



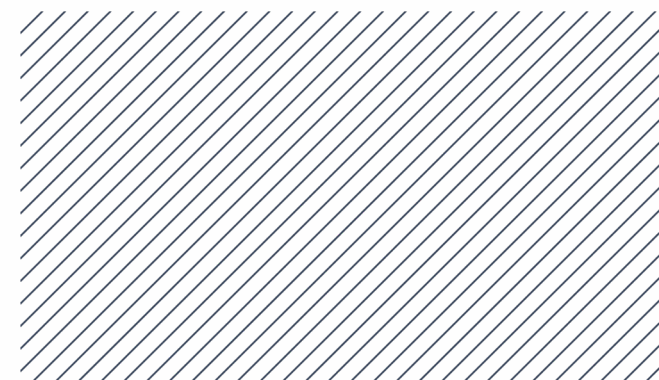
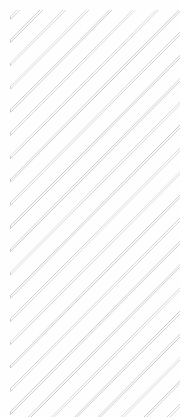
Lecture 14

Templates

Konstantin Leladze

OOP in C++

DIHT MIPT 2021



Function overloading

```
#include <iostream>
#include <cmath>

int ceil_num (int x) {
    std::cout << "first function called" << std::endl;
    return x;
}

double ceil_num (double x) {
    std::cout << "second function called" << std::endl;
    return std::ceil(x);
}

int main () {
    std::cout <<
        ceil_num(2) << std::endl <<
        ceil_num(2.5) << std::endl;
}
```

In C++ it is possible to define functions with the same names, but with different arguments and return types (this is not a compilation error!)

When a function call occurs, the most appropriate one is selected from all available overloads.

The question arises: what will happen in the case of the next expression: **f('a')**?

```
first function called
2
second function called
3
```

Function overloading: resolution rules

This is where the overload resolution rules come into play. The rules are a long and complex. They are a list of formal instructions for the compiler, so we will not analyze it exactly. Instead, I will give a general rule:

If the function has received exactly the types of arguments for which it is defined, then it is called.

Otherwise, there are various attempts to convert the original type to types defined in any overload versions.

Compilation errors occur only in two cases:

- 1) There was no suitable function at any of the stages
- 2) At some stage, several suitable functions were found

An example of an error and its fix is on the right side of the slide.

```
void f (int x) {}  
void f (char x, int y) {}  
void f (float z) {}  
  
int main () {  
    f(3.14);  
}
```

Call to 'f' is ambiguous
candidate function
candidate function

Declared in: main.cpp

void f(float z)



```
void f (int x) {}  
void f (char x, int y) {}  
void f (float z) {}  
  
int main () {  
    f(3.14f);  
}
```



Templates: motivation

```
void swap (int& first, int& second) {  
    int temp = first;  
    first = second;  
    second = temp;  
}
```

```
std::vector<double> vector_of_doubles;  
std::vector<int> vector_of_ints;  
std::vector<char> vector_of_chars;
```

We would like to be able to write universal code for different types.
(Just like in the case of the `std::vector<...>`)

Templates: definition

In C++, functions, classes, and lambda expressions can be template. Let's look at template functions and classes:

```
template <typename type>
void swap (type& first, type& second) {
    type temp = first;
    first = second;
    second = temp;
}
```

```
template <typename type>
class a_class {
    /// ...
};
```

The word **type** here defines the type of the variable for which this function will be called. For example, if the function is called with the template **type** = int, then wherever the word **type** occurs in the swap function, it will be replaced with int.

The template mechanism in C++ works entirely at compile-time, meaning that the template parameters are substituted at compile time.

```
int main () {
    int x = 3, y = 2;
    swap(x, y); // inferred: [type = int]
    swap<double>(x, y); // [type = double]
}
```

```
int main () {
    a_class<int> a;
    a_class<double> b;
}
```

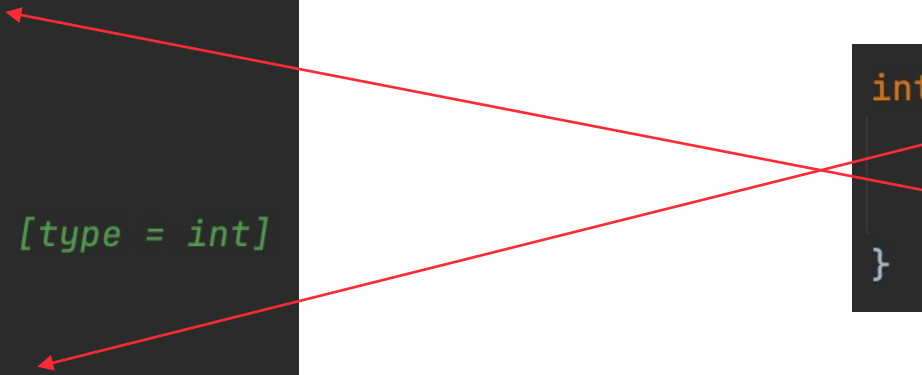
Templates: specialization

Similar to functions, templates can be specialized. Which means that you can create a version of a template function or class that will work for a specific type. Rule: the most highly specialized of suitable substitutions is chosen.

```
/// General template
template <typename type>
void f (type x) {
    /// ...
}

/// Specialization [type = int]
template <>
void f (int x) {
    /// ...
}
```

```
int main () {
    f(1); /// [type = int]
    f(1.); /// [type = double]
}
```



