

TEMPLATES - BASICS



CODERS
SCHOOL

ŁUKASZ ZIOBRÓŃ

AGENDA

1. Intro and pre-test (30")
2. Template functions (30")
3. Template classes (45")
4. `type_traits` library (45")
5. Specialization (45")
6. Partial specialization (45")
7. Overload resolution rules (30")
8. Variable templates (30")
9. Recap (20")

REPO

Repo GH `coders-school/templates`

<https://github.com/coders-school/templates/tree/cr/module1>

SOMETHING ABOUT YOU

- Have you ever wrote a template function or class?
- What do you expect from today's session?

ŁUKASZ ZIOBRÓŃ

NOT ONLY A PROGRAMMING XP

- Front-end dev, DevOps & Owner @ Coders School
- C++ and Python developer @ Nokia & Credit Suisse
- Team leader & Trainer @ Nokia
- Scrum Master @ Nokia & Credit Suisse
- Code Reviewer @ Nokia
- Web developer (HTML, PHP, CSS) @ StarCraft Area

EXPERIENCE AS A TRAINER

- C++ online course @ Coders School
- Company trainings @ Coders School
- Practical Aspects Of Software Engineering @ PWr & UWr
- Nokia Academy @ Nokia





PUBLIC SPEAKING EXPERIENCE

- code::dive conference
- code::dive community
- Academic Championships in Team Programming
- Coders School YouTube channel

HOBBIES

- StarCraft Brood War & StarCraft II
- Motorcycles
- Photography
- Archery
- Andragogy

CONTRACT

-  Vegas rule
-  Discussion, not a lecture
-  Additional breaks on demand
-  Be on time after breaks

PRE-TEST 1/3 🤯

WHICH OF THE FOLLOWING USAGES OF A TEMPLATE FUNCTION ARE CORRECT?

```
template <typename T, typename U>  
U fun(T arg1, U arg2);
```

1. `fun(4, 5)`
2. `fun(4, 5.5)`
3. `fun<int, int>(4, 5)`
4. `fun<int, double>(4, 5.5)`
5. `fun<int, int>(4, 5.5)`

PRE-TEST 2/3 🤯

WHICH OF THE FOLLOWING USAGES OF A TEMPLATE FUNCTION ARE CORRECT?

```
template <typename T, typename U, typename V>  
V fun(T arg1, U arg2);
```

1. fun(4, 5.5)
2. fun(4, 5.5, int)
3. fun(4, 5.5, 'c')
4. fun<int, double, int>(4, 5.5)
5. fun<int, double, char>(4, 5.5)
6. fun<int, double, char>(4, 5.5, 'c')
7. fun<char>(4, 5.5)
8. fun<V = char>(4, 5.5)

PRE-TEST 3/3 🤯

WHICH OF THE FOLLOWING USAGES OF A TEMPLATE FUNCTION ARE CORRECT?

```
template<typename To, typename From>  
To convert(From f);
```

1. `auto a = convert(3.14);`
2. `auto b = convert<int>(3.14);`
3. `auto c = convert<int, double>(3.14);`
4. `int(*p1)(float) = convert;`
5. `int(*p2)(float) = convert<int, float>;`

INTRO



CODERS
SCHOOL

RATIONALE

Templates are used to avoid code duplication.

Later it was discovered that C++ templates are a Turing-complete language. It means that the templates itself are a programming language. It is often called **C++ template meta-programming**.

All templates are evaluated during the compilation phase.

In templates types are something like variables.

OBSERVATION

- A lot of C/C++ developers are afraid of C++ templates and thus don't know them, their features and usage
- They blame templates for:
 - magically generating the code
 - need of putting the most of them in the header files
 - code bloat
 - hard to analyze compiler errors
 - problems with debugging the code
- The same developers are also amazingly comfortable with using preprocessor macros
- Preprocessor macros:
 - are generating the code
 - need to be put mostly in the header files
 - generate the code in place causing code duplications all over the binary
 - generate inscrutable compiler errors in places where you don't expect them
 - are really hard to debug

MACROS VS TEMPLATES

Both are code generation tools.

