# Class Templates and Function Templates

Sukhbir Tatla

sukhbir.tatla@senecacollege.ca

## TEMPLATES

- Type-independent patterns that can work with multiple data types.
  - Generic programming
  - Code reusable
- Class Templates
  - These define generic class patterns into which specific data types can be plugged in to produce new classes.
- Function Templates
  - These define logic behind the algorithms that work for multiple data types.

## Class Templates

- Class templates allow us to apply the concept of parametric polymorphism on a class-by-class basis.
- Their purpose is to isolate type dependencies for a particular class of objects.
- Using class templates, we shift the burden of creating duplicate classes to handle various combinations of data types to the compiler.

## Class Templates

- With class templates, we simply write one class and shift the burden of creating multiple instances of that class to the compiler.
- The compiler automatically generates as many instances of that class as is required to meet the client's demands without unnecessary duplication.
- The syntax for implementing class templates is similar to that of defining and declaring classes themselves. We simply preface the class definition or declaration with the keyword template followed by a parameterized list of type dependencies.

## Class Templates

- We begin with the keyword template followed by the class template's formal argument list within angle brackets (< >). This is followed by the class interface.
- The template formal argument list consists of a comma separated list containing the identifiers for each formal argument in the list. There must be at least one formal argument specified.
- The scope of the formal arguments is that of the class template itself.
- **Syntax:**

```
template <class TYPE, TYPE VALUE>
class Stack {
        //data members
        //member functions
};
```

## Example of a Class Template

```
template <class T>
class mypair {
     T values [2];
      public:
      mypair (T first, T second)
      {
          values[0]=first; values[1]=second;
       }
};
```

- The class that is just defined serves to store two elements of any valid type.

## Instantiating a Class Template

- Class template arguments *must* be explicit.
- The compiler generates distinct class types called template classes or generated classes.
- When instantiating a template, a compiler substitutes the template argument for the template parameter throughout the class template.

## Example of a Class Template

- For example, if we wanted to declare an object of this class to store two integer values of type int with the values 115 and 36 we would write:

```
mypair<int> myobject (115, 36);
```

- this same class would also be used to create an object to store any other type:

```
mypair<double> myfloats (3.0, 2.18);
```

## Example of a Class Template

```
template<class ItemType>
class GList
{
 public:
    bool IsEmpty() const;
    bool IsFull() const;
    int  Length() const;
    void Insert( ItemType item );
    void Delete( ItemType item );
    bool IsPresent( ItemType item ) const;
    void SelSort();
    void Print() const;
    GList();                          // Constructor
 private:
    int      length;
    ItemType data[MAX_LENGTH];
};
```

*Template parameter*

## Instantiating a Class Template

To create lists of different data types

template argument

```
// Client code

GList<int> list1;
GList<float> list2;
GList<string> list3;

list1.Insert(356);
list2.Insert(84.375);
list3.Insert("Muffler bolt");
```

Compiler generates 3 distinct class types

```
GList_int list1;
GList_float list2;
GList_string list3;
```

# Substitution Example

```
class GList_int
{
public:                                              int

void Insert( ItemType item );
                                                             int
    void Delete( ItemType item );

    bool IsPresent( ItemType item ) const;

private:                                               int
    int        length;
    ItemType  data[MAX_LENGTH];
};
                   int
```

## Function Definitions for Members of a Template Class

```
template<class ItemType>

void GList<ItemType>::Insert(  ItemType item )

{

    data[length] = item;

    length++;

}
```

```
//after substitution of float

void GList<float>::Insert(  float item )

{

    data[length] = item;

    length++;

}
```

## CAUTION

- Realize that when we write a class template we may not know the types that will be used by the client.

- Therefore, we recommend using type dependencies only once within the class's formal argument list.

- When we implement our own user defined types, realize the importance of overloading operators in a consistent fashion -- so that if template classes are used with those new data types, the operators used by the member functions will behave as expected <u>and</u> will not cause <u>syntax errors</u> when used…

## Function Templates

## Function and function templates

- C++ routines work on specific types. We often need to write different routines to perform the same operation on different data types.

```
int maximum(int a, int
  b, int c)
 {
     int max = a;
     if (b > max) max = b;
     if (c > max) max = c;
     return max;
}
```

```
float maximum(float a,
  float b, float c)
 {
     float max = a;
     if (b > max) max = b;
     if (c > max) max = c;
     return max;
}
```

## Function and function templates

```
double maximum(double a, double b, double c)
 {
     double max = a;
     if (b > max) max = b;
     if (c > max) max = c;
     return max;
}
```

- The logic is exactly the same, but the data type is different.

- Function templates allow the logic to be written once and used for all data types – generic function.

## FUNCTION TEMPLATES

- Generic function (function template) to find a maximum value: (see maximum example).

```
        Template <class T>
        T maximum(T a, T b, T c)        {
            T max = a;
            if (b > max) max = b;
            if (c > max) max = c;
            return max;                 }
```

- Template function itself is incomplete because the compiler will need to know the actual type to generate code. So template program are often placed in .h or .hpp files to be included in program that uses the function.

- C++ compiler will then generate the real function based on the use of the function template.

## Function Templates Usage

- After a function template is included (or defined), the function can be used by passing parameters of real types.

```
Template <class T>
T maximum(T a, T b, T c)
…
int i1, i2, i3;
…
int m = maximum(i1, i2, i3);
```

- maximum(i1, i2, i3) will invoke the template function with T==int. The function returns a value of int type.

## Function Templates Usage

- Each call to maximum() on a different data type forces the compiler to generate a different function using the template.
- See the maximum example.
  - One copy of code for many types.

```
int i1, i2, i3;
// invoke int version of maximum
cout << "The maximum integer value is: "
   << maximum( i1, i2, i3 );
// demonstrate maximum with double values
double d1, d2, d3;
// invoke double version of maximum
cout << "The maximum double value is: "
   << maximum( d1, d2, d3 );
```

## CAUTION

- Realize that when we write a function template we may not know the types that will be used by the client.
- Therefore, we recommend using type dependencies only once within the function's formal argument list.
- Also, make sure to handle the use of both built-in and pointer types.
- This may mean that we provide overloaded generalized function templates or template function specializations.

## THANK YOU!

Any Questions Please?