Семинар #4: Шаблоны.

Шаблоны (англ. *templates*) – это механизм языка C++, который позволяет писать обобщённый код, то есть код, который может работать с различными типами данных.

Шаблоны функций

Написание обобщённых функций

Одна из проблем языков программирования, не поддерживающих шаблоны или аналогичные механизмы, заключается в необходимости создавать множество схожих функций для каждого типа данных.

В одном из предыдущих семинаров уже рассматривалсь пример функции нахождения модуля числа. В языке C нет шаблонов, поэтому функцию нахождения модуля нужно было написать для каждого типа и дать каждой из этих функций уникальное имя (в стандартной библиотеке C эти функции назывались: `abs` для нахождения модуля числа типа `int`, `fabs` – для чисел типа `double` и т. д.). В результате код на языке C выглядел так:

```
int abs(int x)
{
    if (x < 0)
        return -x;
    return x;
}
double fabs(double x)
{
    if (x < 0)
        return -x;
    return x;
}
// И так далее ещё много раз для всех возможных типов
```

Этот код имеет следующие проблемы:

1. Придётся придумывать и запоминать разные странные названия функций для разных входных типов. Легко случайно ошибиться и вызвать функцию для другого типа, что может привести к ошибке.

2. Приходится писать много похожих функций. Получается очень много повторяемого кода.

В языке C++ есть механизм перегрузки функций, который позволяет улучшить этот код, дав каждой функции одинаковое имя. Перегрузка функций решает первую проблему, но не решает вторую.

```
int abs(int x)
{
    if (x < 0)
        return -x;
    return x;
}
double abs(double x)
{
    if (x < 0)
        return -x;
    return x;
}
// И так далее ещё много раз для всех возможных типов
```

Вторую проблему можно решить, используя *шаблон функции* (для этого понятия есть ещё один эквивалентный термин – *шаблонная функция*). Шаблоны создаются с помощью ключевого слова `template`, после которого в угловых скобках указываются параметры шаблона. Самым частым параметром шаблона является тип, такие параметры обозначаются ключевым словом `typename`. Шаблонная функция вычисления модуля числа будет выглядеть следующим образом:

```
#include <iostream>

template<typename T>
T abs(T x)
{
    if (x < 0)
        return -x;
    return x;
}

int main()
{
    std::cout << abs<int>(-5) << std::endl;       // Напечатает 5
    std::cout << abs<double>(-1.5) << std::endl;  // Напечатает 1.5
}
```

Пояснения по этому коду:

- В данной программе написана шаблонная функция `abs`. Шаблонная функция не является обычной функцией. Это новая абстракция, которая не относится ни к функциям, ни к классам, ни к другим понятиям, с которыми мы сталкивались в этом курсе.
- Шаблонная функция используется для генерации обычных функций. Например, если компилятор увидит в коде:

  ```
  abs<int>
  ```

  то он сгенерирует новую обычную функцию из шаблонной функции `abs`. Для этого компилятор просто возьмёт шаблон функции и везде вмето `T` подставит `int`. Таким образом из шаблона `abs` получится обычная функция `abs<int>`. Процесс получение обычной функции из шаблона называется *инстанцированием*. После того как компилятор получил обычную функцию из шаблонной, он будет её компилировать также как он компилирует другие обычные функции.
- Параметры шаблона задаются в угловых скобках. В данном примере шаблонная функция `abs` имеет один шаблонный параметр под названием `T`, который имеет "тип" `typename`. У шаблонов может быть и несколько параметров разных типов.
- Использование имени `T` для параметра шаблона является общепринятым соглашением. Но никто не запрещает использовать и другие имена. Однако имеется правило хорошего стиля, которое говорит о том, что параметры шаблонов следует именовать с заглавной буквы, например, так:

  ```
  template<typename Cat, typename Dog>
  ...
  ```