MACROS

```
#define min(i, j)  (((i) < (j)) ? (i) : (j))
```

Simple text substitution mechanism done during preprocessing stage (before compilation)

TEMPLATES

```
template<class T>  
T min(T i, T j) { return (i < j) ? i : j; }
```

Functional Turing complete feature-rich language that is executed at compile time and is integrated into the C++ type system

PROBLEMS WITH MACROS

- There is no way for the compiler to verify that the macro parameters are of compatible types
- The macro is expanded without any special type checking
- Macro parameters are evaluated many times (in every place they are put to the code)
- Compiler error messages will refer to the expanded macro, rather than the macro definition itself
- Problems with single-stepping macro code during debugging

TEMPLATE TYPES

In C++03 we have had

- template functions
- template classes

From C++11 we can also have

- template variables
- template aliases

And from C++20 we can have

- template lambdas

TEMPLATE FUNCTIONS



CODERS
SCHOOL

EXAMPLES

Let's assume that we have a function below:

```
int add(int first, int second) {  
    return first + second;  
}
```

If we want to have a function that takes doubles as well, we need to write:

```
double add(double first, double second) {  
    return first + second;  
}
```

And if we want a function that can take complex or any other numbers we would need to write:

```
std::complex<int> add(std::complex<int> first, std::complex<int> second) {  
    return first + second;  
}
```

You can clearly see that we have a code duplication here.

AVOIDING CODE DUPLICATION

Instead of writing so many functions we can have only one - template function:

```
template <typename Type>
Type add(Type first, Type second) {
    return first + second;
}
```

Instead of `Type`, you can have any name you wish. Typically you will see just `T` as a typename, but it is better to have a longer name than only one character, especially, when there is more than only one template parameter. Now, you can use this function like this:

```
auto resultI = add<int>(4, 5); // resultI type is int
auto resultD = add<double>(4.0, 5.0); // resultD type is double
auto resultC = add<std::complex<int>>({1, 2}, {2, 3}); // resultC type is std::complex<int>
```

You can play with the code [here](#)

FUNCTION TEMPLATE TYPE DEDUCTION

There is a function template types deduction in C++. It means that you can skip part with angle braces <> and write previous example like this:

```
auto resultI = add(4, 5); // resultI type is int
auto resultD = add(4.0, 5.0); // resultD type is double
auto resultC = add({1, 2}, {2, 3}); // error, does not compile
```

resultC will not compile, because in this case compiler will not know what is the type of {1, 2} or {2, 3}. `std::initializer_list` can never be a result of parameter type deduction in templates. In this case we have to type it explicitly:

```
auto resultC = add(std::complex<int>{1, 2}, std::complex<int>{2, 3});
```

or

```
auto resultC = add<std::complex<int>>({1, 2}, {2, 3});
```

EXERCISE

Write a function that creates `std::complex` number from two provided numbers. If the types of numbers are different, it should create `std::complex` of the first parameter. Usage:

```
std::complex<int> a = makeComplex(4, 5);           // both ints
std::complex<double> b = makeComplex(3.0, 2.0);    // both doubles
std::complex<int> c = makeComplex(1, 5.0);         // int, double -> takes int
```

MULTIPLE TEMPLATE PARAMETERS

The compiler itself deduce which template function parameters should be used. However, if you write the code like this:

```
auto resultC = add(4, 5.0); // error: int + double
```

We will have a compilation error. The compiler will not deduce parameter, because our template function takes only one type, and both parameters have to be of the same type. We can fix this by adding a new version of the template of add function.

```
template <typename TypeA, typename TypeB>  
TypeA add(TypeA first, TypeB second) {  
    return first + second;  
}
```

Now the code should work:

```
auto resultC = add(4, 5.0); // resultC type is int
```

The output type is the same as the first argument type because it was defined in the template function above as TypeA.

typeid

Generally, you can freely use template types inside functions. For example, you can create new variables of provided types:

```
#include <typeinfo>

template <class T>
void showType() {
    T value;
    std::cout << "Type: " << typeid(value).name() << std::endl;
}
```

You can use `typeid().name()` to print variable type. You need to include the `typeinfo` header for this. The output is implementation-defined.

