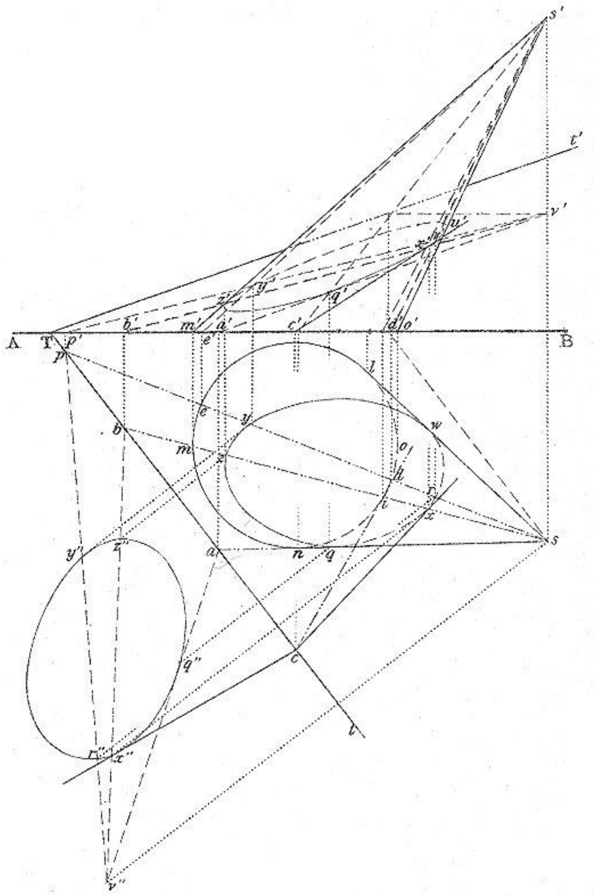

C++ Templates

Type-less Programming

The Problem

- “For many programmers, the largest single problem using C++, prior to the introduction of templates, was the lack of an extensive standard library. The major problem in producing such a library was that C++ did not provide a sufficiently general facility for defining “container classes”.”
  - Dr. Stroustrup

Scenario

- Many C++ programs use common data structures like stacks, queues and lists. One could easily implement a queue data structure. A queue of “customers” could be made, then, later implement a queue of “messages”.

- The program requirement grows, and now there is a need for a queue of orders. So just take the queue of messages and convert that to a queue of orders (*Copy, paste, find, replace*).

Make Changes?

- Errors are found in the original implementation. There are three queue versions to change now: Customer, message and order.

- Since the code has been duplicated in many places, the errors will have to found, fixed and re-tested.

- Re-inventing source code is not an intelligent approach in an object oriented environment which encourages re-usability.

Template Answer

- It makes more sense to implement a generic queue that can contain any arbitrary type rather than duplicating code.
- In C++ the answer is to use the concept of “type parameterization”, commonly known as templates.
- C++ provides two kinds of templates: class templates and function templates.

Class Templates

- A class template definition looks like a regular class definition, except it is prefixed by the keyword template. For example, here is the definition of a class template for a Stack:

```
template <typename T> class Stack
{
public:
            Stack(int = 10) ;
            ~Stack() { delete [] stackPtr ; }
            int push(const T&);
            int pop(T&) ;
            int isEmpty()const { return top == -1 ; }
            int isFull() const { return top == size - 1 ; }
private:
            int size ;  // number of elements on Stack.
            int top ;
            T* stackPtr ;
} ;
```

T is a type parameter and it can be any type.

Using a class template

- Using a class template is easy. Create the required classes by plugging in the actual type for the type parameters:

```
Stack<float> FloatStack;
Stack<int> IntStack;
float f = 1.1;
int I = 1;
FloatStack.push(f);
IntStack.push(i);
```

Function Templates

- To perform identical operations for each type of data, use function templates. You can write a single function template definition.

- Based on the argument types provided in calls to the function, the compiler automatically instantiates separate object code functions to handle each type of call appropriately.

A Template Function

- max returns the maximum of the two elements:

```
template <typename T>
T max(T a, T b)
{
        return a > b ? a : b ;
}
```

Using a template function

```
void main()
{
        cout << max(10, 15) << endl ;
        cout << max('k', 's') << endl ;
        cout << max(10.1, 15.2) << endl ;
}
```

Summary

- Templates are very useful when implementing generic class containers like vectors, stacks, lists, queues which can be used with any arbitrary type.
- Templates provide a way to re-use source code as opposed to inheritance and composition which provide a way to re-use object code.
- Template functions also provide a way to reuse code at the functional level.