UNIVERSITY of LOUISVILLE
dare to be great

CPP Chapter 9:
Templates

CECS130
Introduction to Programming Languages
Dr. Roman V. Yampolskiy

Introduction to Templates

- It would be nice if we could write universal code capable of working with all types of data

- This is the idea behind **Templates**

Multiple max() Functions

```
int max (int x, int y)
{
  return (x > y) ? x : y;
}  // max
```

(a) Integer max

```
long max (long x, long y)
{
  return (x > y) ? x : y;
}  // max
```

(b) Long max

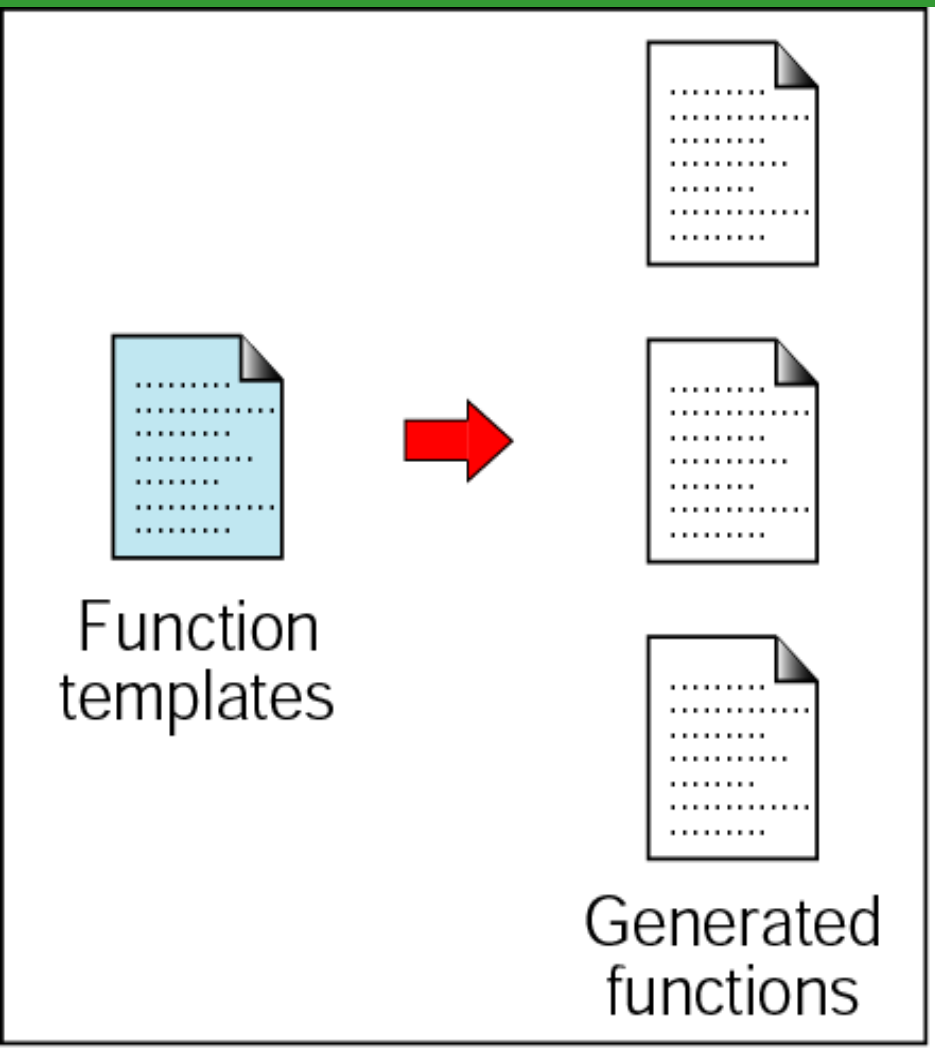
```
float max (float x, float y)
{
  return (x > y) ? x : y;
}  // max
```

