girmiti software™

Introduction to C++

girmiti software™

- What is Template
- Function Template
- Function Template instances
- Function Template with multiple template type parameters
- Overloaded Function Template
- Class Template
- Template & Inheritance

Template

Templates:

Templates are the foundation of generic programming, which involves writing code in a way that is independent of any particular type.
A template is a blueprint or formula for creating a generic class or a function. The library containers like iterators and algorithms are examples of generic programming and have been developed using template concept.

Templates can be categorized into 2 categories:

- Function Template
- Class Template

girmiti software™

Template

Function templates:

In C++, function templates are functions that serve as a pattern for creating other similar functions. The basic idea behind function templates is to create a function without having to specify the exact type(s) of some or all of the variables. Instead, we define the function using placeholder types, called template type parameters. Once we have created a function using these placeholder types, we have effectively created a "function stencil".

We can't call a function template directly , this is because the compiler doesn't know how to handle placeholder types directly. Instead, when we call a template function, the compiler "stencils" out a copy of the template, replacing the placeholder types with the actual variable types in function call! Using this methodology, the compiler can create multiple "flavors" of a function from one template.

The general form of a template function definition is shown here:

```
template <class type> ret-type func-name(parameter list)
{
 // body of function
}
```

Here, type is a placeholder name for a data type used by the function. This name can be used within the function definition.

Template

Advantages of function templates:

- Template functions can save a lot of time, because we only need to write one function, and it will work with many different types.
- Template functions reduce code maintenance, because duplicate code is reduced significantly.
- Template functions can be safer, because there is no need to copy functions and change types by hand whenever we need the function to work with a new type.

Template

Example of Function Templates:

```
#include <iostream>
using namespace std;
template <typename T>
T Max (T a, T b)
{
    return a < b ? b:a;
}
int main ()
{
    int i = 39;
    int j = 20;
    cout << "Max(i, j): " << Max(i, j) << endl;
    double f1 = 13.5;
    double f2 = 20.7;
    cout << "Max(f1, f2): " << Max(f1, f2) << endl;
    string s1 = "Hello";
    string s2 = "World";
    cout << "Max(s1, s2): " << Max(s1, s2) << endl;
    return 0;
}
```

girmiti software™

Template

Output:

```
Max(i, j): 39
Max(f1, f2): 20.7
Max(s1, s2): World
```

Template

Function template instances:

As we know that C++ does not compile the template function directly. Instead, at compile time, when the compiler encounters a call to a template function, it replicates the template function and replaces the template type parameters with actual types.The function with actual types is called a function template instance.

Suppose we have a templated function:

```
template <typename Type>  // this is the template parameter declaration
Type max(Type tX, Type tY)
{
    return (tX > tY) ? tX : tY;
}
```

When compiling our program, the compiler encounters a call to the templated function:

```
int nValue = max(3, 7);    // calls max(int, int)
```

The compiler then replicates the function template and creates the template instance .

```
int max(int tX, int tY)
{
    return (tX > tY) ? tX : tY;
}
```

This is now a “normal function” that can be compiled into machine language.
It’s also worth noting that if we create a template function but do not call it, no template instances will be created.

Template

Function Template with multiple template type parameters:

We can define the function using multiple template type parameters.

The general form of a template function definition with multiple type parameters is shown here:

```
template <class type1,class  type2>
ret-type func-name(type1 a,type2 b)
{
 // body of function
}
```

Template

Example:

```
#include<iostream>
using namespace std;
template<class A,class B>
A larger(A a1,B b1)
{
		return (a1>b1)?a1:b1;
}
int main()
{
		int x=10;
		double y=20.5;
		cout<<larger(x,y)<<endl;
		float z=22.5f;
		long m=99999;
		cout<<larger(z,m);
}
```

Template

Overloaded Function Templates:

We use templates when we need functions that apply the same algorithm to a variety of types. It might be, however, that not all types would use the same algorithm.
To handle this possibility, we can overload template definitions, just as we overload regular function definitions. As with ordinary overloading, overloaded templates need distinct function signatures.

We may overload a function template either by a non-template function or by another function template.
If we call the overloaded function ,non-template functions take precedence over template functions.

girmiti software™

Template

```
#include<iostream>
using namespace std;

template<class T>
class Sample
{
        public:
        void display(T a);
        void display(T a,T b);
        void display(T a,T b,T c);
};
template<class T>
void Sample<T>::display(T a)
{       cout<<"a="<<a<<endl;
}
template<class T>
void Sample<T>::display(T a,T b)
{
        cout<<"a="<<a<<"b="<<b<<endl;
}
```

Template

```
template<class T>
void Sample<T>::display(T a,T b,T c)
{
        cout<<"a="<<a<<"b="<<b<<"c="<<c<<endl;
}
int main()
{
        Sample<int> obj1;
        obj1.display(10);
        obj1.display(10,20);
        obj1.display(10,20,30);
        Sample<double> obj2;
        obj2.display(1.1);
        obj2.display(1.1,2.2);
        obj2.display(1.1,2.2,3.3);
}
```

Template

Class Template:

As we can define function templates, we can also define class templates.