Автоматический вывод типа для шаблона функции

Компилятор может сам догадаться какой тип подставить в шаблонную функцияю по типу аргументов. Если мы в программе вызовем шаблонную функцию `abs` вот так:

```
abs(-5)
```

то компилятор сам догадается, что нужно создать функцию `abs<int>` и вызвать её в этом месте.

```
#include <iostream>
template<typename T>
T abs(T x)
{
    if (x < 0)
        return -x;
    return x;
}

int main()
{
    int a = -5;
    std::cout << abs<int>(a) << std::endl;  // Можно указать значение параметра шаблона
    std::cout << abs(a) << std::endl;       // Можно не указывать, тогда компилятор сам
                                            // догадается, что это тип int
}
```

Примеры шаблонов функций

- Шаблон функции `getMax`, для нахождения максимума двух объектов.

  ```
  #include <iostream>
  #include <string>
  using namespace std::string_literals;

  template<typename T>
  T getMax(const T& a, const T& b)
  {
      if (a > b)
          return a;
      return b;
  }

  int main()
  {
      std::cout << getMax(5, 10) << std::endl;           // Напечатает 10
      std::cout << getMax(0.2, 0.1) << std::endl;        // Напечатает 0.2
      std::cout << getMax("Cat"s, "Dog"s) << std::endl;  // Напечатает Dog

  }
  ```

  Обратите внимание, что в эту функцию объекты передаются по константной ссылке, а не по значению. Потому что мы не хотим копировать объекты внутрь функции. Ведь эта функция может работать не только с числами, но и с большими объектами, копирование которых внутрь функции может быть неэффективно.

- Шаблон функции `swap`, для обмена значения двух объектов.

```cpp
#include <iostream>
#include <string>
using namespace std::string_literals;

template<typename T>
void swap(T& a, T& b)
{
    T temp = a;
    a = b;
    b = temp;
}

int main()
{
    int a = 10;
    int b = 20;
    swap(a, b);
    std::cout << a << " " << b << std::endl;  // Напечатает 20 10

    std::string c = "Cat";
    std::string d = "Dog";
    swap(c, d);
    std::cout << c << " " << d << std::endl;  // Напечатает Dog Cat
}
```

- Шаблон функции `sum`, для нахождения суммы элементов вектора.

```cpp
#include <iostream>
#include <string>
#include <vector>
using std::cout, std::endl, std::size_t;

template <typename T>
T sum(const std::vector<T>& v)
{
    T result{};

    for (size_t i = 0; i < v.size(); ++i)
        result += v[i];

    return result;
}

int main()
{
    std::vector<int> a {10, 20, 30, 40, 50};
    cout << sum(a) << endl;  // Напечатает 150

    std::vector<std::string> b {"Cat", "Dog", "Mouse"};
    cout << sum(b) << endl;  // Напечатает CatDogMouse
}
```

Ограничения, накладываемые на тип шаблона функции

Рассмотрим шаблон функции `abs`. С какими типами может работать эта шаблонная функция? Можно ли в эту функцию передать, например, объект типа `std::string`? Если передать в эту функцию строку `std::string`, то компилятор сгенерирует обычную функцию `abs<std::string>` из шаблона, подставив везде `std::string` вместо `T`. После этого он попытается скомпилировать эту функцию, но у него ничего не получится, так как эта функция использует операторы, которых у класса `std::string` нет. Ошибка компиляции произойдёт внутри уже сгенерированной функции, когда к объекту типа `std::string` будет применён оператор, который он не поддерживает.

Всего шаблон `abs` требует следующих операций:

1. Оператор `<` для сравнения с целым числом.
2. Унарный оператор минус (`-`).
3. Копирование. Так как функция `abs` принимает аргумент по значению, то этот аргумент должен скопироваться внутрь функции. Поэтому тип `T` должен поддерживать копирование.

