TEMPLATES

CODERS SCHOOL

ŁUKASZ ZIOBROŃ

AGENDA

1. Introduction
2. Template functions
3. Template classes
4. Specialization
5. Variadic templates

ZADANIA

Repo GH `coders-school/templates`

https://github.com/coders-school/templates/tree/master/module1

Zadania wykonywane podczas zajęć online nie wymagają ściągania repo. Pliki będą tworzone od zera.

CO WIECIE O SZABLONACH?

CO DO TEJ PORY UTWORZYLIŚCIE Z WYKORZYSTANIEM SZABLONÓW?

TEMPLATES

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 metaprogramming**.

All templates are evaluated during compilation and only during that phase.

TEMPLATE TYPES

In C++03 we could have

- template classes
- template functions

From C++11 we can also have

- template variables

And from C++20 we will have

- template lambdas

We will cover only template functions and classes here.

TEMPLATE FUNCTIONS

EXAMPLES

Let's assume that we have a function below:

```cpp
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:

```cpp
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:

```cpp
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 type. 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
```

You can play with the code here

FUNCTION TEMPLATE TYPE DEDUCTION

In C++ there is an implemented function template types deduction. 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}`. Unfortunately 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 which 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:

```cpp
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. 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 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 first argument type, because it was defined in template function above as `TypeA`.

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

```cpp
#include <typeinfo>

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

You can use `typeid().name()` to print variable type. You need to include `typeinfo` header for this. You can also notice, that instead of `typename` keyword, you can also use `class` keyword. They are interchangeable.

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

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

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

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

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

or

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

You can also play with the code here

TEMPLATE CLASSES

EXAMPLES

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 it's usage:

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

template <typename T>
class SomeClass {
public:
    T getValue() { return value; }
private:
    T value;
};

int main() {
    SomeClass<int> 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'};
```

EXERCISE 2

Write a template class which create an over-engineered `std::map`. It should hold 2 `std::vectors` inside with the same size, each with different types. Usage should look like this:

```
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
```

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' above. Try to implement as much of `std::map` interface as possible, at least the mentioned above `insert`, `operator[]`, `at`

Use cppreference.

TEMPLATE SPECIALIZATION

SPECIALIZATION EXAMPLES

We have a class `is_int` with boolean field value, which is by default initialized with `false`. However, if we provide an `int` as a parameter type, we want to have a `true` value there. This is how to achieve it:

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

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

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


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;
}
```

You can play with the code here

SPECIALIZATION IN `<type_traits>` LIBRARY

The concept of specialization is heavily used in `<type_traits>` library. Please take a look there.

Generally, we can have different behavior of the code, depending on the type that we provided.

Of course we could write normal structures or functions above, but this solution with templates is more generic. It means that for every type that we provide to `is_int` by default value `false` will be returned.

SPECIALIZATION IN `<type_traits>` LIBRARY EXAMPLE

To achieve mentioned behavior, we can use `std::false_type` and `std::true_type`. This is equivalent code from previous slide with `std::false_type` and `std::true_type` used.

```cpp
#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;
}
```

Interactive version of this code is here

EXERCISE

Write a template class `IsSmallPrime` which holds boolean value `true` or `false`, depending on the integer value passed into template parameter.

You should use specialization for values 2, 3, 5 and 7. In case of 0, 1, 4, 6, 8 and 9 it should hold `false` value.

Hint: Beside of typename or class parameter, templates can also have int parameters: eg. `template <typename T1, class T2, int N> class C {}`.

VARIADIC TEMPLATES

MOTIVATION

Variadic templates can be used to create template functions or classes which accept any numer of arguments of any type.

Do you know `printf()` function (from C)?

```
#include <cstdio>

int main() {
   printf("Hello %s, you are %d years old\n", "John", 25);
   printf("Just a text\n");
   return 0;
}
```

SYNTAX

Templates with variable number of arguments use new syntax of parameter pack, that represents many (or zero) parameters of a template.

```
template<class... Types>
void variadic_foo(Types&&... args)
{ /*...*/ }

int main() {
    variadic_foo(1, "", 2u);
    return 0;
}
```

```
template<class... Types>
class variadic_class
{ /*...*/ };

int main() {
    variadic_class<float, int, std::string> v{};  // default c-tor
    variadic_class v{2.0, 5, "Hello"}; // automatic template type deduction fo
}
```

UNPACKING FUNCTION PARAMETERS

Unpacking group parameters uses new syntax of elipsis operator - `...`

In case of function arguments it unpacks them in the order given in template function call.

It is possible to call a function on a parameter pack. In such case given function will be called on every argument from a function call.

