Software engineering

v2024

Lab 4:
Programming with Modern C++ – Part I

Contents

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

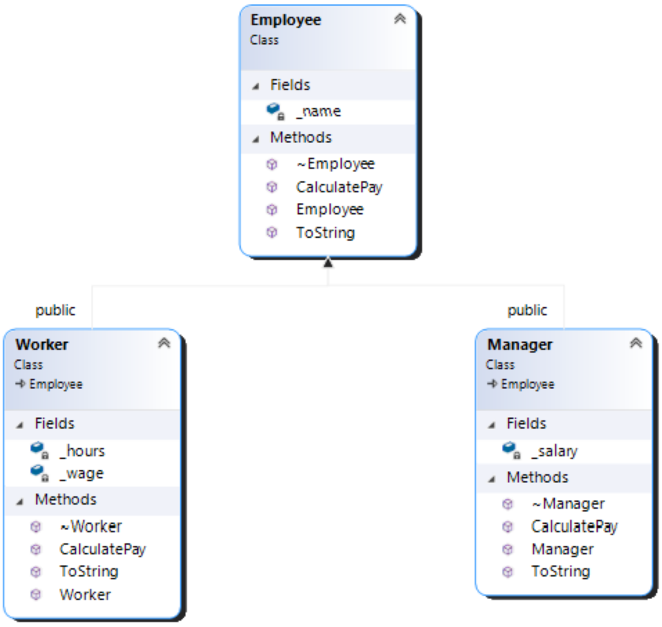
Exercise 1: Generalization

One of the pillars of OOP is *inheritance*, which provides a way to express hierarchical relationships. A class *hierarchy* is a set of classes created by *derivation*. When representing concepts, common features are extracted into what is known as a *base class*. Classes that share those features (i.e. inherit from the base) but expand the concept with additions (data and functions) of their own, are known as *derived classes*. The base represents a general type, whereas the derived classes are more specific and detailed.

Classes can either be *concrete* (if their representation is part of their definition) or *abstract* (contain *pure virtual functions*, i.e. leave the implementation out). A pure virtual function is denoted with syntax =0 and forces derived classes to provide the implementation. Non-pure virtual functions only indicate the *possibility* of being overridden in derived classes.

Resolving function calls to their proper implementation is achieved via *virtual function table (vtbl)*, which is a table of pointers to functions and is added for each class that contains virtual functions. Beside instance data, objects will have *pointers to virtual tables (vptr)*, which make it possible to access the v-table.

The UML in the figure below shows the class hierarchy that will be created in the exercise:



Create a Project

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

Add a Header File to the Project

1. Add a file Employee.h to the project.
2. Protect the file from multiple inclusions.
3. Include the iostream and string header files.
4. Add namespace com.
5. Add class Employee with a private member variable _name of type wstring.
6. Implement a constructor which takes in a parameter of type wstring. Prevent implicit conversion to Employee.
7. Implement a virtual function CalculatePay so that it returns 0.0.
8. Implement a constant, virtual function ToString which returns the value of _name.
9. Implement a virtual destructor.

Add a Header File to the Project

1. Add a file Worker.h to the project.
2. Protect the file from multiple inclusions.
3. Include Employee.h.
4. Add the namespace com.
5. Add a class named Worker which publicly inherits from Employee class.
6. Add private member variables _wage and _hours of types double and int, respectively.
7. Implement the constructor.
8. Override the function CalculatePay so that it calculates a worker’s paycheck.
9. Override the function ToString to output the type name and the paycheck.
10. Implement a virtual destructor.

Add a Header File to the Project

1. Add file Manager.h to the project.
2. Protect the file from multiple inclusions.
3. Include Employee.h.
4. Inside the com namespace add a class Manager which inherits from Employee class and has a private member variable _salary of type double.
5. Implement the constructor.

6. Override the CalculatePay function to return the manager’s salary.
7. 7. Override the function ToString to output the type and the salary.
8. Implement a destructor.

