.
2. возможность выполнения **операции перемножения** объекта этого типа со значением типа **int**.

Второй пункт – ключевой: если действия (вызов операторов и/или методов), описанные в теле шаблонной функции, не определены для объектов некоторого типа – далее компилятор не сможет создать нашу функцию для него.

Шаблонные функции

Работа компилятора с шаблонной функцией заключается в следующем:

1. компилятор проверяет шаблонную функцию на корректность: отсутствие синтаксических ошибок в её определении;
2. компилятор создаёт реальную функцию (т.е. создаёт исполняемый код) для использования в рамках программы для конкретного типа данных. Это называется – **инстанцированием** шаблона.

Важный момент

Инстанцирование происходит **тогда и только тогда**, когда в коде встречается *вызов* шаблонной функции.

Шаблонные функции

Использование шаблонной функции:

```
1 cout << multiply_by(10, 2) << endl;
2
3 double real = 55.72;
4 cout << multiply_by(real, -3) << endl;
5
6 Sector sc = { 34 };
7 next_sc = multiply_by(sc, 4);
8 cout << next_sc.angles << endl;
```

Шаблонные функции: технические детали

- компилятор встречает вызов шаблонной функции (для примера – **строка 1** предыдущего слайда);
- поскольку делается вызов шаблонной функции **multiply_by**, он смотрит на её аргумент, который в определении функции соответствует параметрическому типу;
- этот аргумент (число **10**) имеет тип **int**;
- компилятор *инстанцирует* шаблонную функцию в конкретную реализацию для него.

Аналогично и для строк **4, 7** – происходит создание реализации функции для двух других типов. Такое инстанцирование в некоторых книгах называют **неявным**.

Шаблонные функции: технические детали

В противоположность неявному, существует и **явное инстанцирование** – когда тип мы сами указываем компилятору в момент вызова шаблонной функии. В таком случае пример меняется на:

```
1 cout << multiply_by<int>(10, 2) << endl;
2
3 double real = 55.72;
4 cout << multiply_by<double>(real, -3) << endl;
5
6 Sector sc = { 34 };
7 next_sc = multiply_by<Sector>(sc, 4);
8 cout << next_sc.angles << endl;
```

Шаблонные функции: технические детали

Ещё один аспект инстанцирования — возможность заставить компилятор сделать реализацию шаблонной функции для нужного типа, при этом не делая ни одного вызова этой функции с этим типов в коде. Для примера, код вида

```
1 template
2 short multiply_by<short>(short, int);
```

– создаст функцию **multiply_by** для типа **short**, даже несмотря на отсутствие её прямого вызова. Совсем *явное* инстанцирование.

Больше применяется при проектировании и реализации отдельных библиотек с различным функционалом на C++, но если где-нибудь встретите, то вопросов уже не вызовет подобный код. Вероятно...

Шаблонные функции: технические детали

Несмотря на то, что по задумке шаблонные функции определяют одинаковый набор действий для различных комбинаций параметров-типов, иногда возникает необходимость для одного или нескольких типов данных предоставить другой набор действий для выполнения задачи, которую решает шаблонная функция.
Для примера, в первом семестре рассматривался класс **FP003Array**, представляющий собой динамический массив. И захотелось определить для него операцию *умножения на целочисленный множитель* как умножение каждого элемента массива на него. Но использовать перегрузку оператора умножения не хочется. Поскольку семантика такой операции неоднозначна. В частности, в некоторых языках программирования умножение массива на целое число ведёт к его расширению и копированию. . .

Шаблонные функции: технические детали

Тогда можно использовать уже придуманную шаблонную функцию **multiply_by** (используем знакомое название для знакомых действий). Реализовать такую задумку можно с помощью **специализации** шаблонной функции для типа **FP003Array**. Выглядит это так:

```
1  template <>
2  FP003Array multiply_by<FP003Array>(FP003Array inst,
3                                     int factor)
4  {
5    const size_t len = inst.length();
6    FP003Array other{len};
7    for (size_t i = 0; i < len; i++) {
8      other[i] = inst[i] * factor;
9    }
10
11   return other;
12 }
```

Шаблонные функции: технические детали

Специализация шаблонной функции:

- указываем ключевое слово **template**;
- везде вместо параметра-типа указываем конкретный тип данных;
- тело данной специализации отличается от общей шаблонной функции.