Поскольку у типа `std::string` отсутствуют первые два необходимых оператора, функция `abs` не может быть применена к объектам этого типа. Однако для классов, которые поддерживают все требуемые операции, использование функции `abs` не вызовет никаких проблем.

```
#include <iostream>
#include <string>

template<typename T>
T abs(T x)
{
    if (x < 0)
        return -x;
    return x;
}

struct Alice
{
    int x;
    Alice(int x) : x(x) {}

    Alice(const Alice& a) : x(a.x) {}
    bool operator<(int b) const {return x < b;}
    Alice operator-() const
    {
        Alice result(-x);
        return result;
    }
};

int main()
{
    std::string s = "cat";
    std::string t = abs(s);         // Ошибка компиляции

    Alice a(-5);
    Alice b = abs(a);               // OK
    std::cout << b.x << std::endl;  // Напечатает 5
}
```

Шаблоны функций от нескольких шаблонных параметров

Шаблоны могут иметь и несколько параметров.

```
#include <iostream>
#include <string>

template<typename Cat, typename Dog>
Cat func(Dog x)
{
    Cat result(x);
    result += result;
    return result;
}

int main()
{
    double a = func<double, int>(10);  // func вернёт число типа double со значениеи 20.0
    std::string b = func<std::string, const char*>("Hello"); // b будет строкой std::string
                                                             // со значением HelloHello
}
```

Если у шаблона есть несколько параметров, то можно задать только первые несколько параметров, а остальные компилятор попытается вывести. Но не во всех ситуациях вывести тип параметра возможно. Например, нельзя вывести тип, если он используется для только для возвращаемого значения.

```
double a1 = func<double, int>(10);  // Задаём значения для Cat и для Dog

double a2 = func<double>(10);       // Задаём значения для Cat.
                                    // Компилятор догадается, что Dog должен быть int

double a3 = func(10);               // Компилятор не может знать, чему должно быть равно Cat
                                    // Будет ошибка компиляции.
```

Шаблонные аргументы по умолчанию

Также как и для обычных параметров функции, шаблонным параметрам можно дать значение по умолчанию.

```
#include <iostream>

template<typename Cat, typename Dog = char>
void func()
{
    std::cout << sizeof(Cat) << " " << sizeof(Dog) << std::endl;
}

int main()
{
    func<float, double>();  // Напечатает 4 8
    func<float>();          // Напечатает 4 1, тип Dog будет выбран по умолчанию как char
}
```

Значение по умолчанию берётся только тогда, когда тип не был указан при вызове и вывести тип по передаваемому аргументу функции нельзя.

```
#include <iostream>

template<typename Cat, typename Dog = short>
void func(Cat c, Dog d)
{
    std::cout << sizeof(c) << " " << sizeof(d) << std::endl;
}

int main()
{
    // В предположении, что размеры char, short, int равны 1, 2, 4 байт соответственно
    int a = 1;
    int b = 1;
    func<char, char>(a, b);  // Напечатает 1 1
    func<char>(a, b);        // Напечатает 1 4
    func(a, b);              // Напечатает 4 4
}
```

Шаблоны и перегрузка

Шаблоны можно использовать вместе с перегрузкой функций. В случае наличия выбора между шаблоном и точным соответствием перегрузки предпочтение отдается перегрузке

```
#include <iostream>

template<typename T>
void func(T x)
{
    std::cout << "Template" << std::endl;
}

void func(int x)
{
    std::cout << "Int" << std::endl;
}

int main()
{
    func(10);     // Напечатает Int
    func(1.5);    // Напечатает Template
    func("Cat");  // Напечатает Template
}
```

Шаблоны с нетиповыми параметрами

Шаблонными аргументами могут быть не только типы но и объекты некоторых типов, примерно также как и обычные аргументы функций. Только в отличии от значений обычных аргументов, значения шаблонных аргументов должны быть известны на этапе компиляции.

```
#include <iostream>

template<int A, int B>
int sum()
{
    return A + B;
}

int main()
{
    std::cout << sum<10, 20>() << std::endl;  // Напечатает 30

    int a, b;
    std::cin >> a >> b;
    int c = sum<a, b>();  // Ошибка. Аргументы шаблона должны быть известны на этапе компиляции
}
```

Такие шаблонные параметры раскрываются также, как и типовые параметры шаблонов. То есть, когда компилятор видит в коде `sum<10, 20>()` он по шаблонной функции `sum` создаёт обычную функцию `sum<10, 20>` заменяя `A` на `10`, а `B` на `20`. Значения аргументов шаблонной функции должны быть известны на этапе компиляции, так как генерация функции из шаблона и подстановка всех значений происходит на этапе компиляции.