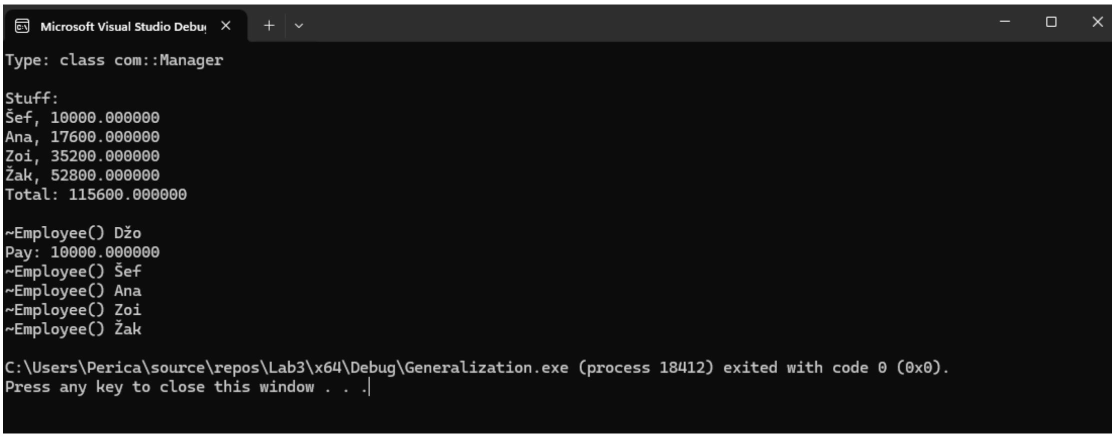
Test Inheritance

1. In Program.cpp include vector, iostream and typeinfo header files.
2. Include Helpers.h, Employee.h, Worker.h and Manager.h.
3. Use the using directive for namespaces com and std.
4. Inside the main function enable UTF8 on the wcout stream helper function.
5. Create a dynamic object of type Manager and assign it to a variable named manager.
6. Output the type name of the manager object by using typeid.
7. Declare a vector of pointers to type Employee named stuff.
8. Push the manager object back into the vector. Add three more entries.
9. Output the content of the vector to the console.
10. Add a variable total of type double and initialize it to 0.0.
11. Calculate the sum of paychecks of all the Employees in the vector.
12. Output the result to the console.
13. Add a dynamic object of type Employee and assign it to a variable named employee.

    • Can you assign employee to manager?
14. Delete the object employee points to.
15. Assign manager to employee.
16. Output the employee’s pay.
17. Delete the objects that pointers from the vector point to.

    • What would happen if you tried to assign a wstring to a variable of type Employee?

Run the App



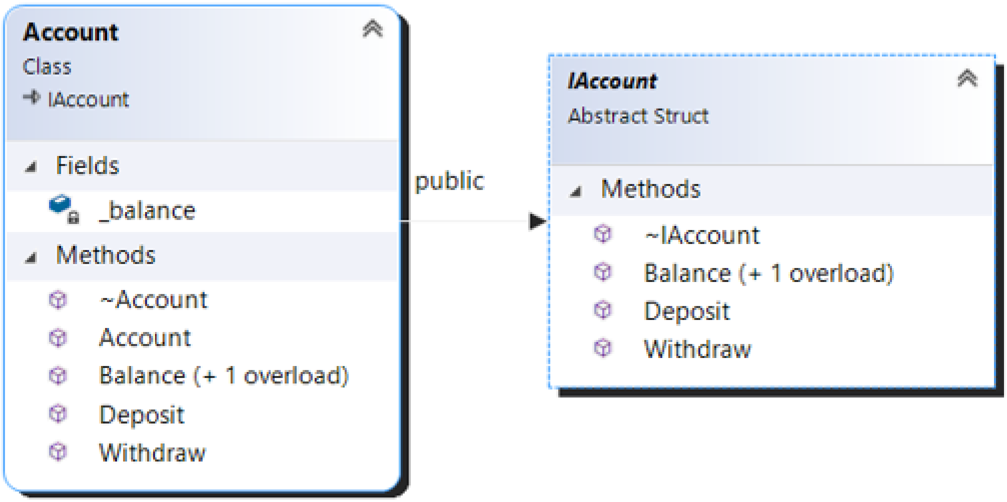
```
Type: class com::Manager

Stuff:
Šef, 10000.000000
Ana, 17600.000000
Zoi, 35200.000000
Žak, 52800.000000
Total: 115600.000000

~Employee() Džo
Pay: 10000.000000
~Employee() Šef
~Employee() Ana
~Employee() Zoi
~Employee() Žak

C:\Users\Perica\source\repos\Lab3\x64\Debug\Generalization.exe (process 18412) exited with code 0 (0x0).
Press any key to close this window . . .
```

Exercise 2: Interfaces

As mentioned in the previous exercise, classes with one or more pure virtual functions are abstract classes and cannot be instantiated. Because objects of derived classes are usually manipulated through the interface of the base class, a *virtual destructor* is essential for an abstract class (if the object is deleted through a pointer to the base class, the virtual function call mechanism enables the invocation of the proper destructor).