(c) Float max

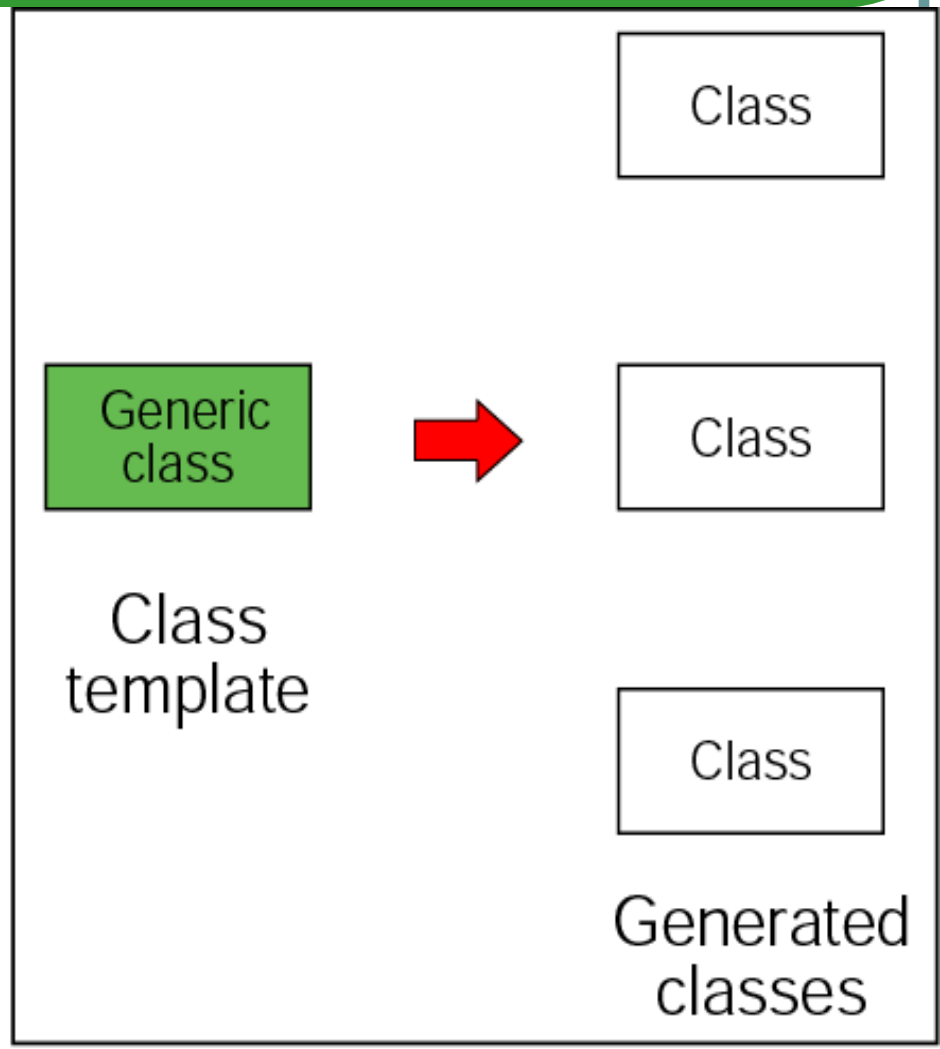
```
double max (double x, double y)
{
  return (x > y) ? x : y;
}  // max
```