Далеко не каждый тип может быть типом шаблонного параметра. Например, `std::string` не может быть типом шаблонного параметра, то есть нельзя написать так:

```
template<std::string A>
A func()
{
    return A;
}
```

В качестве параметров можно использовать целые числа и числа с плавающей точкой и некоторые другие типы. Точные правила какие типы можно использовать для шаблонных параметров, а какие нет достаточно сложны и не будут тут приводиться.

Вообще, нетиповые параметры используются реже, чем типовые(`typename`) параметры. Когда же такие параметры используются, то обычно используются целочисленные типы, такие как `int` или `size_t`.

Два этапа компиляции шаблона

Рассмотрим следующую программу, содержащую шаблон `func`:

```
template<typename T>
int func()
{
    T x;
    x.abracadabra();
}

int main() {}
```

Функция `func` создаёт объект типа `T` и вызывает у этого объекта метод `abracadabra`. При этом ни одного класса с методом `abracadabra` у нас нет. Скомпилируется ли эта программа?

Вывод типа в шаблонных функциях по аргументу

В `seminar4_templates/theory/code/template_type_deduction/get_type_name.hpp` лежит функция `getTypeName`, которая принимает на вход (через шаблонный параметр) тип и возвращает строковое представление данного типа. Эта функция будет очень полезна при анализировании того, какие типы выводятся в шаблонах. Помимо этой функции будет использоваться оператор `decltype`, который просто возвращает тип выражения.

Использовать функцию `getTypeName` можно следующим образом:

```
#include <iostream>
#include "get_type_name.hpp"
int main()
{
    int a = 10;

    std::cout << getTypeName<int>() << std::endl;
    std::cout << getTypeName<decltype(a)>() << std::endl;
}
```

Шаблонная функция принимает по значению

Рассмотрим случаи автоматического вывода шаблонного типа, при котором получаемый тип будет не соответствовать типу аргумента функции. Для начала, случай когда шаблонная функция принимает по значению. В этом случае, итоговый тип будет являться decay-версией типа аргумента. Всего есть три правила для распада типа:

1. Ссылки и `const`-квалификаторы отбрасываются.
2. Массивы в стиле C преобразуются в указатели (array to pointer decay).
3. Функции преобразуются в указатели на функции.

Рассмотрим пример, иллюстрирующий такое поведение:

```
#include <iostream>
#include "get_type_name.hpp"
template<typename T>
void func(T x)
{
    std::cout << getTypeName<T>() << std::endl;
}

int main()
{
    int a = 10;
    func(a);                // int, так как тип выведется как int
    func<int&>(a);          // int&, если мы указываем тип, то он не выводится
    func<const int&>(a);    // const int&, если мы указываем тип, то он не выводится

    int& b = a;
    func(b);                // int, так как амперсанд отбрасывается
    const int& c = a;
    func(c);                // int, так как const и амперсанд отбрасывается

    int d[3] = {10, 20, 30};
    func(d);                // int*, так как массив разлагается в указатель
    func("Hello");
}
```