Abstract classes support *interface inheritance*, as opposed to implementation inheritance. The interface represents a 'contract' in a way that all types implementing it are obliged to provide implementation for its functions. In C++ interfaces are usually represented as structs due to the default public accessibility of its members.

The UML in the figure below represents the interface which you will create in this exercise and the class that will implement it:



Create a Project

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

Add a File to the Project

1. Add file IAccount.h to the project.
2. Protect the file from multiple inclusions.
3. Include iostream header file.

4. Inside a namespace fin add a struct named IAccount.
5. Inside the struct declare a pure virtual function named Deposit, which takes in a double and returns a bool.
6. Declare a pure virtual function Withdraw, which takes in a double and returns a bool.
7. Declare a pure virtual function Balance, which takes in a double and returns a void.
8. Declare a constant, pure, virtual function named Balance, with no parameters and returns a double.
9. Declare a virtual destructor.
10. Outside of the class provide an inline implementation of the destructor which outputs the name of the function (__func__ identifier).

Add a Header File to the Project

1. Add file Account.h to the project.
2. Protect the file from multiple inclusions.
3. Include iostream and IAccount.h.
4. Inside the fin namespace add a class Account which publicly derives from IAccount.
5. Add a private member variable _balance of type double.
6. Implement a constructor which takes in a parameter of type double.
7. Override the Deposit function so that it increases the balance by the amount received in the argument.
8. Override the Withdraw function so that it decreases the balance by the amount specified in the argument.
9. Override the Balance function to set a new value of balance.
10. Override the Balance function (no parameters) to return the value of _balance.
11. Implement a virtual destructor.

Add a Header File to the Project

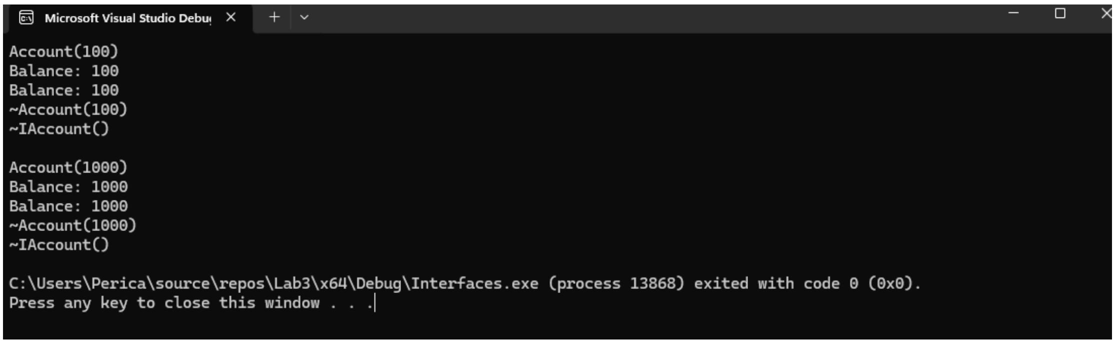
1. Add file Factory.h to the project.
2. Include cassert, IAccount.h and Account.h header files.
3. Inside the fin namespace implement an inline function CreateAccount which takes in a parameter of type double and returns a pointer to IAccount, so that it returns a pointer to a dynamically created Account object.

4. Implement an inline function CreateAccount, which takes in a pointer to a pointer (pp) to IAccount and a double, while the return type is void. Make the pp point to a dynamically created object of type Account initialized with the second parameter.
5. Implement an inline function TestAcount which takes in a pointer to IAccount and two values of type double. Call the function Balance on the pointer and output the value. Then call Deposit and Withdraw and output the new balance. Use the assert macro to see if the initial balance matches the one.

Test Interfaces

1. In Program.cpp include iostream, IAccount.h and Factory.h.
2. Use the using directive for namespaces std and fin.
3. In the main function call CreateAccount to create an Account object initialized with 100.0. Assign the object to variable account.
4. Pass account to the TestAccount function along with 100.0 as an amount for both depositing and withdrawing.
5. Delete the object account points to.
6. Use account and 1000.0 as arguments for a second call to CreateAccount.
7. Call TestAccount once again.
8. Delete the object account points to.