(d) Double max

Basic Template Concepts



(a) Function template



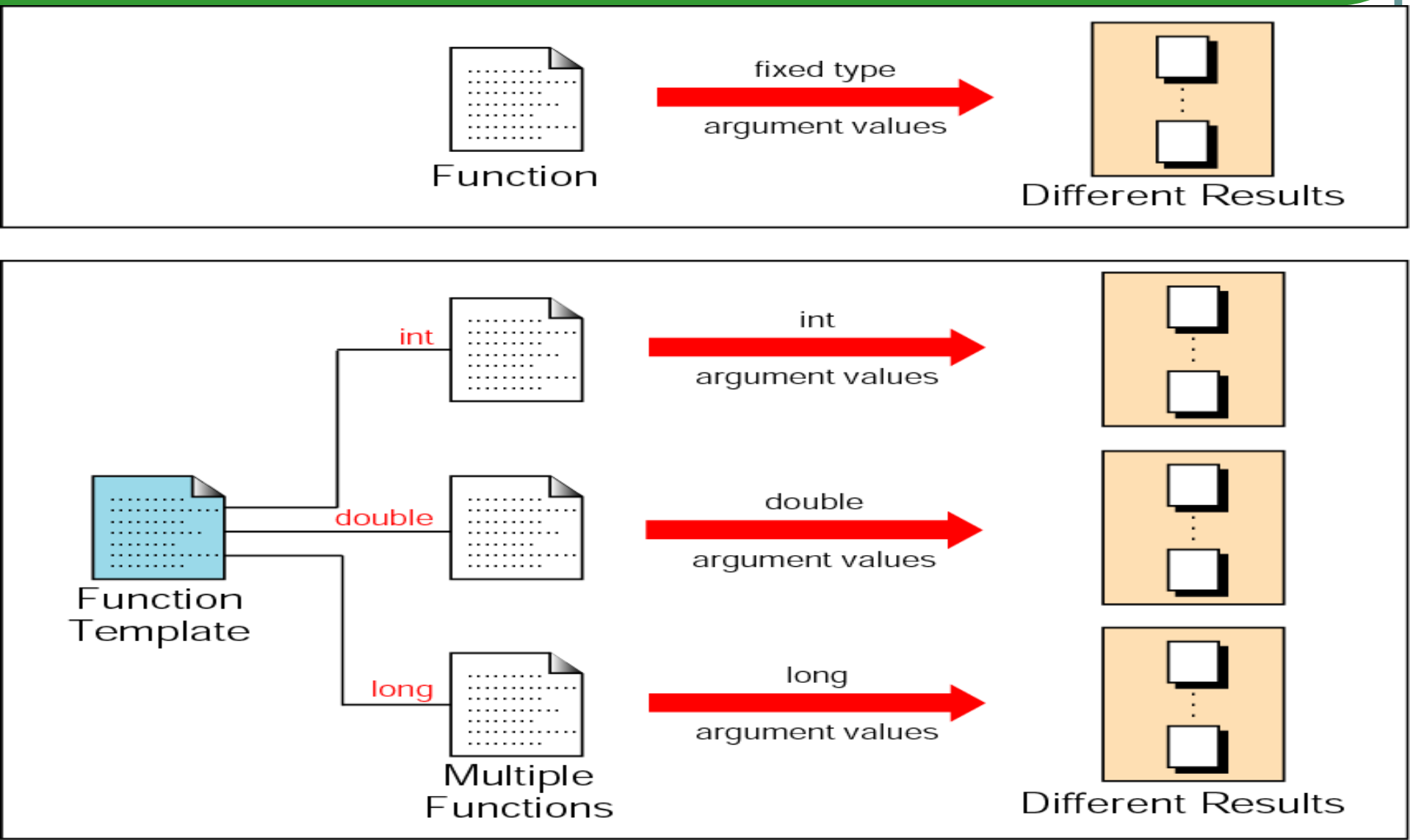
(b) Class template

Note:

A function template can create multiple functions, each with potentially different arguments and return types.

Function templates allow us to write a single function for a whole family of similar functions.

Function template operation



Generated functions

```
template <class TYPE>
TYPE max (TYPE x, TYPE y)
{
   return (x > y) ? x : y;
}  // max
```

Compiler

```
int max (int x, int y)
{
   return (x > y) ? x : y;
}  // max
```

```
long max (long x, long y)
{
   return (x > y) ? x : y;
}  // max
```

```
float max (float x, float y)
{
   return (x > y) ? x : y;
}  // max
```

```
double max (double x, double y)
{
   return (x > y) ? x : y;
}  // max
```

Templates

- Templates: a single code body for a set of related functions (called function template) and related classes (called class template)
- The syntax for templates is:

```
template <class Type>
    declaration
```

where `Type` is the type of the data and declaration is either a function declaration or a class declaration

Templates (continued)

- `template` is a reserved word
- The word `class` in the heading refers to any user-defined type or built-in type
- `Type` is called a formal parameter to the template
- Just as variables are parameters to functions
  - Data types are parameters to templates

Function Templates

- The syntax of the function template is:

```
template <class Type>
    function definition
```

  where `Type` is called a formal parameter of the template

- `Type`
  - Specifies type of parameters to the function
  - Specifies return type of the function
  - Declares variables within the function

Function Template Generation

```
template <class TYPE>
TYPE max (TYPE x, TYPE y)
{
   return (x > y) ? x : y;
}  // max
```

Generate

```
int max (int x, int y)
{
   return (x > y) ? x : y;
}  // max
```

Call

```
int num1;
int num2;
int result;
   .
   .
   .
result = max (num1, num2);
```

Example

- For example, to create a template function that returns the greater one of two objects we could use:

  ***template* <*class* myType>**
  myType GetMax (myType a, myType b)
  { *return* (a>b?a:b); }

  Here we have created a template function with myType as its template parameter. This template parameter represents a type that has not yet been specified, but that can be used in the template function as if it were a regular type. As you can see, the function template GetMax returns the greater of two parameters of this still-undefined type.

