C/C++: Лекция 5

Воробьев Д.В

02.10.2020

Шаблоны

Шаблонный класс

```
template<typename T>
class Foo {
    private:
        T a;
};

int main() {
    Foo<int> foo;
    return 0;
}
```

Шаблонная функция

```
template<typename T>
T Max(T a, T b) {
    return a < b ? b : a;
}

int main() {
    std::cout << Max(1, 2);

    return 0;
}
```

Различий между class и typename нет

```
template<typename T>
void foo(T a) {
    std::cout << a;
}

int main() {
    foo(1);
    return 0;
}
```

```
template<class T>
void foo(T a) {
    std::cout << a;
}

int main() {
    foo(1);
    return 0;
}
```

Параметр

Можно задавать параметры по умолчанию

```
template<class T=int>
struct array {
    T a[3];
};

int main() {
    array x;
    return 0;
}
```

Параметр

Можно задавать числа параметрами

```
template<class T=int, std::size_t size=3>
struct array {
    T a[size];
};

int main() {
    array x;
    return 0;
}
```

Параметр

Шаблонный шаблонный параметр

Во вложенном template<typename T1, typename T2> указывается число шаблонных параметров, требуемых для шаблонного шаблонного параметра

```
template<typename K, template<typename T1, typename T2> typename C>
class Map {
    C<K, T1> x;
    C<T1, T2> y;
};
```

Инстанцирование шаблона

Создание экземпляра шаблона с фиксированным типом

Инстанцирование шаблона

```
template<typename T>
T foo(T a) {
    std::cout << typeid(a).name();
    return a;
}

int main() {
    double x = 10;
    std::cout << foo(x);
    int y = 20;
    std::cout << foo(y);
    return 0;
}
```

Инстанцирование происходит единожды

```
template<typename T>
void foo(T a) {
    static int count = 0;
    count++;
    std::cout << count;
}

int main() {
    // 1
    foo(1.0);
    // 2
    foo(4.0);
    return 0;
}
```

Полная специализация

Имплементация шаблонной entity с указание всех типов

```
template<typename T>
void foo(T a) {
    std::cout << "templ";
}

template<>
void foo(int a) {
    std::cout << "spec";
}

int main() {
    // templ
    foo(1.0);
    // spec
    foo(4);
    return 0;
}
```

Специализация

Специализируемые функции должны иметь ровно тот же parameter list

```
template<typename T>
void foo(T a) {
    std::cout << typeid(a).name();
}

template<>
// CE: это не специализация foo
// т.к. другой parameter list
void foo(int a, int b) {
    std::cout << a;
}

int main() {
    return 0;
}
```

Частичная специализация

Имплементация шаблонного класса с указанием части типов

```
template<typename T, std::size_t size>
 class array {
   private:
       T a[size];
 };

 template<std::size_t size>
 class array<int, size> {
   private:
       int a[size];
 };
```

Частичная специализация

Для функций отсутствует (смотри внимательно на <> после identifier)

Перегрузка

```
template<typename T1, typename T2>
void foo(T1 a, T2 b) {}

// перегрузка
template<typename T1>
void foo(T1 a, T1 b) {}
```

Частич. специализации нет

```
template<typename T1, typename T2>
void foo(T1 a, T2 b) {}

// CE
template<typename T1>
void foo<T1, T1>(T1 a, T1 b) {}
```

Статический полиморфизм

Полиморфизм - это общее имя, единый интерфейс и множество реализаций данного интерфейса.

Статический т.к. разрешение сценария "Какую реализацию вызвать под данным общим именем?" происходит на этапе компиляции

К нему относятся:

- шаблоны
- перегрузка функций

Статический полиморфизм: перегрузка

- Общее имя: foo
- Общий интерфейс: foo
- Реализации: 4 перегрузки
- Разрешение вызываемой версии: compile time

Статический полиморфизм: перегрузка

```
void foo(double a, double b) { std::cout << 1; }

void foo(int a, double b) { std::cout << 2; }

void foo(double a, int b) { std::cout << 3; }

void foo(int a, int b) { std::cout << 4; }

int main() {
    foo(1, 1.0);
}
```

Статический полиморфизм: шаблоны

```
struct Dog {
    void Age() { return 10; }
};

struct Person {
    void Age() { return 45; }
};

template<typename T>
void foo() {
    T obj;
    std::cout << obj.Age();
}

int main() {
    foo(Dog());
    foo(Person());
}
```

Статический полиморфизм: шаблоны

- Общее имя: T
- Общий интерфейс: Age
- Реализации: реализация Age в Dog, Person
- Разрешение вызываемой версии: compile time

CRTP

Curiously recurring template pattern

Реализуется 3 составляющими:

- наследованием от базы с шаблонным параметром равным дочернему типу
- явным приведением к дочернему типу в static_cast
- вызовом реализации дочернего из интерфейса

CRTP

```
template <class T>
struct Base {
    void interface() {
        static_cast<T*>(this)->implementation();
    }
};
struct Derived : Base<Derived> {
     void implementation() {
        std::cout << "Derived";
     }
};
template<typename T>
void foo(Base<T>& b) {
    b.interface();
}

int main() {
    Derived d;
    foo(d);
}
```

template typedef, using

До C++11 хотели делать так: вводить шаблонный алиас с typedef, но возможности не было

```
template<typename CharT>
typedef std::basic_string<CharT, std::char_traits<CharT>> mystring;

int main() {
    mystring<char> str;
    return 0;
}
```

template typedef, using

В C++11 возможность добавили с синтаксисом using

```
template<typename CharT>
using mystring = std::basic_string<CharT, std::char_traits<CharT>>;

int main() {
    mystring<char> str;
    return 0;
}
```

typename для доступа к полю тип

Без typename

```
template<typename T>
class Vec {
    typedef T AliasedT;
};

template<typename T>
void foo() {
    // CE: AliasedT считается п
    Vec<T>::AliasedT x;
}
```

С typename

```
template<typename T>
class Vec {
    typedef T AliasedT;
};

template<typename T>
void foo() {
// Получаем доступ к AliasedT
    typename Vec<T>::AliasedT x;
}
```

Реализация некоторых type_traits

Позволяет с типа снять ссылку

```
template<class T> struct remove_reference {
    typedef T type;
}
template<class T> struct remove_reference<T&> {
    typedef T type;
}
```

Реализация некоторых type_traits

Позволяет с типа снять const

```
template<class T> struct remove_const {
    typedef T type;
}
template<class T> struct remove_const<const T> {
    typedef T type;
}
```

Правила вывода типов для шаблонов

Параметр (P) есть non reference type

A без const => T = A

```
template<typename T>
void foo(T arg) {}

int main() {
    int x = 3;
    // A = int, P=T => T = int
    foo(x);
}
```

A с const => T = A

```
template<typename T>
void foo(T arg) {}

int main() {
    const int x = 3;
    // A = const int, P=T => T = int
    foo(x);
}
```

Правила вывода типов для шаблонов

Параметр (P) reference type

T = тип, на который ссылается P

```
template<typename T>
void foo(T& arg) {}

int main() {
    int x = 3;
    // A = int, P = T& => T = int
    foo(x);
}
```