Шаблонные функции: технические детали

И пример использования:

```
1 FP003Array my_vec = {4, -3, 5, 1, 8, 9, 11};
2 FP003Array vec2 = multiply_by(my_vec, 5);
3
4 for (const int elem : vec2) {
5   cout << "|" << elem << "| ";
6 }
7 cout << endl;
```

Шаблонные функции: технические детали

Шаблонная функция может быть разделена на две части – *объявление* и *определение*.

```
1  // Объявление, имена параметров — опциональны
2  template <typename Type>
3  Type multiply_by(const Type&, int);
4
5  // какой-то код...
6
7  // Определение шаблонной функции:
8  template <typename Type>
9  Type multiply_by(const Type& value, int factor)
10 {
11   return value * factor;
12 }
```

Шаблонные функции: технические детали

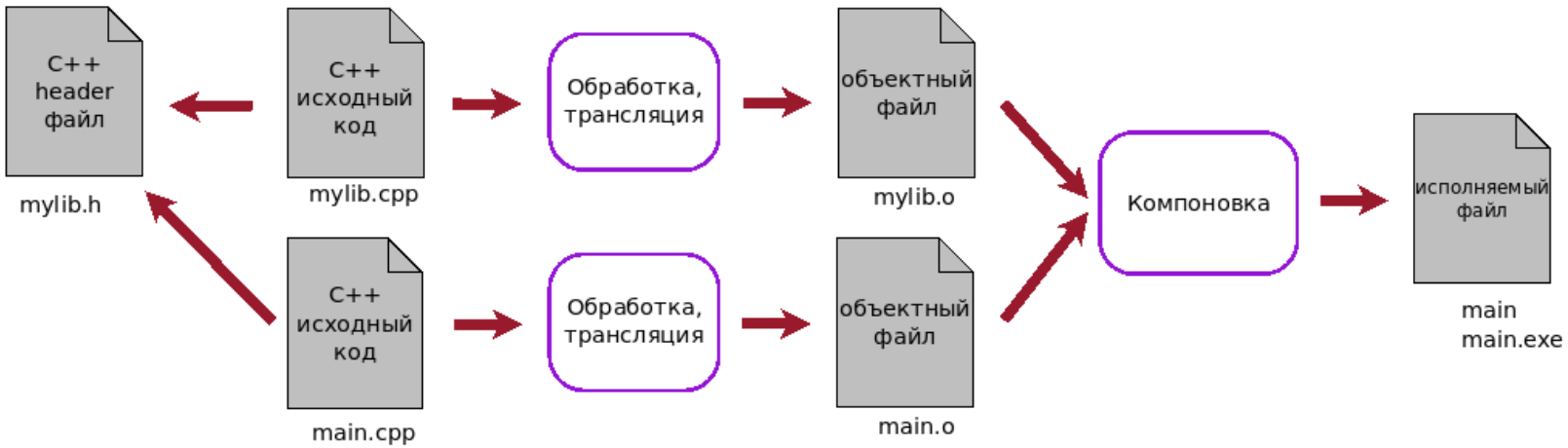
И совсем уж технический момент, относящийся к тому, как компилятор создаёт исполняемые файлы.

Важный момент

При наличии шаблонных функций, для создания *объектного файла* компилятор должен иметь доступ к их реализации. Прямое следствие из этого правила – определения шаблонных функций, в общем случае, не могут быть вынесены в отдельный ***.cpp** файл

Шаблонные функции: технические детали

Для того, чтобы понять о чём предыдущий слайд, вспомним самую привычную схему для разделения кода без шаблоннов на C++:



Есть файл **mylib.h** с объявлениями функций/типов, **mylib.cpp** – с их определениями и файл **main.cpp**, который как-то эти функции/типы использует. **mylib.cpp** и **main.cpp** преобразуются в отдельные объектные файл и затем компоновщик их «склеивает» в исполняемый файл.