Шаблонная функция принимает по ссылке или const ссылке

Если же шаблонная функция принимает по ссылке (обычной или константной), то никакого разложения происходить не будет. Тип T будет подбираться таким образом, чтобы функция смогла принять передаваемый аргумент.

```
#include <iostream>
#include "get_type_name.hpp"

template<typename T>
void func(T& x)
{
    std::cout << getTypeName<decltype(x)>() << std::endl;
}

int main()
{
    int a = 10;
    func(a);                // int&
    func<int&>(a);          // int&
    func<const int&>(a);    // int&

    const int& b = a;       // const int&
    func(b);

    int arr[3] = {10, 20, 30};
    func(arr);              // int(&)[3] (ссылка на массив)
}
```

Ключевое слово auto

Ключевое слово auto используется для автоматического вывода типа.

```
#include <string>
int main()
{
    auto a = 123;   // a будет иметь тип int
    auto b = 4.1;   // b будет иметь тип double
    auto c = 4.1f;  // c будет иметь тип float

    auto s1 = "Hello";               // s1 будет иметь тип const char*
    auto s2 = std::string("Hello");  // s2 будет иметь тип std::string
}
```

Вывод типа при использовании auto

Правила вывода типа при использовании ключевого слова auto такие же, что и при выводе типов шаблонных параметров.

```
#include <iostream>
#include "get_type_name.hpp"

void f(int x) {}

int main()
{
    auto a = 10;
    std::cout << getTypeName<decltype(a)>() << std::endl;

    int& r = a;
    auto b = r;
    std::cout << getTypeName<decltype(b)>() << std::endl;

    const int& cr = a;
    auto c = cr;
    std::cout << getTypeName<decltype(c)>() << std::endl;


    int arr[3] = {10, 20, 30};
    auto d = arr;
    std::cout << getTypeName<decltype(d)>() << std::endl;


    std::cout << "---\n";
    std::cout << getTypeName<decltype(f)>() << std::endl;
    auto e = f;
    std::cout << getTypeName<decltype(e)>() << std::endl;
}
```

auto и возвращаемое значение функции

trailing return type. хвостовой тип возвращаемого значения.

```
auto factorial(int n)
```

```
{
    if (n > 0)
        return n * factorial(n - 1);
    return 1;
}
```

auto и шаблоны

Ключевое слово auto можно использовать не только при объявлении новых переменных, но и для параметров функций. То есть можно писать вот так:

```
void func(auto x)
{
    ...
}
```

В этом случае компилятор будет воспринимать эту функцию не как обычную, а как шаблонную функцию следующего вида:

```
template<typename T>
void func(T x)
{
    ...
}
```

Приведём примеры функций, которые используют auto для типа параметра:

- Функция, которая печатает размер объекта в байтах:

```
#include <iostream>

void printSize(auto x)
{
    std::cout << sizeof(x) << std::endl;
}

int main()
{
    printSize(10);
    printSize(1.5);
}
```

- Функция нахождения модуля числа:

```
#include <iostream>

auto abs(auto x)
{
    if (x < 0)
        return -x;
    return x;
}

int main()
{
    std::cout << abs<int>(-5) << std::endl;       // Напечатает 5
    std::cout << abs<double>(-1.5) << std::endl;  // Напечатает 1.5
}
```

Range-based циклы

Циклы, основанные на диапазоне, предоставляют более простой способ обхода контейнера:

```
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> v {6, 1, 7, 4};
    for (int num : v)
        std::cout << num << std::endl;
}
```

Для изменения элементов контейнера при обходе нужно использовать ссылки:

```
for (int& num : v)
    num += 1;
```

Structure binding (структурное связывание)

Для объявления и инициализации нескольких переменных через агрегатный объект можно использовать так называемое структурное связывания. В коде ниже мы объявляем переменные `a` и `b` одной строкой с помощью структурного связывания.

```
#include <iostream>
struct Point
{
    int x;
    int y;
};
int main()
{
    Point p {5, 1};
    auto [a, b] = p;

    std::cout << a << " " << b << std::endl;
}
```