Templates: specialization

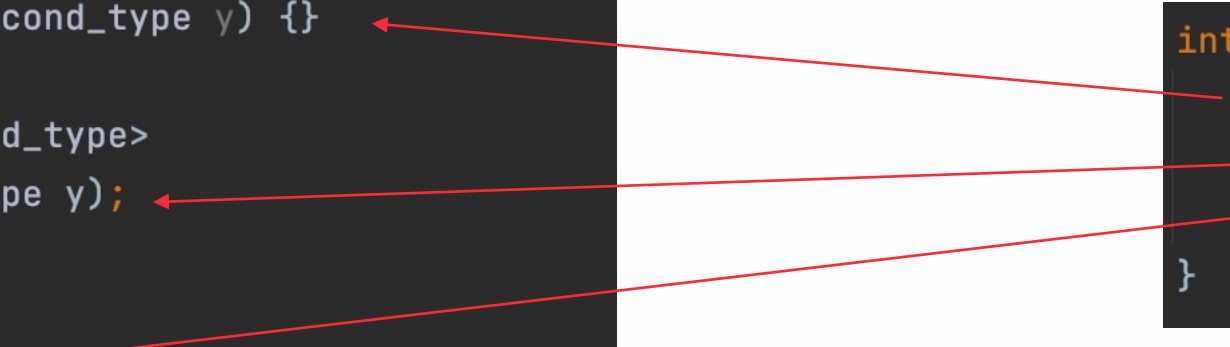
For example, this time there are two template arguments, and there is a so-called partial specialization (the second version of the g function).

```
template <typename first_type, typename second_type>
void g (first_type x, second_type y) {}

template <typename second_type>
void g (int x, second_type y);

template <>
void g (int x, int y);
```

```
int main () {
    g(2., 2.);
    g(2, 2.);
    g(2, 2);
}
```



Templates: specialization

By analogy with function overloading, specialization must be unambiguous. For example, here is the problem (it is not clear which specialization to call):

```
template <typename first_type, typename second_type>
class a_class {
    /// ...
};

template <typename second_type>
class a_class<int, second_type> {
    /// ...
};

template <typename first_type>
class a_class<first_type, int> {
    /// ...
};
```

```
int main () {
    a_class<int, int> a;
}
```


Templates: specialization

```
template <typename first_type, typename second_type>
class a_class {
    /// ...
};

template <typename second_type>
class a_class<int, second_type> {
    /// ...
};

template <typename first_type>
class a_class<first_type, int> {
    /// ...
};

template <>
class a_class<int, int> {
    /// ...
};
```

Now everything is fine: the latest version of the class is instantiated (**a_class<int, int>**)

```
int main () {
    a_class<int, int> a;
}
```



Typedef & using

C++ allows us to create our own shorthand for types. For example, if the name of some type is too long, we can create a new type that is semantically equivalent to the previous one. Moreover, we can parameterize the new type:

```
template <typename first_type, typename second_type>
class a_class {
    // ...
};

typedef a_class<int, std::vector<int>> short_name;

template <typename type>
using short_name_template = a_class<int, std::vector<type>>;

int main () {
    short_name a; /// type of the a variable is "a_class<int, std::vector<int>>"
    short_name_template<double> b; /// type of b is "a_class<int, std::vector<double>>"
    short_name_template<long> c; /// type of c is "a_class<int, std::vector<long>>"
    return 0;
}
```



Non-type template parameters

You can put some value directly into the data type at compile time

```
template <typename type, size_t size>
class array {
    // ...
};

int main () {
    array<int, 10> a;
    array<int, 20> b;
}
```



Template template parameters

```
template <typename type>
class a_class {
    /// ...
};

template <template <typename> typename template_type, typename inner_type>
void f () {
    template_type<inner_type> instance;
}

template <template <typename> typename template_type, typename inner_type>
void g (const template_type<inner_type>& t) {
    template_type<double> instance_double;
    inner_type x;
}

int main() {
    f<a_class, int>();
    g(a_class<int>());
}
```

Variadic templates

```
#include <iostream>

void print_all () {}

template <typename head, typename... tail>
void print_all (head the_head, tail... the_tail) { /// the_tail is the pack of values
    std::cout << the_head << " ";
    print_all(the_tail...);
}

int main () {
    print_all(4, 4.13, 'a'); /// args = [int, double, char]
    std::cout << std::endl;
    print_all("string", true, 4l, 1.f); /// args = [std::string, bool, long, float]
    std::cout << std::endl;

    return 0;
}
```

```
4 4.13 a
string 1 4 1
```



Functors

```
template <class type, class comparator>
void sort (type* const begin, type* const end, comparator c) {
    /// ...
    c(*begin, *end);
    /// ...
}

template <typename type>
struct less {
    bool operator() (const type& a, const type& b) {
        return a < b;
    }
};

int main () {
    int a[100];
    sort(a, a + 100, less<int>());
}
```



Lecture 14

Templates

Konstantin Leladze

OOP in C++

DIHT MIPT 2021

