Class Templates

- Assume a container class, such as stack. Do you have to write a separate stack code for a stack that stores ints, doubles, strings?
- Answer is no! That would be too inefficient!
- Solution is: Similar to functions, classes can also be parameterized with one or more types.
- Container classes are typical examples for this feature
- By using class templates, you can implement such container classes while the element type is still open.
Running example: Stack

- Interface of a stack
  - void push(Element)
  - void pop()
  - bool empty()
  - Element top()

- We need to declare the stack. Declaring class templates is similar to declaring function templates: Before the declaration, a statement declares an identifier as a type parameter. T is usually used as an identifier:

```cpp
template<typename T>
class Stack {
};
```
Running example: Stack

- Inside the class template, T can be used just like any other type to declare members and member functions.
- In this example, T is used to declare the type of the elements as vector of Ts, to declare push() as a member function that gets a constant T reference as an argument, and to declare top() as a function that returns a T:

```cpp
template <typename T>
class Stack {

private:
  std::vector<T> elems; // elements.

public:
  Stack(); // constructor.
  void push(T const&); // push element.
  void pop(); // pop element.
  T top() const; // return top element.
};
```
To define a member function of a class template, you have to specify that it is a function template, and you have to use the full type qualification of the class template.

You have 2 options for the implementation of the member functions

- Inline – Inside of class declaration

```cpp
template <typename T>
class Stack {

    ...

    void push (T const& elem) {
        elems.push_back(elem);  // append copy of passed elem
    }

    ...

};
```

- Outside of class declaration

```cpp
template <typename T>
void Stack<T>::push (T const& elem) {
    elems.push_back(elem);  // append copy of passed elem
};
```
Running example: Stack
Implementation of member functions

- Implementation of pop and top

```cpp
template<typename T>
void Stack<T>::pop ()
{
    if (elems.empty()) {
        throw std::out_of_range("Stack<>::pop(): empty stack");
    }
    elems.pop_back(); // remove last element.
}

template<typename T>
T Stack<T>::top () const
{
    if (elems.empty()) {
        throw std::out_of_range("Stack<>::top(): empty stack");
    }
    return elems.back(); // return copy of last element.
}
```
```cpp
#include <iostream>
#include <string>
#include <cstdlib>
#include "stack.hpp"

int main()
{

    try {
        Stack<int> intStack;  // stack of ints
        Stack<std::string> stringStack;  // stack of strings

        // manipulate int stack
        intStack.push(7);
        std::cout << intStack.top() << std::endl;

        // manipulate string stack
        stringStack.push("hello");
        std::cout << stringStack.top() << std::endl;
        stringStack.pop();
        stringStack.pop();
    }
    catch (std::exception const& ex) {
        std::cerr << "Exception: " << ex.what() << std::endl;
        return EXIT_FAILURE;  // exit program with ERROR status
    }

}.
```
Running example: Stack Use of Class template

- By declaring type Stack<int>, intStack is created as an object that uses a vector of ints as elements and, for all member functions that are called, code for this type is instantiated.

- Code is instantiated only for **member functions that are called**.

- For class templates, member functions are instantiated only when they are used.

- This, saves time and space.
C++ Code Jam

- Implement the Stack template as a header file and test it!
**Type alias - typedef**

- By using a type definition, you can make using a class template more convenient:

  ```
  typedef Stack<int> IntStack;
  ```

- in C++ a type definition does define a "type alias" rather than a new type. Thus, after the type definition `IntStack` and `Stack<int>` are the same type and can be used for and assigned to each other.

- Template arguments may be any type, such as pointers to floats or even stacks of ints:

  ```
  Stack<float*> floatPtrStack; // stack of float pointers.
  Stack<Stack<int> > intStackStack; // stack of stack of ints.
  ```

Don't forget to put this space here
Specializing class templates

- You can specialize a class template for certain template arguments.

- Similar to the overloading of function templates, specializing class templates allows you to optimize implementations for certain types or to fix a misbehavior of certain types for an instantiation of the class template.

- However, if you specialize a class template, you must also specialize all member functions.

- Although it is possible to specialize a single member function, once you have done so, you can no longer specialize the whole class.
Specializing class templates

- To specialize a class template, you have to declare the class with a leading `template<>` and a specification of the types for which the class template is specialized.

- In this example, a deque instead of a vector is used to manage the elements inside the stack inorder to demonstrate that the implementation of a specialization might look very different from the implementation of the primary template.

```cpp
template<>

class Stack<std::string> {
   private:
      std::deque<std::string> elems; // elements

   
   void Stack<std::string>::push (std::string const& elem) {
      elems.push_back(elem); // append copy of passed elem.
   }

```
Use case: STL stack

- A deque is used in the primary class template in class `std::stack<>` of the C++ standard library.

- Check STL files `stack` and `stl_stack.h` to see the implementation

Download STL (v3.3) source code from here:
http://www.sgi.com/tech/stl/stl.tar.gz
Partial specialization of class templates

- Class templates can be partially specialized. You can specify special implementations for particular circumstances, but some template parameters must still be defined by the user.

```cpp
template <typename T1, typename T2>
class MyClass {
    ...
};

// partial specialization: both template parameters have same type.
template <typename T>
class MyClass<T,T> {
    ...
};

// partial specialization: second type is int.
template <typename T>
class MyClass<T,int> {
    ...
};

// partial specialization: both template parameters are pointer types.
template <typename T1, typename T2>
class MyClass<T1*,T2*> {
    ...
};
```
**Example: Finding specializations**

- Find the specializations for each of the following:

  Check `class_spec.cpp` for the code

```cpp
MyClass<int, float> mif;
MyClass<float, float> mff;
MyClass<float, int> mfi;
MyClass<int*, float*> mp;
MyClass<int, int> m;
MyClass<int*, int*> m;
```
Example: Finding specializations

- Find the specializations for each of the following:

  ```
  MyClass<int, float> mif;    // uses MyClass<T1, T2>
  MyClass<float, float> mff;  // uses MyClass<T, T>
  MyClass<float, int> mfi;    // uses MyClass<T, int>
  MyClass<int*, float*> mp;   // uses MyClass<T1*, T2*>  
  MyClass<int, int> m;        // ERROR: matches MyClass<T, T>
                              //       and MyClass<T, int>
  MyClass<int*, int*> m;      // ERROR: matches MyClass<T, T>
                              //       and MyClass<T1*, T2*>  
  ```

- To resolve the second ambiguity, you can provide an additional partial specialization for pointers of the same type:

  ```
  template <typename T>
  class MyClass<T*, T*> {
  ...
  }
  ```
Default template arguments

- For class templates you can also define default values for template parameters. These values are called default template arguments.

- With the following code you declare a stack for doubles that uses a `std::deque<>` to manage the elements internally

```cpp
template <typename T, typename CONT = std::vector<T>>
class Stack {
    private:
    CONT elems; // elements.

    public:
    void push(T const&); // push element.
    void pop(); // pop element.
    T top() const; // return top element.
    bool empty() const {
        return elems.empty();
    }
};

Stack<double, std::deque<double>>;
```
A class template is a class that is implemented with one or more type parameters left open.

To use a class template, you pass the open types as template arguments. The class template is then instantiated (and compiled) for these types.

For class templates, only those member functions that are called are instantiated.

You can specialize class templates for certain types.

You can partially specialize class templates for certain types.

You can define default values for class template parameters. These may refer to previous template parameters.