  To use this function template we use the following format for the function call:

  function_name <type> (parameters);

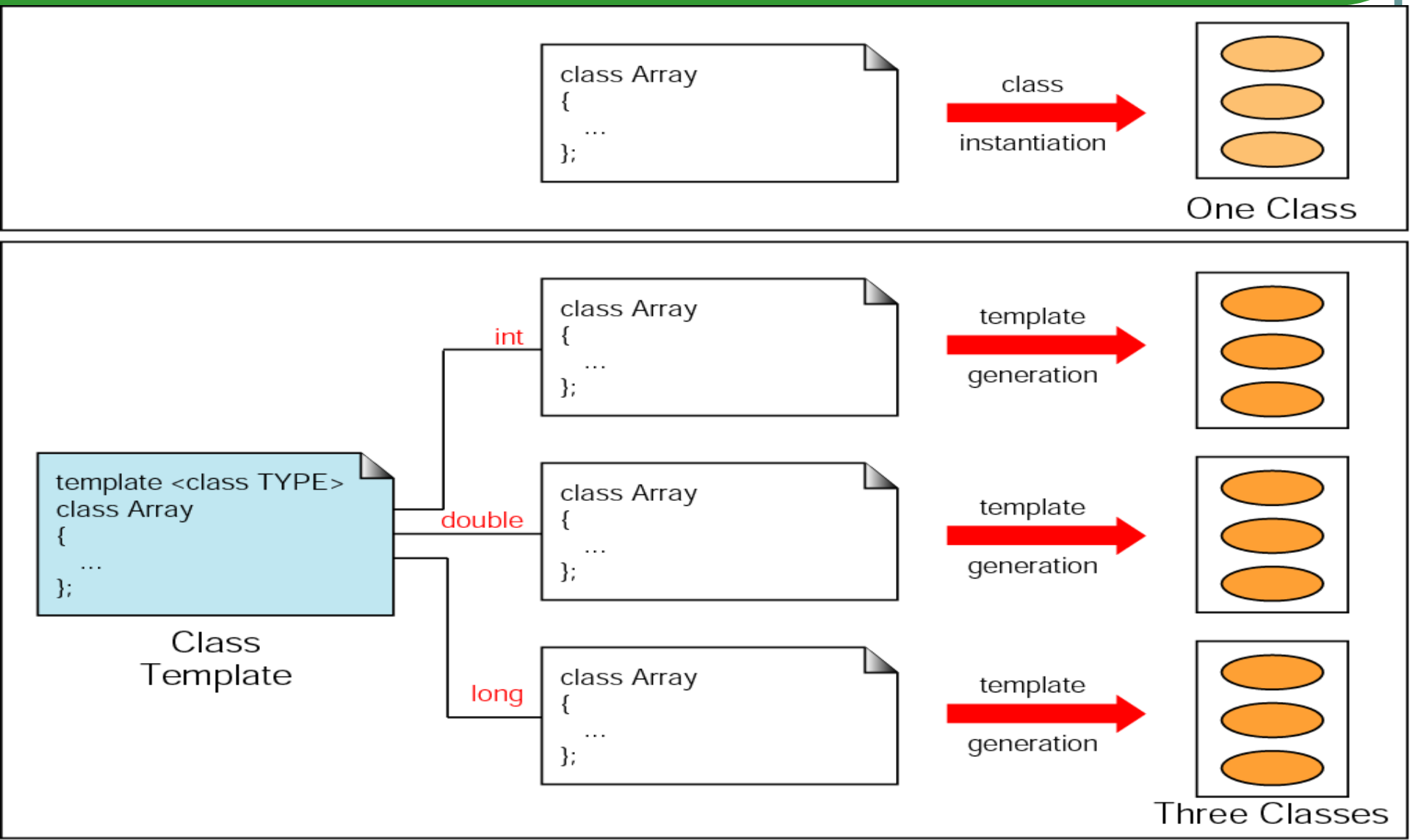
  For example, to call GetMax to compare two integer values of type int we can write:
  ***int* x,y;**
  GetMax <*int*> (x,y);

Generic Function

The generic **max()** function can be used to return a maximum of two values of *any type*, provided that

- The two values have the same type
- The two values can be compared using the > operator

Class template operation



Class Templates

- Class templates: a single code segment represents a set of related classes
- Syntax:

```
template <class Type>
class declaration
```

- Called parameterized types
  - A specific class is made based on the parameter type
- A template instantiation can be created with either a built-in or user-defined type

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

- 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

Instantiating a Class Template

To create lists of different data types

```
// Client code

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

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

template argument

Compiler generates 3 distinct class types

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

Substitution Example