Run the App



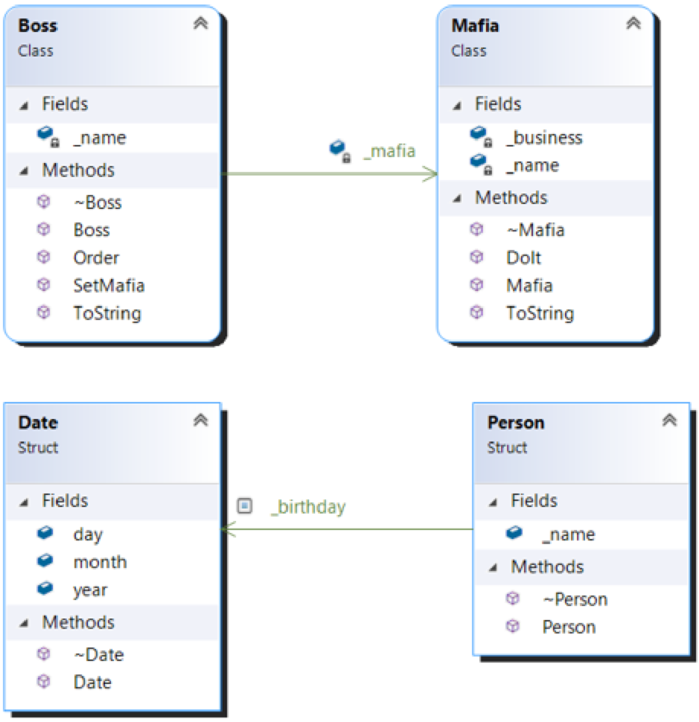
```
Account(100)
Balance: 100
Balance: 100
~Account(100)
~IAccount()

Account(1000)
Balance: 1000
Balance: 1000
~Account(1000)
~IAccount()

C:\Users\Perica\source\repos\Lab3\x64\Debug\Interfaces.exe (process 13868) exited with code 0 (0x0).
Press any key to close this window . . .
```

Exercise 3: Associations

Relationships between classes can be either a generalization, association or a parametric relationship. An association is a relationship which indicates that an object of one type is connected and can navigate to object(s) of another type.

In this exercise you will implement types and associations between them.

The UML in the figure below shows the classes that will be implemented:



Create a Project

3. Create an Empty Project named Associations.
4. Add file Program.cpp to the project.

Add a Header File to the Project

1. Add file Date.h to the project.
2. Protect the file from multiple inclusions.
3. Include the iostream and string header files.
4. Add a namespace abc.
5. Inside the namespace add a struct Date with variables day, month and year of type int.

6. Implement a constructor which has three parameters of type int and initializes the member variables.

7. Implement a virtual destructor.

Add a Header File to the Project

1. Add a file Person.h to the project.
2. Protect the file from multiple inclusions.
3. Include iostream and string header files.
4. Include Date.h.
5. Inside the abc namespace add a struct Person. Add a variable _name of type wstring and a constant Date variable named _birthday.
6. Implement a constructor which initializes the member variables.
7. Implement a virtual destructor.

Add a Header File to the Project

1. Add file Mafia.h to the project.

2. Protect the file from multiple inclusions.

3. Include iostream and string header files.

4. Inside mob namespace add a class named Mafia, with two private wstring member variables named _name and _business. Set _business to nullptr.

5. Implement a constructor which will initialize the member variables.

6. Implement a constant virtual function ToString.

7. Implement a virtual function DoIt, which returns a void and outputs what the mafia does (_business).

8. Implement a virtual destructor.

Add a Header File to the Project

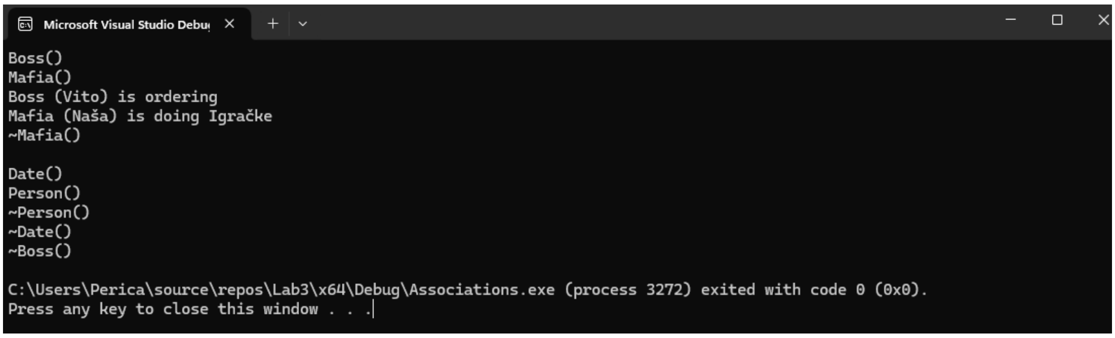
1. Add a file Boss.h to the project.
2. Protect the file from multiple inclusions.
3. Include iostream, string and Mafia.h header files.
4. Inside the mob namespace add a class Boss which has two private member variables: _name of type wstring, and a pointer _mafia to type Mafia. Set _mafia to nullptr.
5. Implement a constructor which initializes the _name variable.

