Software engineering

v2024

Lab 5:
Programming with Modern C++ – Part I

Contents

FESB in cooperation with ZORAJA Consulting d.o.o.

Exercise 1: Templates

Templates in C++ are a way of supporting generic programming. Template mechanism allows definitions of classes, functions or type aliases to use types or values as parameters, i.e. templates are parameterized by template parameters. We differentiate *function templates* (1) and *class templates* (2).

(1) template <typename T>

T Max(T x, T y);

(2) template <typename T, std::size_t N>

class Stack;

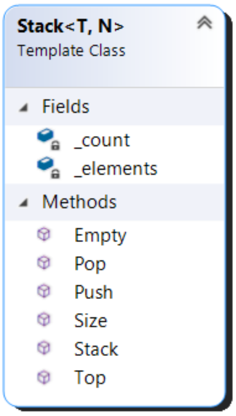
The template provides a pattern, but the actual code is generated by the compiler according to the provided argument types in the process of template instantiation. In order for the compiler to generate the code, both the declaration and the definition of the template must be visible. A common solution is to provide definition inside the header file, but outside the declaration to keep the readability.

Template parameters can be:

1. Type template parameters → defined using typename
2. Non-type template parameters
3. Template template parameters

When considering function overloading, if the overload(s) would perform the same operations but on different types, a function template might be more appropriate.

The UML below represents the class that will be implemented.



Create a Project

1. Create an Empty Project named Templates.
2. Add file Program.cpp to the project.

Add a Header File to the Project

1. Add a file Functions.h to the project.
2. Protect the file from multiple inclusions.
3. Include iostream header file.
4. Add a namespace tpl.
5. Inside the namespace add a templated function Max with a typed template parameter T. The function should take in two parameters of type T, which is also the return type.
6. Implement the function so that it returns the larger of the two

Add a Header File to the Project

1. Add a file Stack.h to the project.
2. Protect the file from multiple inclusions.
3. Include iostream, array and cassert header files.
4. Inside the tpl namespace add a templated class Stack with a typed template parameter T and a non-type parameter N of type size_t.
5. Add a private member variable _elements of type array<T, N>
6. Add a private member variable _count of type size_t.
7. Declare a constructor.
8. Declare a function Push that returns a void and takes in a constant l-value reference to T.
9. Declare a function Pop that returns a void.
10. Declare a constant function Top that returns a constant l-value reference to T.
11. Implement a constant function Empty with a return type of bool, which checks if the stack is empty (checks the _count variable).
12. Implement a constant function Size which returns a size_t. The function should return the value of the _count variable.
13. Recall the effects that functions Pop, Push and Top are supposed to have on a stack data structure and implement them outside of the class.
14. Implement an inline, templated function Show with a typed template parameter T and a non-type parameter N of type size_t. The function should take in an object

of type **Stack<T, N>**, an int representing the number of elements in the stack, and output the content of the stack using the Top and Pop functions.

Add a Header File to the Project

1. Add a file Variadic.h to the project.
2. Protect the file from multiple inclusions.
3. Include the iostream header file.
4. Inside the tpl namespace add an inline function Print which returns a void and outputs a new line to the console.
5. Add a variadic, templated function Print which can take in an arbitrary number of arguments. Implement the function using recursion so that it outputs all of its arguments. A suggested signature is given below:

```
template <typename T, typename... Ts>
inline void Print(T head, Ts... rest)
```