Шаблонные функции: технические детали

Если добавить объявление шаблонной функции в **mylib.h**, а её реализацию – в **mylib.cpp**, то при обработке файла **main.cpp** компилятор не сможет найти достаточно информации, как из прототипа построить конкретную функцию (содержимое **mylib.cpp** недоступно в процессе обработки **main.cpp**).

Следствие

Говоря техническим языком, определения шаблонных функций должны всегда присутствовать в том заголовочном файле, где сделаны их объявления.
Другая формулировка: определение шаблонной функции всегда должно быть доступно в каждой *единице трансляции*, в которой происходит использование (вызов) этой функции.

Шаблонные функции

Параметры шаблона могут быть не только **псевдонимами** для типа, но и значениями конкретных типов. Главное условие на значения – они должны быть вычислимы 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, 5>(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 возращать 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";
```

Сравните со страшным способом:

```
1  int arr3[] = {1, 4, 5, 6, 7, 8, 9, 11, 13, 15, 17};
2  size_t arr_size = sizeof(arr3) / sizeof(arr3[0]);
3
4  cout << "size of arr3 is " << arr_size << "\n";
```

, который ещё и ограничен в использовании только той областью, где был массив определён.

Шаблонные типы

C++ позволяет создавать шаблонные типы данных. Принцип – тот же, что и с функциями: мы можем написать «прототип» типа, который компилятор будет превращать в конкретные типы.

Для примера, структура, позволяющая хранить три значения разных типов.

```
1 template<typename T1, typename T2, typename T3>
2 struct Trio
3 {
4   T1 first;
5   T2 second;
6   T3 third;
7 };
```

Аналогично функциям, добавилась шапка с ключевым словом **template**, списком типовых параметров. В остальном – обычная структура, только вместо конкретных типов указаны параметрические псевдонимы (**T1, T2, T3**).

Шаблонные типы

При использовании шаблонного типа не обойтись без явного указания всех параметров:

```
1 Trio<int, double, string> item =
2                             {455, -3.33, "Ya stroka"};
3 cout << item.first + item.second << endl;
4
5 item.first *= 3;
6
7 item.third.replace(0, 2, "Kakaya-to");
8 cout << item.third << end;
```

Аналогично шаблонным функциям, конкретный тип инстанцируется компилятором только, когда в коде используется шаблонный тип с заданными параметрами (**строка 1**).

Шаблонные типы

```
1 Trio<int, double, string> item =
2                             {455, -3.33, "Ya stroka"};
3
4 Trio<int, int, chat> other =
5                           {5, -5, 'W'};
6 cout << other.third << ": "
7      << other.first * other.second << endl;
```

В скромном примере компилятором будут созданы **две независимые структуры**: для набора типов **int, double, string** и для набора типов **int, int char**. Можно сформулировать вывод о том, что имя реализации *шаблонного типа* определяется не только его названием, но и именами всех шаблонных параметров.

Шаблонные типы

Конечно же, для шаблонных типов в качестве параметров шаблона можно использовать и конкретные значения. Например, шаблонная структура для определения произвольной *статической матрицы* заданного размера

```
1  template <typename TElem, size_t N>
2  struct StaticMatrix
3  {
4    TElem data[N][N];
5  };
6
7  using IntMatrix4x4 = StaticMatrix<int, 4>;
8  using RealMatrix3x3 = StaticMatrix<double, 4>;
9
10 RealMatrix3x3 matr = {{4, -3, 5}, {0, 1, 2}, {9, 0, ↩
       -4}};
11 cout << matr.data[0][1] * matr.data[1, 2] << endl;
```