

# Introduction to C++ Templates and Exceptions

- **C++ Function Templates**
- **C++ Class Templates**



## C++ Function Templates

- **Approaches for functions that implement identical tasks for different data types**
  - **Naïve Approach**
  - **Function Overloading**
  - **Function Template**
- **Instantiating a Function Templates**



## Approach 1: Naïve Approach

- create unique functions with unique names for each combination of data types
  - difficult to keeping track of multiple function names
  - lead to programming errors

Copyright 2006, The Ohio State University



## Example

```
void PrintInt( int n )
{
    cout << "***Debug" << endl;
    cout << "Value is " << n << endl;
}
void PrintChar( char ch )
{
    cout << "***Debug" << endl;
    cout << "Value is " << ch << endl;
}
void PrintFloat( float x )
{
    ...
}
void PrintDouble( double d )
{
    ...
}
```

To output the traced values, we insert:

```
PrintInt( sum );
PrintChar( initial );
PrintFloat( angle );
```



## Approach 2:Function Overloading (Review)

- The use of the same name for different C++ functions, distinguished from each other by their parameter lists
  - Eliminates need to come up with many different names for identical tasks.
  - Reduces the chance of unexpected results caused by using the wrong function name.

Copyright 2006, The Ohio State University



## Example of Function Overloading

```
void Print( int n )
{
    cout << "***Debug" << endl;
    cout << "Value is " << n << endl;
}
void Print( char ch )
{
    cout << "***Debug" << endl;
    cout << "Value is " << ch << endl;
}
void Print( float x )
{
}
```

To output the traced values, we insert:

```
Print(someInt);
Print(someChar);
Print(someFloat);
```

Copyright 2006, The Ohio State University



## Approach 3: Function Template

- A C++ language construct that allows the compiler to generate multiple versions of a function by allowing parameterized data types.

### FunctionTemplate

```
Template < TemplateParamList >  
FunctionDefinition
```

### TemplateParamDeclaration: placeholder

```
{  
class typeIdentifier  
typename variableIdentifier
```

Copyright 2006, The Ohio State University



## Example of a Function Template

```
template<class SomeType>  
void Print( SomeType val )  
{  
    cout << "****Debug" << endl;  
    cout << "Value is " << val << endl;  
}
```

*Template parameter*  
(class, user defined type, built-in types)

*Template argument*

To output the traced values, we insert:

```
Print<int>(sum);  
Print<char>(initial);  
Print<float>(angle);
```

Copyright 2006, The Ohio State University



# Instantiating a Function Template

- When the compiler instantiates a template, it substitutes the **template argument** for the **template parameter** throughout the function template.

## TemplateFunction Call

Function < TemplateArgList > (FunctionArgList)

# A more complex example

```
template<class T> void sort(vector<T>& v)
{
    const size_t n = v.size();

    for (int gap=n/2; 0<gap; gap/=2)
        for (int i=gap; i<n; i++)
            for (int j=i-gap; 0<j; j-=gap)
                if (v[j+gap]<v[j]) {
                    T temp = v[j];
                    v[j] = v[j+gap];
                    v[j+gap] = temp;
                }
}
```

# Summary of Three Approaches

## Naïve Approach

Different Function Definitions  
Different Function Names

## Function Overloading

Different Function Definitions  
Same Function Name

## Template Functions

One Function Definition (a function template)  
Compiler Generates Individual Functions

Copyright 2006, The Ohio State University



# Class Template

- A C++ language construct that allows the compiler to generate multiple versions of a class by allowing parameterized data types.

## Class Template

```
Template < TemplateParamList >  
ClassDefinition
```

## TemplateParamDeclaration: placeholder

```
{  
  class typeIdentifier  
  typename variableIdentifier
```

Copyright 2006, The Ohio State University



## Example of a Class Template

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

Template parameter

Copyright 2006, The Ohio State University



## 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.**

Copyright 2006, The Ohio State University



# 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;
```

Copyright 2006, The Ohio State University



# Substitution Example

```
class GList_int  
{  
public:  
void Insert( /* in */ ItemType item );  
void Delete( /* in */ ItemType item );  
bool IsPresent( /* in */ ItemType item ) const;  
private:  
int length;  
ItemType data[MAX_LENGTH];  
};
```

*int*

*int*

*int*

*int*

Copyright 2006, The Ohio State University



## Function Definitions for Members of a Template Class

```
template<class ItemType>
void GList<ItemType>::Insert( /* in */ ItemType item )
{
    data[length] = item;
    length++;
}
```

```
//after substitution of float
void GList<float>::Insert( /* in */ float item )
{
    data[length] = item;
    length++;
}
```



## Another Template Example: passing two parameters

```
template <class T, int size>
class Stack {...
    T buf[size];
};
Stack<int,128> mystack;
```

non-type parameter



## Standard Template Library

- In the late 70s Alexander Stepanov first observed that some algorithms do not depend on some particular implementation of a data structure but only on a few fundamental semantic properties of the structure
- Developed by Stepanov and Lee at HP labs in 1992
- Become part of the C++ Standard in 1994

Copyright 2006, The Ohio State University



## What's in STL?

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

Copyright 2006, The Ohio State University



# 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()

Copyright 2006, The Ohio State University



## 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
```

Copyright 2006, The Ohio State University



## Take Home Message

- Templates are mechanisms for generating functions and classes on type parameters. We can design a single class or function that operates on data of many types
  - function templates
  - class templates