You can also notice, that instead of the `typename` keyword, you can also use the `class` keyword. They are interchangeable.

```
template <typename T> == template <class T> != template <struct T>
```

NO MATCHING FUNCTION

In previous case if you want to use `showType()` function without providing explicit templates, the code will not compile:

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

```
prog.cpp: In function 'int main()':  
prog.cpp:15:12: error: no matching function for call to showType()  
    showType();  
        ^  
prog.cpp:7:6: note: candidate: template<class T> void showType()  
    void showType()  
        ^~~~~~  
prog.cpp:7:6: note:   template argument deduction/substitution failed:  
prog.cpp:15:12: note:   couldn't deduce template parameter 'T'  
    showType();
```

TEMPLATE FUNCTION PARAMETER TYPE DEDUCTION

The compiler cannot deduce parameters, because the functions do not take any parameters. You need to provide the type explicitly:

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

or

```
int main() {  
    doNothing<std::vector<char>>>();  
    return 0;  
}
```

You can also play with the code [here](#)

STL EXAMPLE

```
template<class InputIt, class UnaryPredicate>
InputIt find_if(InputIt first, InputIt last, UnaryPredicate p)
{
    for(; first != last; ++first) {
        if(p(*first)) {
            return first;
        }
    }
    return last;
}
```

```
std::vector<std::pair<int, int>> v{{-3, 1}, {2, 3}, {4, -5}};
auto it = std::find_if(begin(v), end(v), [](auto& e){ return e.first == 2; });
if(it != std::end(v)) {
    /* ... */
}
```

TEMPLATE CLASSES



CODERS
SCHOOL

TEMPLATE CLASS EXAMPLE

Template classes are as well used to avoid code duplication, as to create so-called meta-programs within them. Here is an example of a simple template class and its usage:

```
#include <iostream>

template <typename T, typename U>
class SomeClass {
public:
    T getValue() { return value; }
private:
    T value = {};
    U* ptr = nullptr;
};

int main() {
    SomeClass<int, void> sc;
    std::cout << sc.getValue() << std::endl;
    return 0;
}
```

TEMPLATE CLASSES IN STL

Template classes are heavily used in STL. For example `std::vector`, `std::list` and other containers are template classes and if you want to use them, you do it like here:

```
std::vector<int> v = {1, 2, 3};  
std::list<char> l{'c', 'd', 'b'};
```

TEMPLATE TYPE DEDUCTION FOR CLASSES (C++17)

Template types can also be automatically deduced by the compiler thanks to... template functions.

The compiler uses class constructors to achieve that.

From C++17 you can write a code like this:

```
std::vector v = {1, 2, 3}; // std::vector<int> is deduced
std::list l{'c', 'd', 'b'}; // std::list<char> is deduced
```

That's not gonna work:

```
// std::vector v1;           // compilation error, vector of what?
// std::vector v2(10)        // compilation error, 10 elements of what?
// clang error: no viable constructor or deduction guide for deduction of
//   template arguments of 'vector'
```

EXERCISE - VectorMap

Write a template class `VectorMap` that represents an over-engineered `std::map`.

Inside, it should hold 2 `std::vectors` of the same size, each with different types. The first vector should hold keys, the other one values.

Elements at the same position in both vectors should create a pair like 1 and 'c' below.

```
VectorMap<int, char> map;  
map.insert(1, 'c');  
map[1] = 'e';           // replaces value under 1  
std::cout << map[1];    // prints 'e'  
map.at(2);              // throw std::out_of_range
```

Implement the mentioned above `insert()`, `operator[]`, `at()` methods.

Do not bother about duplicated keys for now. You can also try to implement additional methods from the `std::map` interface 😊

Use [cppreference](#).

<type_traits>



CODERS
SCHOOL

CONSTRAINTS

`<type_traits>` library is used to examine type properties. It is usually used in templates to gain some knowledge about provided types.

