C/C++: Lecture 5

Vorobev D.V

02.10.2020

Templates

Class template

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

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

Function template

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

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

    return 0;
}
```

There are no differences between class and typename keywords.

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

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

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

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

Template parameters

Template parameter can be one of three:

1) Type template parameter
2) Non-type template parameter
3) Template template parameter

Type and non-type template parameters

```
// T is a type parameter
// size is a non-type parameter
template<typename T=int, size_t size=3>
struct array {
    T a[size];
};

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

Template template parameter

In the angle brackets template<typename , typename > we specify a number of template parameters used by template template parameter.

```
template<typename K, typename T1, template<typename, typename> typename C>
class Map {
    C<K, T1> x;
    C<T1, 12> y;
};
```

Template parameter

We can specify default template arguments for some template parameters in the parameter list

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

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

Class template instantiation

Class template instantiation

It is a generation of a code by a compiler for the provided set of template arguments.

C++

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

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

x86-64 clang 10.0.0

```
main:
    ...
    call    void foo<double>(double)
    ...
    call    void foo<int>(int)
    ...
    ret

// an instance for foo<double>
void foo<double>(double):
    ...
    ret

// an instance for foo<int>
void foo<int>(int):
    ...
    ret
```

For the provided set of template arguments a class template instantiation occurs only once.

C++

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

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

x86-64 clang 10.0.0

```
main:
    push    rbp
    ...
    call    void foo<double>(double)
    ...
    call    void foo<double>(double)
    ...
    pop     rbp
    ret

// A single instance of the foo for double
void foo<double>(double):
    push    rbp
    ...
    pop     rbp
    ret
```

Template specialization

Explicit specialization

A customization of the template code for a given set (full set) of template arguments

```cpp
template<typename T>
void foo(T a) {
    std::cout << "template";
}
template<>
void foo(int a) {
    std::cout << "explicit specialization";
}

int main() {
    // template
    foo(1.0);
    // explicit specialization
    foo(4);
    return 0;
}
```

The parameter list of a specialization of a primary template function must match the parameter list of the primary template

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

template<>
// CE: different parameter lists
void foo(int a, int b) {}

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

Partial specialization

A customization of the template code for a given **subset** (not full) of template arguments

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

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

Note

Functions can not be partially specialized

Function overloading

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

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

There is no partial spec

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

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

Static polymorphism

Definition

Polymorphism it is a single name, a single interface and multiple implementations of these interface.

Note

The word **static** in the term "static polymorphism" means that the resolving the implementation used occurs at compile time.

Note

Static polymorphism includes **templates** and **function overloading**.

Static polymorphism: function overloading

```
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);
}
```

Static polymorphism: function overloading

- Single name: foo
- Single interface: foo
- Implementation: 4 overloaded functions

Static polymorphism: function templates

```
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());
}
```

Static polymorphism: function templates

- Single name: T
- Single interface: Age
- Implementations: Age function implementation inside Person and Dog classes

CRTP=Curiously recurring template pattern

It is a technique of emulating virtual functions

It has 3 key points:

- Deriving base class whose template argument is a derived class
- Explicit cast using a static_cast operator to template parameter at the base class
- Calling an implementation after the casting to the template parameter

CRTP

```
template <typename 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);
}
```

Alias templates

Since **C++11** we can use alias templates.

```
template<typename T>
struct Alloc { };

// alias template
template<typename T>
using Vec = vector<T, Alloc<T>>;

Vec<int> v;
```

typename as a disambiguator

Without typename

```
template<typename T>
struct Vec {
    using AliasedT = T;
};

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

With typename

```
template<typename T>
struct Vec {
    using AliasedT = T;
};

template<typename T>
void foo() {
    // no ambiguity
    typename Vec<T>::AliasedT x;
}
```

type_traits: examples

```
template<typename T> struct remove_reference {
    using type = T;
}
template<typename T> struct remove_reference<T&> {
    using type = T;
}
```

```
template<typename T> struct remove_const {
    using type = T;
}
template<typename T> struct remove_const<const T> {
    using type = T;
}
```

Template argument deduction

P is a non-reference type

A is a non-const => T = A

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

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

A is a const qualified => T = A

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

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

Template argument deduction

P is a reference type

A is a non-const => T = A

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

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

A is a const qualified => T = const A

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

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