TWO WAYS OF PASSING ARGUMENTS

```cpp
template <typename... Args>
void print_all(Args&&... args) {
    printMany(args...);
    // expands to printMany(arg1, arg2, ..., argN);
}

template <typename... Args>
void print_each(Args&&... args) {
    (printOne(args), ...);  // fold expression from C++17
    // expands to printOne(arg1), printOne(arg2), ..., printOne(argN);
}

int main() {
    print_all("Hello!", "Lukin");
    print_each("Hello!", "Lukin");
    return 0;
}
```

UNPACKING PARAMETER TYPES

```
Foo...                  => Foo1, Foo2, Foo3, etc
vector<Foo>...          => vector<Foo1>, vector<Foo2>, vector<Foo3>, etc
tuple<Foo...>(bar...) => tuple<Foo1, Foo2, Foo3, etc>(bar1, bar2, bar3, etc)
&bar...                 => &bar1, &bar2, &bar3, etc
&&bar...                => &&bar1, &&bar2, &&bar3, etc
```

HEAD TAIL RECURSION

It is also possible to use recursion to unpack every single argument. It requires the variadic template Head/Tail and non-template function to be defined.

```
void variadic_foo() {}

template<class Head, class... Tail>
void variadic_foo(Head const& head, Tail const&... tail) {
    takesOneParam(head);    //action on head
    variadic_foo(tail...);
}
```

HEAD TAIL RECURSION IN CLASSES

It is possible to unpack all types at once (e.g. in case of base class that is variadic template class) or using partial and full specializations.

```
template<int... Number>
struct Sum;

template<int Head, int... Tail>
struct Sum<Head, Tail...> {
    const static int RESULT = Head + Sum<Tail...>::RESULT;
};

template<>
struct Sum<> {
    const static int RESULT = 0;
}

constexpr auto value = Sum<1, 2, 3, 4, 5>::RESULT; // = 15
```

FOLD EXPRESSIONS (C++17)

Fold expressions allow to write compact code with variadic templates without using explicit recursion.

```
template<typename... Args> auto sum(Args... args) {
    return (args + ...);
    // return (... + args);      // the same
    // return (0 + ... + args);  // the same
    // return (args + ... + 0);  // the same
    // return args + ...;        // error - missing parentheses
}

template<typename... Args> bool f(Args... args) {
    return (true && ... && args); // OK
}

template<typename... Args> bool f(Args... args) {
    return (args && ... && args); // error: both operands
}                                 // contain unexpanded
                                  // parameter packs
```

EXERCISE

Write a `print()` function that can print anything on a screen. You should be able to pass any number of parameters to it. Use fold expressions.

Write an `areEven()` function, that

- prints all parameters on screen using above `print()` function
- check if all numbers in parameter pack are even and return a proper boolean value

SOLUTION

```cpp
template<typename ...Args>
void print(Args&&... args) {
    (std::cout << ... << args) << '\n';
}

template<typename... Numbers>
auto areEven(Numbers... nums) {
    print(nums...);
    return ((nums % 2 == 0) && ...);
}
```

Try to add a space between elements in `print()`.

FOLD EXPRESSIONS

Default values when parameter pack is empty

Operator	Value
`*`	`1`
`+`	`int()`
`&`	`-1`
`|`	`int()`
`&&`	`true`
`||`	`false`
`,`	`void()`

HANDLING INHERITANCE FROM VARIADIC CLASSES

```
template<class... Types>
struct Base
{};

template<class... Types>
struct Derived : Base<Types...>
{};
```

`sizeof...` OPERATOR

`sizeof...` returns the number of parameters in a parameter pack.

```
template<class... Types>
struct NumOfArguments {
    const static unsigned value = sizeof...(Types);
};

constexpr auto num = NumOfArguments<A, B, C>::value;  // 3
```

PERFECT FORWARDING

It is recommended to use `&&` and `std::forward` with variadic arguments to handle r-values and l-values correctly. It is so called **perfect forwarding**.

```cpp
#include <utility>

template<typename Type>
void single_perfect_forwarding(Type&& t) {
    callable(std::forward<Type>(t));
}

template<typename... Types>
void variadic_perfect_forwarding(Types&&... args) {
    callable(std::forward<Types>(args)...);
}
```

CONCEPTS

CONCEPTS

Right now when we need special checks on types, it is usually done by using `<type_traits>` library.

Concepts are introduced in C++20 to help programmers to apply a specific requirements on types in templates. Named sets of such requirements are called **concepts**.

Variadic templates can be used to create template functions or classes which accept any numer of arguments of any type.

There are 2 new keywords: `concept` and `requires`.

There will one major advantage of concepts. Error messages from templates will not take a few pages to read. They will be very readable and meaningful one-liners 🙂.

Check an example on cppreference.com

PODSUMOWANIE

CO PAMIĘTASZ Z DZISIAJ?

NAPISZ NA CZACIE JAK NAJWIĘCEJ HASEŁ

POST-WORK

`make_unique` (40 XP)

Implement `make_unique` function. It should be able to take any numer of parameters and it should return a unique_ptr with a given type. The object of a given type should be created with all parameters passed to it. The main challenge is to properly unit tests for l-values and r-values. Do not forget about CI.

```cpp
struct MyType {
    MyType(int&, double&&, bool) { std::cout << "lvalue, rvalue, copy\n"; }
    MyType(int&&, double&, bool) { std::cout << "rvalue, lvalue, copy\n"; }
};

int main(){
    int lvalue{2020};
    std::unique_ptr<int> uniqZero = make_unique<int>();
    auto uniqEleven = make_unique<int>(2011);
    auto uniqTwenty = make_unique<int>(lvalue);
    auto uniqTypeLRC = make_unique<MyType>(lvalue, 3.14, true);
    auto uniqTypeRRC = make_unique<MyType>(2020, 3.14, true);
}
```

CODERS SCHOOL