We can for example constrain our templates to work only with some specific types. The common practice is using it together with `static_assert`.

```
#include <type_traits>

template <typename E>
class Choice {
    static_assert(std::is_enum<E>::value, "You need to provide an enum");

    E choice;
public:
    Choice(E arg) { /* ... */ }
};
```


`<type_traits>` ON CPPREFERENCE

ADVANCED CONSTRAINTS

- SFINAE
- `constexpr if`
- named requirements (C++20)
- `concept` (C++20)

```
template <class T>
concept copy_constructible =
    std::move_constructible<T> &&
    std::constructible_from<T, T&> && std::convertible_to<T&, T> &&
    std::constructible_from<T, const T&> && std::convertible_to<const T&, T> &&
    std::constructible_from<T, const T> && std::convertible_to<const T, T>;
```

EXERCISE - `static_assert`

Add a constraint to our `VectorMap`.

Do not allow to create an object when `ValueType` does not have a default constructor.

Use `static_assert` and proper trait from `<type_traits>` library.

Check if it works properly.

EXERCISE - `isIntKey()`

Write a function `isIntKey()` in `VectorMap`. It should return `true` when the `KeyType` is `int` and `false` otherwise.

Check the `<type_traits>` library for some inspiration 😊

SPECIALIZATION



CODERS
SCHOOL

FUNCTION SPECIALIZATION

If we want to have the same function name, but we want our code to behave differently for some types, we can create a specialization.

```
//generic function
template <typename T>
void print(T arg) {
    std::cout << arg << '\n';
}
```

```
// specialization for `T = double`
template <>
void print<double>(double arg) {
    std::cout << std::setprecision(10) << arg << '\n';
}
```

```
// better: overload
void print(double arg) {
    std::cout << std::setprecision(10) << arg << '\n';
}
```

Tip: do not use function specializations. Always prefer function overloads.

Template function specializations do not take part in overload resolution. Only the exact type match is considered.

Above specialization does not work for `float`. Overload does.

CLASS SPECIALIZATION

A class can have not only different behaviour (different methods implementations) but also different layouts. You can have completely different fields and/or their values.

SPECIALIZATION EXAMPLE #1 - METHODS

```
#include <iostream>

template<typename T>    // primary template
struct is_int {
    bool get() const { return false; }
};

template<>    // explicit specialization for T = int
struct is_int<int> {
    bool get() const { return true; }
};

int main() {
    is_int<char> iic;
    is_int<int> iii;
    std::cout << iic.get() << '\n';    // prints 0 (false)
    std::cout << iii.get() << '\n';    // prints 1 (true)
    return 0;
}
```

SPECIALIZATION EXAMPLE #2 - FIELD VALUES

```
#include <iostream>

template<typename T>    // primary template
struct is_int {
    static constexpr bool value = false;
};

template<>    // explicit specialization for T = int
struct is_int<int> {
    static constexpr bool value = true;
};

int main() {
    std::cout << is_int<char>::value << '\n';    // prints 0 (false)
    std::cout << is_int<int>::value << '\n';    // prints 1 (true)
    return 0;
}
```

You can play with the code [here](#)

SPECIALIZATION EXAMPLE #3 - <TYPE_TRAITS>

To achieve the last behavior, we can use `std::false_type` and `std::true_type`. The below code is equivalent to the one from the previous example.

```
#include <iostream>
using namespace std;

template<typename T>    // primary template
struct is_int : std::false_type
{};

template<>    // explicit specialization for T = int
struct is_int<int> : std::true_type
{};

int main() {
    std::cout << is_int<char>::value << std::endl;    // prints 0 (false)
    std::cout << is_int<int>::value << std::endl;    // prints 1 (true)
    return 0;
}
```

The interactive version of this code is [here](#)

EXERCISE - `is_int_key`

In `VectorMap` write a class constant `is_int_key` that holds a boolean value. It should be `true` when the key is `int` and `false` otherwise.

Generally, it should do the same job as the `isIntKey()` method, but we want to have it available even without having an object.

Take a look in the `<type_traits>` library for that. It should be useful 😊

PARTIAL SPECIALIZATION



CODERS
SCHOOL

TEMPLATE FUNCTION PARTIAL SPECIALIZATION

In C++ we cannot partially specialize functions.

