Introduction to C++ 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**

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

- **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.**

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);
```

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
```

Example of a Function Template


```
template<class SomeType>

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

Template parameter(can use typename)

(class, user defined type, built-in types)

Can also use

```
Print(12);//int
Print(1.78f);//float
Print("Hello");//char *
```

```
#include<iostream>
using namespace std;
template <class T>
void print(T val)
{
        cout<<"\n value is::"<<val;
}
int main()
{
        print<int>(10);
        print<char>('c');
        print<float>(10.302);
        return 0;
}
```

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
```

class a template exmple

```
#include<iostream>
using namespace std;
template <class T>
class array {
      T arr[20];
      public:
array() {
      for(int i=0;i<5;i++)
      arr[i]=0;
}
void read(){
      cout<<"\n enter elements...\n";
      for(int i=0;i<5;i++)
      cin>>arr[i];
}
void display(){
cout<<"\n entered elements are as follows...\n";
            for(int i=0;i<5;i++)
            cout<<arr[i]<<"\t";
      }
};
int main()
{
      class array<int> a;
      cout<<"\n now we are creating integer array";
      a.read();
      a.display();
      class array<char> b;
      cout<<"\n now we are creating char array";
      b.read();
      b.display();
      class array<double> c;
      cout<<"\n now we are creating float array";
      c.read();
      c.display();
      cout<<"\n";
      return 0;
}
```

Output:

```
now we are creating integer array
 enter elements...
1          2          3          4          5
 entered elements are as follows...
1       2       3       4       5
 now we are creating char array
 enter elements...
a          b          c          d          e
 entered elements are as follows...
a       b       c       d       e
 now we are creating float array
 enter elements...
1.32       5.656      7.65       89.655    343.566
 entered elements are as follows...
1.32    5.656   7.65    89.655  343.566
```

Overloading a template function by another template function


```
template <class  t>
void somefunction(t a1)
{
}

template<class t, class u>
void somefunction(t a1, u b1)
{
}
```

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

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

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

template argument

Compiler generates 3 distinct class types

```
List_int list1;
List_float list2;
List_string list3;
```

Substitution Example

```
class List_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

Function Definitions for Members of a Template Class

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

```
//after substitution of float
void List<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