Templates in C++

Templates

- Templates in C++ programming allows function or class to work on more than one data type at once without writing different codes for different data types.
- Using templates, it is possible to create generic functions and classes.
- In a generic function or class, the type of data upon which the function or class operates is specified as a parameter.
- Thus, you can use one function or class with several different types of data without having to explicitly recode specific versions for each data type.

Generic Functions

- A generic function defines a general set of operations that will be applied to various types of data. The type of data that the function will operate upon is passed to it as a parameter.

- Through a generic function, a single general procedure can be applied to a wide range of data.

- The Quicksort sorting algorithm is the same whether it is applied to an array of integers or an array of floats. It is just that the type of the data being sorted is different.

- By creating a generic function, you can define the nature of the algorithm, independent of any data.

- A generic function is created using the keyword template.

- template <class Ttype> ret-type funcname(parameter list)
- {
- // body of function
- }

- Ttype is a placeholder name for a data type used by the function. This name may be used within the function definition.

- However, it is only a placeholder that the compiler will automatically replace with an actual data type when it creates a specific version of the function. Although the use of the keyword class to specify a generic typein a template declaration is traditional, you may also use the keyword typename.

Function Template Example

```
#include <iostream>
using namespace std;
template<class T> T add(T &a,T &b)
{
   T result = a+b;
   return result;
}
int main()
{
 int i =2;
 int j =3;
 float m = 2.3;
 float n = 1.2;
 cout<<"Addition of i and j is :"<<add(i,j);

 cout<<'\n';
 cout<<"Addition of m and n is :"<<add(m
,n);
 return 0;
}
```

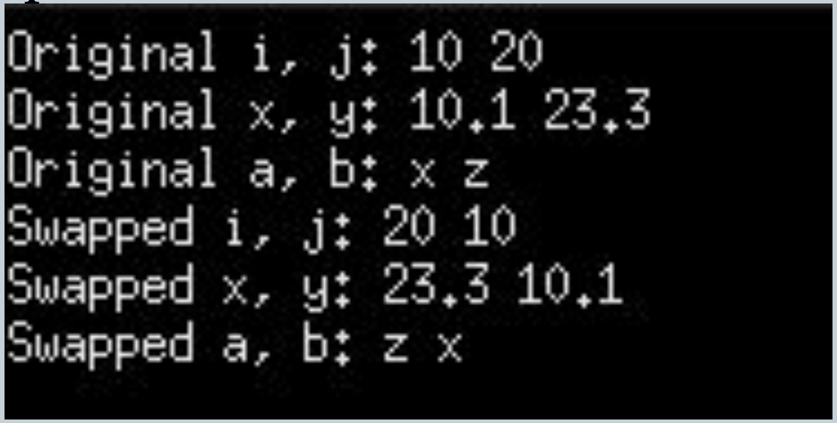
Output:

```
 Addition of i and j is :5
 Addition of m and n is :3.5
```

Function template example.

```
  #include <iostream>
using namespace std;
// This is a function template.
template <class X> void swapargs(X &a, X &b)
{
X temp;
temp = a;
a = b;
b = temp;
}
int main()
{
int i=10, j=20;
double x=10.1, y=23.3;
char a='x', b='z';
cout << "Original i, j: " << i << ' ' << j << '\n';
cout << "Original x, y: " << x << ' ' << y << '\n';
cout << "Original a, b: " << a << ' ' << b << '\n';
swapargs(i, j); // swap integers
swapargs(x, y); // swap floats
swapargs(a, b); // swap chars
cout << "Swapped i, j: " << i << ' ' << j << '\n';
cout << "Swapped x, y: " << x << ' ' << y <<
'\n';
cout << "Swapped a, b: " << a << ' ' << b <<
'\n';
return 0;
}
```

O/
P

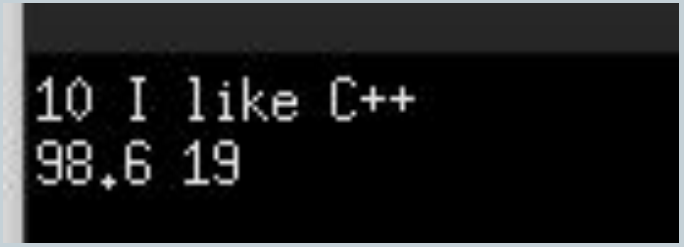


```
Original i, j: 10 20
Original x, y: 10.1 23.3
Original a, b: x z
Swapped i, j: 20 10
Swapped x, y: 23.3 10.1
Swapped a, b: z x
```

A Function with Two Generic Types

```
#include <iostream>
using namespace std;
template <class type1, class type2>void myfunc(type1 x, type2 y)
{
cout << x << ' ' << y << '\n';
}
int main()
{
myfunc(10, "I like C++");
myfunc(98.6, 19L);
return 0;
}
```