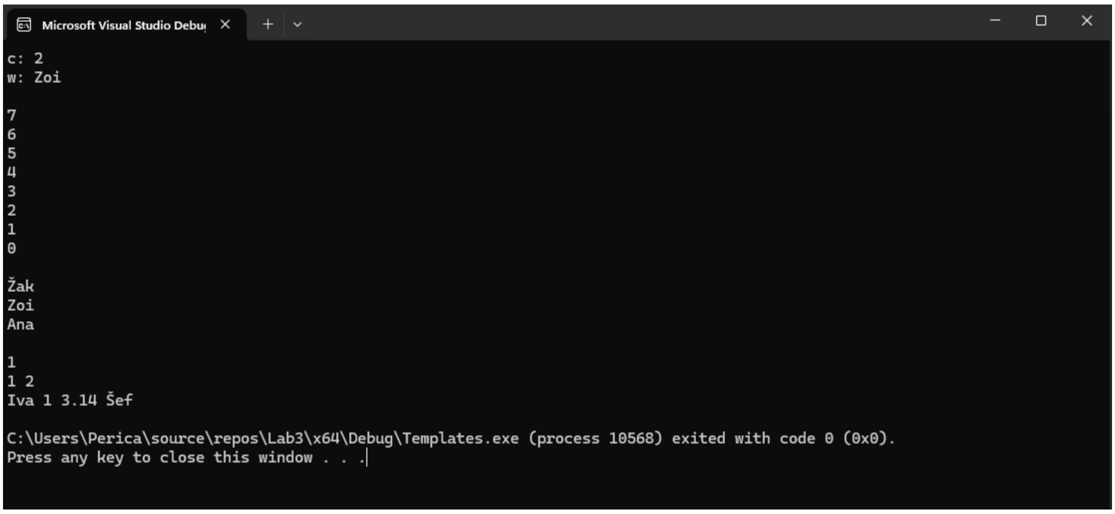
Test Associations

1. Include iostream.h and string in Program.cpp.
2. Include Helpers.h, Functions.h, Stack.h and Variadic.h.
3. Use the using directive for namespaces std and tpl.
4. Add an alias template for the Stack<T, 4> and name it WStack_t.
5. Inside the main function enable UTF8 on the wcout stream.
6. Add two variables a and b of type int and initialize them to 1 and 2.
7. Use the variables as arguments for the Max function and output the result.
8. Add two wstring variables w1 and w2.
9. Call function Max and pass w1 and w2 as arguments. Output the result to the console.
10. Declare a variable istack of type Stack<int, 10>.
11. Using the Push function fill the stack with eight numbers.
12. Call the function Show to output the content of istack.
13. Add a variable wstack of type WStack_t<wstring>.
14. Push three values into the stack.
15. Call function Show to output the content of wstack.
16. Call function Print and pass number 1 as an argument.

17. Call function Print and pass it an integer and a double.

18. Call function Print and pass it a wstring, an int, a double and another wstring.

Run the App



```
c: 2
w: Zoi

7
6
5
4
3
2
1
0

Žak
Zoi
Ana

1
1 2
Iva 1 3.14 Šef

C:\Users\Perica\source\repos\Lab3\x64\Debug\Templates.exe (process 10568) exited with code 0 (0x0).
Press any key to close this window . . .
```

Exercise 2: Variadics

Variadic templates are templates which are defined so that they take in an arbitrary number of arguments of arbitrary types. A list of arguments that is passed to a variadic template can be separated into the first argument (referred to as the head) and the rest (referred to as the tail). After performing some action on the head, the function gets recursively called on the tail. A separate function is needed to deal with the last call when there are no arguments left in the list.

```
template <typename… Types>
void f(Types… args);
```

In the example above, typename… denotes that Types is a template parameter pack. The ellipsis in the type of the function parameter denotes that args is a function parameter pack.

Create a Project

1. Create an Empty Project named Variadics.
2. Add file Program.cpp to the project

Add a File

1. Add file Templates.h to the project.

2. Protect the file from multiple inclusions.

3. Include windows.h

4. Add namespace vtl.

5. Create a block which executes only if DEBUG or _DEBUG symbol is defined. To do so use the #if preprocessor directive.

6. Inside the block define a function Log which returns a void and outputs a new line using the OutputDebugStringW function.

7. Define an inline, variadic, templated function Log with the following signature:

```
template <typename T, typename... Ts>
inline void Log(const T& head, const Ts&... tail);
```

8. Output the first parameter and use recursion to output the rest.

9. Overload the Log function so that, in comparison to the previous implementation, it takes in additional two parameters of type int and uses recursive calls only if both parameters are true.