TEMPLATE CLASS PARTIAL SPECIALIZATION

In C++ we can specialize classes partially. It means that we need to leave at least one template parameter.

```
// primary template
template <typename T1, typename T2, typename T3>
class A {};

// partial specialization with T1 = int
template <typename T2, typename T3>
class A<int, T2, T3> {};

// partial specialization with T1 = int and T2 = double
template <typename T3>
class A<int, double, T3> {};

// partial specialization with T1 = int and T3 = char
template <typename T2>
class A<int, T2, char> {};

// full specialization with T1 = T2 = double and T3 = int
template <>
class A<double, double, int> {};
```

ADVANCED PARTIAL SPECIALIZATION

```
template <typename T1, typename T2, typename I>
class A {}; // primary template

template<typename T, typename I>
class A<T, T*, I> {}; // #1: partial specialization where T2 is a pointer to T1

template<typename T, typename T2, typename I>
class A<T*, T2, I> {}; // #2: partial specialization where T1 is a pointer

template<typename T>
class A<int, T*, double> {}; // #3: partial specialization where T1 is int, I is double,
// and T2 is a pointer

template<typename X, typename T, typename I>
class A<X, T*, I> {}; // #4: partial specialization where T2 is a pointer
```

EXERCISE

Write a partial specialization of `VectorMap` for boolean keys. We can have only 2 values for boolean keys. There is no need to keep vectors inside.

Implement properly all currently available functions.

OVERLOAD RESOLUTION



CODERS
SCHOOL

TEMPLATE INSTANTIATION

- Template by itself is not a type, or an object, or a function, or any other entity
- No code is generated from a source file that contains only template definitions
- In order for any code to appear, a template must be instantiated so that the compiler can generate an actual entity and its implementation

The definition of a template must be visible at the point of implicit instantiation, which is why template libraries typically provide all template definitions in the headers.

OVERLOAD RESOLUTION

Selecting the right function to call consist of a few steps.

- Candidate functions (+)
- Viable functions (-)
- Implicit conversions (+)
- Best viable function (-)

After this process compiler has chosen the winner function. The linker knows which function should it be linked against.

CANDIDATE FUNCTIONS

Candidate functions are selected by the name lookup process.

Name lookup:

- ADL (Argument Dependent Lookup)
- Template Argument Deduction -> implicit specializations are generated

If name lookup produces more than one candidate function, the overload resolution is performed to select the function that will be called.

SO TEMPLATES ARE OVERLOADING, RIGHT?

Yes and no.

- Function templates participate in name resolution for overloaded functions, but the rules are different.
- For a template to be considered in overload resolution, the type has to match exactly.
- If the types **do not match exactly, the conversions are not considered** and the template is simply dropped from the set of viable functions.
- That's what is known as "SFINAE" – Substitution Failure Is Not An Error.

SFINAE - EXAMPLE

```
1 #include <iostream>
2 #include <typeinfo>
3
4 template<typename T>
5 void foo(T* x)      { std::cout << "foo<" << typeid(T).name() << ">(T*)\n"; }
6 void foo(int x)     { std::cout << "foo(int)\n"; }
7 void foo(double x)  { std::cout << "foo(double)\n"; }
```

WHICH FUNCTIONS ARE CALLED?

- `foo(42)`
 - matches `foo(int)` exactly
 - for `foo<T>(T*)` there is a substitution failure
- `foo(42.0)`
 - matches `foo(double)` exactly
 - for `foo<T>(T*)` there is a substitution failure
- `foo("abcdef")`
 - matches `foo<T>(T*)` with `T = char` and it wins

FUNCTION OVERLOADS VS FUNCTION SPECIALIZATIONS

```
template<class T> void f(T);      // #1: overload for all types
template<>         void f(int*);  // #2: specialization of #1 for pointers to int
template<class T> void f(T*);    // #3: overload for all pointer types
```

WHICH FUNCTION IS CALLED?

```
f(new int{1});
```

Calls #3, even though #2 (specialization of #1) would be a perfect match

- Only non-template and primary template overloads participate in overload resolution
- The specializations are not overloads and are not considered
- Only after the overload resolution selects the best-matching primary function template, its specializations are examined to see if one is a better match

GENERAL RULE

If possible, create function overloads, not specializations.

MORE INFORMATION

Overload resolution rules on cppreference.com

VARIABLE TEMPLATES (C++17)



CODERS
SCHOOL

WHY?