O/
P



```
10 I like C++
98.6 19
```

Explicitly Overloading a Generic Function

- If you overload a generic function, that overloaded function overrides (or "hides") the generic function relative to that specific version

```
// Overriding a template function.
#include <iostream>
using namespace std;
template <class X> void swapargs(X &a, X &b)
{
X temp;
temp = a;
a = b;
b = temp;
cout << "Inside template swapargs.\n";
}
```

```
// This overrides the generic version
 of swapargs() for ints.
void swapargs(int &a, int &b)
{
int temp;
temp = a;
a = b;
b = temp;
cout << "Inside swapargs int
specialization.\n";
}
```

```
int main()
{
int i=10, j=20;
double x=10.1, y=23.3;
char a='x', b='z';
cout << "Original i, j: " << i << ' ' << j <<
'\n';
cout << "Original x, y: " << x << ' ' << y <<
'\n';
cout << "Original a, b: " << a << ' ' << b <<
'\n';
swapargs(i, j); // calls explicitly overloaded
swapargs()
swapargs(x, y); // calls generic swapargs()
swapargs(a, b); // calls generic swapargs()
cout << "Swapped i, j: " << i << ' ' << j <<
'\n';
cout << "Swapped x, y: " << x << ' ' << y
<< '\n';
cout << "Swapped a, b: " << a << ' ' << b
<< '\n';
return 0;
}
```

O/P

```
Original i, j: 10 20
Original x, y: 10.1 23.3
Original a, b: x z
Inside swapargs int specialization.
Inside template swapargs.
Inside template swapargs.
Swapped i, j: 20 10
Swapped x, y: 23.3 10.1
Swapped a, b: z x
```

Overloading a Function Template

- In addition to creating explicit, overloaded versions of a generic function, you can also overload the template specification itself

```
// Overload a function template
declaration.
#include <iostream>
using namespace std;
// First version of f() template.
template <class X> void f(X a)
{
cout << "Inside f(X a)\n";
}
// Second version of f() template.
template <class X, class Y> void f(X a, Y b)
{
cout << "Inside f(X a, Y b)\n";
}
```

```
int main()
{
f(10);
// calls f(X)
f(10, 20); // calls f(X, Y)
return 0;
}
```

Template Function Restrictions

- Generic functions are similar to overloaded functions except that they are more restrictive.
- When functions are overloaded, you may have different actions performed within the body of each function.

- But a generic function must perform the same general action for all versions—only the type of data can differ.

Consider the overloaded functions in the following example program.

```
#include <iostream>
#include <cmath>
using namespace std;
void myfunc(int i)
{
cout << "value is: " << i << "\n";
}
void myfunc(double d)
{
double intpart;
double fracpart;
fracpart = modf(d, &intpart);
cout << "Fractional part: " << fracpart;
cout << "\n";
cout << "Integer part: " << intpart;
}
```

```
int main()
{
myfunc(1);
myfunc(12.2);
return 0;
}
```

These functions could not be replaced by a generic function because they do not do the same thing.

Template Classes

- Like function templates, you can also create class templates for generic class operations.
- Sometimes, you need a class implementation that is same for all classes, only the data types used are different.
- Normally, you would need to create a different class for each data type OR create different member variables and functions within a single class.
- This will unnecessarily bloat your code base and will be hard to maintain, as a change is one class/function should be performed on all classes/functions.
- However, class templates make it easy to reuse the same code for all data types.

How to declare a class template?

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

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

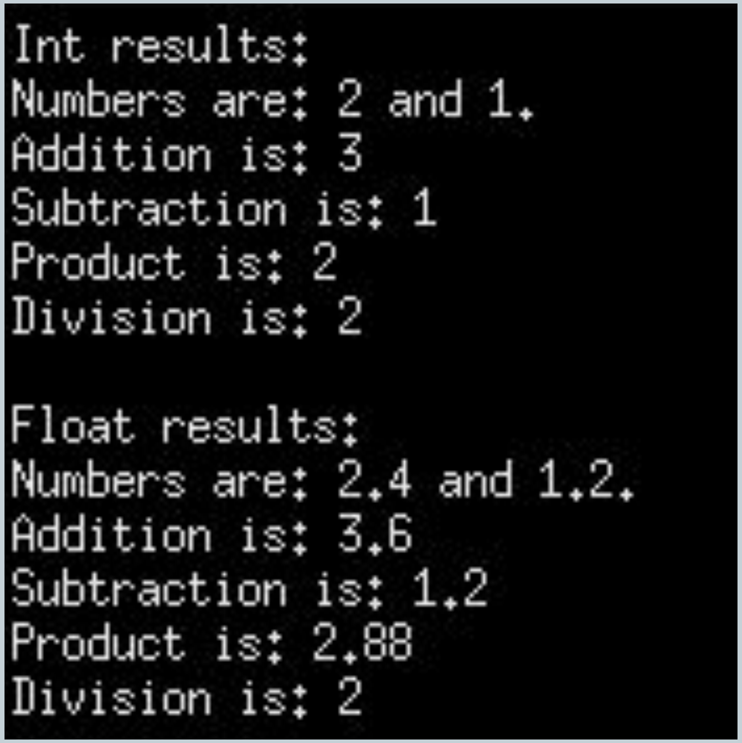
- Here, Ttype is the placeholder type name, which will be specified when a class is instantiated.
- If necessary, you can define more than one generic data type using a comma-separated list.