Шаблоны классов

```
#include <iostream>
#include <string>
#include <cstdlib>
using std::cout, std::endl, std::size_t;

template <typename T, size_t Size>
class Array
{
private:
    T data[Size];

public:
    T& operator[](size_t id)
    {
        return data[id];
    }
};

template <typename T, size_t Size>
std::ostream& operator<<(std::ostream& out, Array<T, Size> array)
{
    out << "[";
    for (size_t i = 0; i < Size - 1; ++i)
        out << array[i] << ", ";

    if (Size != 0)
        out << array[Size - 1];

    out << "]";
    return out;
}

int main()
{
    Array<int, 10> a;
    for (int i = 0; i < 10; ++i)
    {
        a[i] = rand() % 100;
    }
    cout << a << endl;

    Array<std::string, 5> b;
    b[0] = "Cat";
    b[1] = "Dog";
    b[2] = "Zebra";
    b[3] = "Elephant";
    b[4] = "Whale";
    cout << b << endl;
}
```

Зависимые имена

Можно создавать новые имена в классах:

```
#include <iostream>
struct Alice
{
    typedef int cat;
    using dog = int;
};

int main()
{
    std::cout << sizeof(Alice::cat) << std::endl;
    std::cout << sizeof(Alice::dog) << std::endl;
}
```

Но есть неоднозначность между новым именем и статической переменной.

```
#include <iostream>

struct Alice
{
    using cat = int;
};

template<typename T>
struct Bob
{
    void func()
    {
        T::cat* x;
    }
};

int main()
{
    Bob<Alice> b;
    b.func();
}
```

Зависимый шаблон

```
#include <iostream>
struct Alice
{
    template<int A>
    void get()
    {
        std::cout << "get method" << std::endl;
    }
};

struct Bob
{
```

```
    int get {10};
};

template <typename T>
void func(T x)
{
    //x.get<0>(); // Неправильно, так как get будет считаться полем
    x.template get<0>(); // Правильно, теперь get это шаблонный метод
}

int main()
{
    Alice a;
    func(a);
    // Bob b;
    // func(b);
}
```

Специализация

```
template<typename T>
struct Cat {
    static void f() { std::cout << "primary\n"; }
};

template<>
struct Cat<int> {
    static void f() { std::cout << "specialized for int\n"; }
};
```

Шаблоны новых имён типов

```
#include <vector>

template<typename T>
using MyType = std::vector<T>;

int main()
{
    MyType a = {10, 20, 30, 40, 50};
}
```

Примеры шаблонов в стандартной библиотеке C++

Класс `std::vector`

Класс `std::array`

Класс `std::pair`

Класс `std::optional`

Класс `std::basic_string`

`std::basic_string` — это базовый шаблон, на котором построены все обычные строки в C++:

- `std::string = std::basic_string<char>`
- `std::wstring = std::basic_string<wchar_t>`
- и т. д.

Вариативные шаблоны

Вариативные шаблоны (variadic templates) – это механизм, позволяющий шаблонам принимать произвольное число параметров.

```
template<typename... Types>
void func(Types... args) {}
```

Использование пака типов и пака параметров:

```
template<typename... Types>
void func(Types... args)
{
    // Types - особая сущность, пак типов
    // args - особая сущность, пак аргументов

    // Что можно сделать с паками:

    f<Types...>() - развернуть пак типов
    f(args...)    - развернуть тип аргументов

    sizeof...(args)  - количество аргументов в паке
    sizeof...(Types) - количество аргументов в паке

    // fold expressions
}
```

Рекурсивное раскрытие параметров

```
void print() {}

template<typename T, typename... Ts>
void print(T first, Ts... rest)
{
    std::cout << first << "\n";
    print(rest...);
}
```

Fold expressions

```
template<typename... Ts>
auto sum(Ts... ts)
{
    return (ts + ...);
}


template<typename... Ts>
auto sumsq(Ts... ts)
{
    return ((ts * ts) + ...);
}


template<typename T>
void print_one(const T& t) {
    std::cout << t << "\n";
}

template<typename... Ts>
void print_all(const Ts&... ts) {
    (print_one(ts), ...);
}
```