6. Implement a constant, virtual function ToString.
7. Implement a function SetMafia which takes in a pointer to Mafia and assigns it to _mafia member variable.
8. Implement a virtual function Order which returns a void and calls the function DoIt on _mafia.
9. Implement a virtual destructor.

Test Associations

1. Inside Program.cpp include Mafia.h, Boss.h, Person.h and Helpers.h.
2. Use the using directive for namespaces std, mob and abc.
3. Add a global object of type Boss.
4. Inside the main function enable UTF8 on the wcout stream.
5. Add a dynamic Mafia object and assign it to a variable mafia.
6. Associate mafia with boss by using the function SetMafia.
7. Call the function Order on boss.
8. Delete the object mafia points to.
9. Add a temporary Person object without assigning it to a variable.

Run the App



```
Boss()
Mafia()
Boss (Vito) is ordering
Mafia (Naša) is doing Igračke
~Mafia()

Date()
Person()
~Person()
~Date()
~Boss()

C:\Users\Perica\source\repos\Lab3\x64\Debug\Associations.exe (process 3272) exited with code 0 (0x0).
Press any key to close this window . . .
```

Exercise 4: 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;

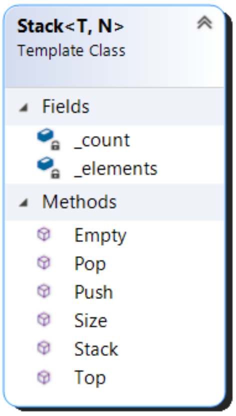
The template provides a pattern, but the actual code is generated by the compiler according to the provided argument types in the process of template instantiation. In order for the compiler to generate the code, both the declaration and the definition of the template must be visible. A common solution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



Stack<T, N>
Template Class

Fields
- _count
- _elements

Methods
- Empty
- Pop
- Push
- Size
- Stack
- Top

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)
```

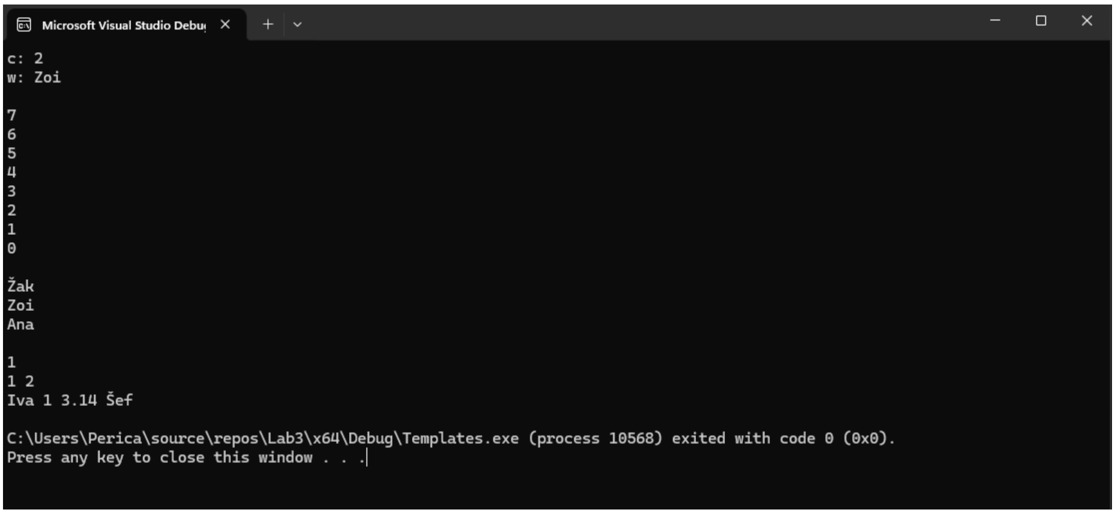
Test Associations

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 . . .
```