How to create a class template object?

- Once you have created a generic class, you create a specific instance of that classusing the following general form:
- lass-name <type> ob;
- Here, type is the type name of the data that the class will be operating upon.
- Member functions of a generic class are themselves automatically generic.

```
#include <iostream>
using namespace std;
template <class T>
class Calculator
{
private:
        T num1, num2;
public:
        Calculator(T n1, T n2)
        {
                num1 = n1;
                num2 = n2;
        }
void displayResult()
        {
cout << "Numbers are: " << num1 << " and " << num2 << 
"." << endl;
cout << "Addition is: " << add() << endl;
cout << "Subtraction is: " << subtract() << endl;
cout << "Product is: " << multiply() << endl;
cout << "Division is: " << divide() << endl;
        }
        T add() { return num1 + num2; }

        T subtract() { return num1 - num2; }

        T multiply() { return num1 * num2; }

        T divide() { return num1 / num2; }
};
```

```
int main()
{
	Calculator<int> intCalc(2, 1);
	Calculator<float> floatCalc(2.4, 1.2);

	cout << "Int results:" << endl;
	intCalc.displayResult();

	cout << endl << "Float results:" <<
endl;
	floatCalc.displayResult();

	return 0;
}
```



```
Int results:
Numbers are: 2 and 1.
Addition is: 3
Subtraction is: 1
Product is: 2
Division is: 2

Float results:
Numbers are: 2.4 and 1.2.
Addition is: 3.6
Subtraction is: 1.2
Product is: 2.88
Division is: 2
```

CLASS TEMPLATE WITH MULTIPLE PARAMETERS

- We can use more than one generic data type in a class template, and each generic data type is separated by the comma.
- Syntax

```
template<class T1, class T2, ......>
class class_name
{
  // Body of the class.
}
```

```
#include <iostream>
    using namespace std;
    template<class T1, class T2>
   class A
   {
       T1 a;
       T2 b;
       public:
      A(T1 x,T2 y)
     {
        a = x;
        b = y;
      }
        void display()
       {
             std::cout << "Values of a and b are : " <
< a<<" ,"<<b<<std::endl;
        }
    };
```

```
int main()
  {
      A<int,float> d(5,6.5);
      d.display();
      return 0;
  }
```

Output:

```
Values of a and b are : 5,6.5
```

Nontype Template Arguments

- The template can contain multiple arguments, and we can also use the non-type arguments In addition to the type T argument, we can also use other types of arguments such as strings, function names, constant expression and built-in types.
- **template**<**class** T, **int** size>

  class array

  {

    T arr[size];   // automatic array initialization.
- };
- Arguments are specified when the objects of a class are created:
- array<**int**, 15> t1;   // array of 15 integers.
- array<**float**, 10> t2;   // array of 10 floats.
- array<**char**, 4> t3;   // array of 4 chars.

Program

```
#include <iostream>
using namespace std;
template<class T, int size>
class A
{
   public:
   T arr[size];
   void insert()
   {
      int i =1;
      for (int j=0;j<size;j++)
      {
         arr[j] = i;
         i++;
      }
   }
   void display()
   {
      for(int i=0;i<size;i++)
      {
         std::cout << arr[i] << " ";
      }
   }
};
int main()
{
   A<int,10> t1;
   t1.insert();
   t1.display();
   return 0;
}
```

Output:

```
1 2 3 4 5 6 7 8 9 10
```

Points to Remember

- C++ supports a powerful feature known as a template to implement the concept of generic programming.
- A template allows us to create a family of classes or family of functions to handle different data types.
- Template classes and functions eliminate the code duplication of different data types and thus makes the development easier and faster.
- Multiple parameters can be used in both class and function template.
- Template functions can also be overloaded.
- We can also use nontype arguments such as built-in or derived data types as template arguments.

References

- C++: The Complete Reference, 4th Edition Herbert Schildt
- https://www.programiz.com/cpp-programming/templates
- https://www.javatpoint.com/cpp-templates