Eg. different precision

```
template<class T>
constexpr T pi = T(3.1415926535897932385L);

template<class T>
T circular_area(T r) { return pi<T> * r * r; }
```

But it is really rarely used.

SPECIALIZATION EXAMPLE #2 - FIELD VALUES

Remember this code?

```
1 #include <iostream>
2
3 template<typename T>    // primary template
4 struct is_int {
5     static constexpr bool value = false;
6 };
7
8 template<>    // explicit specialization for T = int
9 struct is_int<int> {
10     static constexpr bool value = true;
11 };
12
13 template<typename T>
14 constexpr bool is_int_v = is_int<T>::value;
15
16 int main() {
17     std::cout << is_int_v<char> << '\n';    // prints 0 (false)
18     std::cout << is_int_v<int> << '\n';    // prints 1 (true)
19     return 0;
20 }
```

We mainly use template variables as helpers to class template field values.

Check out `type_traits` on cppreference.com

Every trait has a corresponding helper variable template.

EXERCISE

Write a variable template `is_int_key_v`. It should return a value of the `is_int_key` field in a given template type.

POST-WORK



CODERS
SCHOOL

vector - TEMPLATE CLASS

Implement your own `vector<T>` class. It should have the same behaviour as `std::vector<T>`

Remember about internal member types.

Remember about `bool` specialization.

You don't need to remember about all constructors.

A more advanced version can also have a template parameter `Allocator`.

You don't need to implement everything. Just start and try as much as you can.

remove - TEMPLATE FUNCTION

Implement your own `remove()` function. It should mimic the behaviour of `std::remove()` algorithm.

CODE REVIEW

Send me a link to the repository with your implementation for my code review.

LUKASZ@CODERS.SCHOOL

POST-TEST

The link will be sent after the training with all materials.

RECAP



CODERS
SCHOOL

HOW DID YOU ENJOY THIS TRAINING SESSION?

WHAT HAVE YOU LEARNED TODAY?

POINTS TO REMEMBER

- A template is not a class or a function. A template is a "pattern" that the compiler uses to generate a family of classes or functions.
- In order for the compiler to generate the code, it must see both the template definition (not just declaration) and the specific types/whatever used to "fill in" the template.
- Templates itself are only instantiated upon usage. If some function/method was not used, it is not generated, even for class templates
- Full specializations are allowed for both class and function templates
- Partial specializations are only allowed for class templates
- When looking for possible overloads within templates, only perfect matches are considered. No conversions are tried.
- `template <typename T> == template <class T>`
- `type_traits` has a lot of useful stuff. Use it with `static_assert` or SFINAE technique.

PRE-TEST ANSWERS

PRE-TEST 1/3 🤯

WHICH OF THE FOLLOWING USAGES OF A TEMPLATE FUNCTION ARE CORRECT?

```
template <typename T, typename U>  
U fun(T arg1, U arg2);
```

1. `fun(4, 5)`
2. `fun(4, 5.5)`
3. `fun<int, int>(4, 5)`
4. `fun<int, double>(4, 5.5)`
5. `fun<int, int>(4, 5.5)`

PRE-TEST 2/3 🤯

WHICH OF THE FOLLOWING USAGES OF A TEMPLATE FUNCTION ARE CORRECT?

```
template <typename T, typename U, typename V>  
V fun(T arg1, U arg2);
```

1. `fun(4, 5.5)`
2. `fun(4, 5.5, int)`
3. `fun(4, 5.5, 'c')`
4. `fun<int, double, int>(4, 5.5)`
5. `fun<int, double, char>(4, 5.5)`
6. `fun<int, double, char>(4, 5.5, 'c')`
7. `fun<char>(4, 5.5)`
8. `fun<V = char>(4, 5.5)`

PRE-TEST 3/3 🤯

WHICH OF THE FOLLOWING USAGES OF A TEMPLATE FUNCTION ARE CORRECT?

```
template<typename To, typename From>  
To convert(From f);
```

1. `auto a = convert(3.14);`
2. `auto b = convert<int>(3.14);`
3. `auto c = convert<int, double>(3.14);`
4. `int(*p1)(float) = convert;`
5. `int(*p2)(float) = convert<int, float>;`

RATE THIS TRAINING SESSION

CODERS SCHOOL