```
class GList_int
{
public:

void Insert( /*   */ ItemType item );

    void Delete( /*    */ ItemType item );

    bool IsPresent( /*    */ ItemType item ) const;

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

int

int

int

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++;
}
```

Compile Issues

- Passing parameters to a function takes effect at run time
- Passing a parameter to a class template takes effect at compile time
- Normally, you put class declaration and class implementation into two separate files
- However, it is safer to put them together for class templates, because some compliers cannot compile them separately

<typename T> preferred

- You can use either **<typename T>** or **<class T>** to specify a type parameter

- Using **<typename T>** is better because **<typename T>** is descriptive

- **<class T>** could be confused with class declaration

Multiple Type Parameters

- Occasionally, a template function may have more than one parameter
- In this case, place the parameters together inside the brackets, separated by commas, such as:

<typename T1, typename T2, typename T3>

Developing a Generic Function

When you are writing a generic function, it is better to start with non-generic function, debug and test it, and then convert it to a generic function.

Default Type Arguments

C++ allows you to assign a default type for a type parameter in a class template. For example, you may assign **int** as a default type in the generic **Stack** class as follows:

```
template<typename T = int>
class Stack
{
  ...
};
```

Default Type Arguments

You can now declare an object using the default type like this:

Stack<> stack; // stack is a stack for int values

You can only use default type in class templates, not in function templates

nontype parameter

You can also use nontype parameters along with type parameters in a template prefix. For example, you may declare the array capacity as a parameter for the Stack class as follows:

```
template<typename T, int capacity>
class Stack {
 private:
  T elements[capacity];
 };
```

You can then use this class template as follows:

```
Stack<int, 18> MyStack;
```

Templates and Inheritance

- A non-template class can be derived from a class template specialization.
- A class template can be derived from a non-template class.
- A class template can be derived from a class template.

Standard Template Library

- Some algorithms do not depend on some particular implementation of a data structure but only on a few fundamental semantic properties of the structure
- STL was developed at HP labs in 1992
- Become part of the C++ Standard in 1994

What's in STL?

- Container classes: vector, list,set, map, etc…
- A large collection of algorithms, such as reverse, swap, heap, and etc.

Example: Vector

- A sequence that supports random access to elements
  - Elements can be inserted and removed at the beginning, the end and the middle
  - Constant time random access
  - Commonly used operations
    - begin(), end(), size(), push_back(…), pop_back(), insert(…), empty()

Example of vectors

```
// Instantiate a vector
vector<int> V;

// Insert elements
V.push_back(2);           // v[0] == 2
V.insert(V.begin(), 3); // V[0] == 3, V[1] == 2

// Random access
V[0] = 5;                 // V[0] == 5

// Test the size
int size = V.size();      // size == 2
```

Vector Class: Template

```
template <typename T>
class vector
{
private:
    T* vec_data;
    int length;
    int vec_size;
};
```

Vector Class: Specialization

```
template <>
class vector <bool>
{
  private:
    unsigned int *vector_data;
    int length;
    int size;
};
```

Overloading Function Templates: Example

```
#include <iostream.h>

template <class T>
T&  min(T  &tParam1, T  &tParam2) {
   if(tParam1 < tParam2)
      return tParam1;
   else
      return tParam2;
}

template <class T>
T&  min(T  &tParam1, T  &tParam2, T  &tParam3) {
   if(min(tParam1, tParam2) < tParam3)
      return min(tParam1, tParam2) ;
   else
      return tParam3;
}
```

Which function to call?

1. **Find all the versions** of the function that could possibly apply to the arguments.
2. If one function template is a specialization of another, **choose the specialization**.
3. Apply the **normal** overload **rules** for functions to everything that is left.
4. If a function and a function template are left, choose the function.
5. If no **possibilities remain**, the call was an **error**. If two or more possibilities remain, the call was **ambiguous** and is an error.

Why Use Templates?

- One C++ Class Template can **handle different types** of parameters
- Compiler **generates** classes **for only the used types**. If the template is instantiated for int type, compiler generates only an int version for the C++ template class
- Templates **reduce** the effort on **coding** for different data types to a single set of code
- **Testing** and debugging efforts are **reduced**

Why Not Use Templates?

- Many **compilers have poor support** for Templates which might result in less portable code
- **Unreadable error messages** when errors are detected in template code which can make templates difficult to develop
- Each use of a template may cause the compiler to generate **extra code** (an *instantiation* of the template)

The End!

Based on slides by Liang and Malik