A class template is a class definition that describes a family of related classes. C++ offers the user the ability to create a class that contains one or more data types that are generic or parameterized.

The general form of a generic class declaration is shown here:

```
template <class type> class class-name
{
 . . .
}
```

Here, type is the placeholder type name, which will be specified when a class is instantiated. We can define more than one generic data type by using a comma-separated list.
Once the class templete has been defined,it is required to instantiate a class object using a specific primitive or user defined type to replace the parameterised types.

Template

```
#include<iostream>
using namespace std;
template<class T>
class sample
{ public:
  T val1,val2,val3;
  public:
           void getData();
           void sum();
};
template<class T>
void sample<T>::getData()
{
  cin>>val1>>val2;
}
template<class T>
void sample<T>::sum()
{
  T value;
  value=val1+val2;
  cout<<"sum is="<<value<<endl;
}
```

girmiti software™

Template

```
int main()
{
  sample<int> obj1;
  sample<float> obj2;
  sample<string> obj3;
  cout<<"enter 2 integer values";
  obj1.getData();
  obj1.sum();
  cout<<"enter 2 float values";
  obj2.getData();
  obj2.sum();
  cout<<"enter 2 string values";
  obj3.getData();
  obj3.sum();
}
```

Template

Template & Inheritance:

Templates may be used in many combinations with inheritance. It is possible to combine inheritance and templates because a template class is a class,with a parameter.Four combinations of templates and inheritance are possible:

- a template class that inherits from another template class
- a non-template class that inherits from a template class
- a template class that inherits from a non-template class

Template

Inheritance between Template Classes:

In this case, both the base and derived classes are templates. In the usual way, inheritance is used to extend the base-class template through the addition of new methods and/or data in the derived-class template. A common situation in this case is that the template parameter of the derived class is also used as the template parameter of the base class.

Template

Example:

```
#include<iostream>
using namespace std;
template<class T>
class sample
{           public:
            T val1,val2,val3;
            public:
                        void getData();
                        void display();
};
template<class T>
class sample1:public sample<T>
{
            public:
                        void sum();
};
template<class T>
void sample<T>::display()
{
            cout<<val1<<" "<<val2<<endl;
}
```

Template

```
template<class T>
void sample<T>::getData()
{
        cin>>val1>>val2;
}
template<class T>
void sample1<T>::sum()
{
        T value;
        value=this->val1+this->val2;
        cout<<"sum is="<<value<<endl;
}
int main()
{
        sample1<int> obj1;
        sample1<float> obj2;
        sample1<string> obj3;
        cout<<"enter 2 integer values";
        obj1.getData();
        obj1.display();
        obj1.sum();
```

Template

```
        cout<<"enter 2 float values";
         obj2.getData();
         obj2.display();
         obj2.sum();
         cout<<"enter 2 string values";
         obj3.getData();
         obj3.display();
         obj3.sum();
}
```

Template

Inheritance from a Template Class:

A non-template derived class can inherit from a template base class if the template's parameter is fixed by the nature and definition of the derived class. In this case the template parameter is determined by the designer of the derived class.

Template

Example:

```
#include<iostream>
using namespace std;
template<class T>
class sample
{           public:
            T val1,val2,val3;
            public:
                        void getData();
                        void sum();
};
class sample1:public sample<int>
{
            public:
                        void display();
};
void sample1::display()
{
            cout<<val1<<" "<<val2<<endl;
}
template<class T>
```

Template

```
void sample<T>::getData()
{
        cout<<"Enter val1 & val2";
        cin>>val1>>val2;
}
template<class T>
void sample<T>::sum()
{
        T value;
        value=val1+val2;
        cout<<"sum is="<<value<<endl;
}
int main()
{
        sample1 obj1;
        obj1.getData();
        obj1.display();
        obj1.sum();
}
```

Template

Inheritance by a template class:

In this case, the derived class is a template class, but the base class is a non-template class. Under these circumstances, the template parameter only has meaning to the derived class, not to the base class

Template

Example:

```
#include<iostream>
using namespace std;
class sample
{           public:
            void display()
            {
                        cout<<"Common functionality"<<endl;
            }
};
template<class T>
class sample1:public sample
{
            T val1,val2;
            public:
                        void getData();
                        void sum();
};
template<class T>
void sample1<T>::getData()
{
```

Template

```
		cout<<"Enter val1 & val2";
		cin>>val1>>val2;
}
template<class T>
void sample1<T>::sum()
{
		T value;
		value=val1+val2;
		cout<<"sum is="<<value<<endl;
}
int main()
{		sample1<int> obj1;
		sample1<float> obj2;
		cout<<"enter 2 integer values"<<endl;
		obj1.display();
		obj1.getData();
		obj1.sum();
		cout<<"enter 2 float values"<<endl;
		obj2.display();
		obj2.getData();
		obj2.sum();
}
```

girmiti software™

Discussions