Class Templates

- Class templates encourage software usability by enabling a variety of type-specific classes to be instantiated from a single class template.
- Class templates are called *parameterized types*, because they require one or more *type parameters* to specify how to customize a generic class template to form a *class-template-sepecialization*.
- To produce many specializations you write only one class-template definition.
- When a particular specialization is needed, you use a concise, simple notation, and the compiler writes the specialization source code.
- As an example, here we have a Stack class template that is a template for creating many Stack class-template specializations, such as "Stack of doubles", "Stack of ints", "Stack of Chars", etc,

```
template<typename T>
class Stack
{
public:

    // return top element of the stack
    T top()
    {
        if (isEmpty())
        {
            // stack is empty
            return -1;
        }

        return mStack[mSize - 1];
    }

    // push an element onto the stack
    void push(T value)
    {
        mStack[mSize++] = value;
    }

    // pop an element off the stack
    T pop()
    {
        if (isEmpty())
        {
            // stack is empty
            return NULL;
        }

        T pop = top();

        // clear the top value
```

```
        mStack[mSize--] = 0;

        return pop;
    }

    // determine if stack is empty
    bool isEmpty() const
    {
        return (mSize == 0);
    }

    // return size of stack
    int getSize()
    {
        return mSize;
    }

private:
    int mSize = 0;
    int mStack[MAX_SIZE] = {};
};
```

- The Stack class-template definition looks like a conventional class definition, with a few key differences. First, it's preceded by `template<typename T>`.
- All class templates begin with keyword `template` followed by a list of template parameters enclosed in angle brackets (`< >`); each template parameter that represents a type *must* be preceded by either of the *interchangeable* keywords `typename` or `class` (though `typename` is preferred).
- The type parameter `T` acts as a placeholder for the `Stack`'s element type.
- The names of type parameters must be *unique* inside a template definition. You need nor specifically use identifier `T`—any valid identifier can be used.
- The element type is mentioned generically throughout the `Stack` class definition as `T`.
- The type parameter becomes associated with a specific type when you create an object using the class template—at that point, the compiler generates a copy of the class template in which all occurrences of the type parameter are replaced with the specified type.
- Note: it is not possible to write the implementation of a template class in a separate .cpp file from the .hpp if you intend to write a template class library and distribute it with header and lib files to hide the implementation. There are workarounds, but not practical ones for distribution.
- As such, Templates are typically defined in headers, which are then `#included` in the appropriate source-code files. For class templates, this means that the member functions are also defined in the header-typically inside the class definition's body.
- The member-function definitions of a class template are *function templates*, but are not preceded with the template keyword when they're defined within the class template's body. However, they do use the class template's template parameter `T` to represent element type.

- Although not sone so in this `Stack` class template, member-function definitions can appear outside a class template definition. If you do this, each must begin with the template keyword followed by the same set of template parameters as the class template. In addition, the member functions must be qualified with the class name and scope resolution operator, e.g.,

```
template <typename T>
void Stack<T>::push(T value)
{
      mStack[mSize++] = value;
}
```

- `Stack<T>: :` indicates that `push()` is in the scope of class `Stack<T>`. Note: the Standard Library's container classes tend to define all their member functions *inside* their class definitions.
- Here is an example of using the `Stack` class template with a stack of `int` and a stack of `char`.

```
int main()
{
    // demonstrate Stack of ints
    Stack<int>* intStack = new Stack<int>();

    cout << "stack size:" << intStack->getSize() << endl;

    cout << "adding items" << endl;
    intStack->push(11);
    intStack->push(13);
    intStack->push(18);
    cout << "stack size:" << intStack->getSize() << endl;

    cout << "pop:" << intStack->pop() << endl;
    cout << "stack size:" << intStack->getSize() << endl;
    cout << endl;


    // demonstrate Stack of chars
    Stack<char>* charStack = new Stack<char>();

    cout << "stack size:" << charStack->getSize() << endl;

    cout << "adding items" << endl;
    charStack->push('a');
    charStack->push('b');
    charStack->push('c');
    charStack->push('d');
    cout << "stack size:" << charStack->getSize() << endl;

    cout << "pop:" << charStack->pop() << endl;
    cout << "stack size:" << charStack->getSize() << endl;
```

```
    delete intStack;
    delete charStack;

    return 0;
}
```

- Notice that the values pop off in *last-in,first-out* order.
- The important part here is when using the class template to declare the data type in in angle brackets when declaring the variable and instantiating with `new`.
- When creating a Stack of int,
```
Stack<int>* intStack = new Stack<int>();
```
- When creating a Stack of char,
```
Stack<char>* charStack = new Stack<char>();
```