10. In the #else block add the following line:

    a. #define Log(x, …);

Test Variadics

1. In Program.cpp include iostream header file.

2. Include Templates.h.

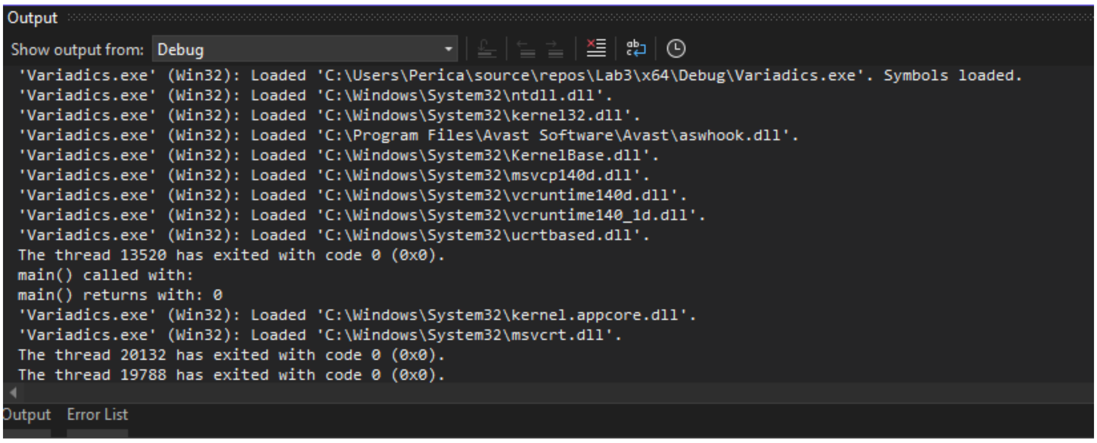
3. Use the using directive for namespaces std, vtl and std::string_literals.

4. Inside the main function call Log and pass in values for channel and verbosity. Using string literals and suffix s, pass two more arguments of type wstring.

5. Call the Log function again, but this time with no channel and verbosity values.

Run the App

1. Running the project should result in a similar debug output.



```
'Variadics.exe' (Win32): Loaded 'C:\Users\Perica\source\repos\Lab3\x64\Debug\Variadics.exe'. Symbols loaded.
'Variadics.exe' (Win32): Loaded 'C:\Windows\System32\ntdll.dll'.
'Variadics.exe' (Win32): Loaded 'C:\Windows\System32\kernel32.dll'.
'Variadics.exe' (Win32): Loaded 'C:\Program Files\Avast Software\Avast\aswhook.dll'.
'Variadics.exe' (Win32): Loaded 'C:\Windows\System32\KernelBase.dll'.
'Variadics.exe' (Win32): Loaded 'C:\Windows\System32\msvcp140d.dll'.
'Variadics.exe' (Win32): Loaded 'C:\Windows\System32\vcruntime140d.dll'.
'Variadics.exe' (Win32): Loaded 'C:\Windows\System32\vcruntime140_1d.dll'.
'Variadics.exe' (Win32): Loaded 'C:\Windows\System32\ucrtbased.dll'.
The thread 13520 has exited with code 0 (0x0).
main() called with:
main() returns with: 0
'Variadics.exe' (Win32): Loaded 'C:\Windows\System32\kernel.appcore.dll'.
'Variadics.exe' (Win32): Loaded 'C:\Windows\System32\msvcrt.dll'.
The thread 20132 has exited with code 0 (0x0).
The thread 19788 has exited with code 0 (0x0).
```

Exercise 3: Pefect forwarding

Perfect forwarding gives the ability to preserve an argument's value category (l-value or r-value) when passing it to another function.
"*If a variable or parameter is declared to have type T&& for some deduced type T, that variable or parameter is a **universal reference**.*"
Most universal references are parameters to function templates. If there is no type deduction, T&& always means r-value reference.
In perfect forwarding we use std::forward, which represents a conditional cast (whereas std::move is unconditional) and takes in a universal reference and casts it into an r-value only if the expression bound to it is also an r-value (otherwise it is treated as an l-value).

Create a Project

1. Create an Empty Project named Perfect Forwarding.
2. Add file Program.cpp to the project

Add a File to the Project

1. Add file Types.h to the project.

2. Protect the file from multiple inclusions.

3. Include iostream and string headers.

4. Inside namespace abc add a struct Something.

5. Implement a constructor which takes in a parameter of type int.

6. Implement a destructor.

7. Add another struct named Nešto.

8. Implement a constructor which takes in parameters of type int, double and wstring.

9. Implement a destructor.

Add a File to the Project

1. Add file Functions.h to the project.

2. Protect the file from multiple inclusions.

3. Include iostream and string headers.

4. Include Smart Pointer.h from Operators project.

5. Inside namespace abc implement a function f which takes in an l-value reference to int and returns a void. The function should output the value of the argument passed to it.

6. Overload the function f so that it takes in an r-value reference to int and outputs the value.

7. Add an inline, templated function Forward with typed template parameter T, which takes in an universal reference to T and uses std::forward to perfectly forward it to function f.

8. Implement an inline, templated, variadic function named MakeSmart. Below is a suggested signature:

```
template <typename T, typename... Ts>
inline mem::SmartPtr<T> MakeSmart(Ts&&... rest)
```

Output the number of parameters and use perfect forwarding to turn them into SmartPtrs.

Test Perfect Forwarding

1. Inside Program.cpp include iostream headers.

2. Include Functions.h, Types.h and Helpers.h.

3. Use the using directive for namespaces std and abc.

4. Inside the main function enable UTF8 in the wcout stream.

5. Add a local object a of type int and initialize it to the value of 1.

6. Add an l-value reference to int named l and assign a to it.

7. Pass l to function f.

8. Add an r-value reference to int named r and assign 2 to it.

9. Pass r to function f.

10. Pass a temporary int object to f.

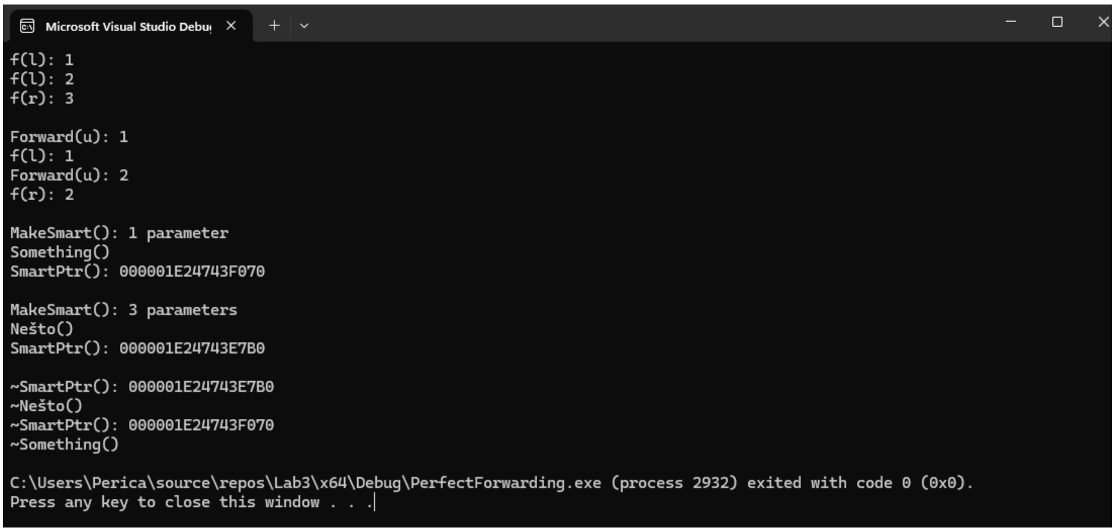
11. Pass a to the function Forward.

12. Pass a temporary object of type int to the function Forward.

13. Call function MakeSmart with Something as its template parameter and pass a number as an argument. Assign the result to a variable something.

14. Call function MakeSmart with Nešto as its template parameter. Pass an int, a double and a wstring as arguments. Assign the result to a variable nešto.

Run the App



```
f(l): 1
f(l): 2
f(r): 3

Forward(u): 1
f(l): 1
Forward(u): 2
f(r): 2

MakeSmart(): 1 parameter
Something()
SmartPtr(): 000001E24743F070

MakeSmart(): 3 parameters
Nešto()
SmartPtr(): 000001E24743E7B0

~SmartPtr(): 000001E24743E7B0
~Nešto()
~SmartPtr(): 000001E24743F070
~Something()

C:\Users\Perica\source\repos\Lab3\x64\Debug\PerfectForwarding.exe (process 2932) exited with code 0 (0x0).
Press any key to close this window . . .
```