

CS-1004 Object Oriented programming

Week 16

Instructor:

ATIYA

LECTURER

(COMPUTER SCIENCE DEPARTMENT)

NATIONAL UNIVERSITY- FAST (KHI CAMPUS)

EMAIL: ATIYA.JOKHIO@NU.EDU.PK

ROOM: (LECTURER ROOM#19, CS DEPT BASEMENT-2)

Generic Programming!

- **Generic programming** means that you are not writing source code that is compiled as-is but that you write "templates" of source codes that the compiler in the process of compilation transforms into source codes.
- Generic programming is basically the idea that your code should be as generic as possible.
- **Generic programming** is a style of computer programming in which algorithms are written in terms of to-be-specified-later types that are then instantiated when needed for specific types provided as parameters.

Generic Programming!

C++11	<ul style="list-style-type: none">• Functional
C++	<ul style="list-style-type: none">• Generic• Object-Oriented
C	<ul style="list-style-type: none">• Procedural• Structured

The goal of *generic programming* is to write code that is independent of the *data types*.

generic programming

- To generate short, simple code and to avoid duplication of code, C++ provides templates to define the same piece of code for multiple data types.
- With templates, programmers can define a family of functions or classes that can perform operations on different types of data.
- Templates comes under the category of meta-programming and auto code generation, where the generated code is not visible in general. Through templates, C++ supports generic programming.
- Generic programming is a type of programming where the programmer specifies a general code first. That code is instantiated based on the type of parameters that are passed later in the program or at execution.

Generics

- Generics can be implemented in C++ using **Templates**.

Templates!

C++ Templates

- Templates are powerful features of C++ which allows you to write generic programs. In simple terms, you can create a single function or a class to work with different data types using templates.
- the simple idea is to pass data type as a parameter so that we don't need to write the same code for different data types. For example, a software company may need `sort()` for different data types. Rather than writing and maintaining the multiple codes, we can write one `sort()` and pass data type as a parameter.

C++ Templates

- The concept of templates can be used in two different ways:
 - Function Templates
 - Class Templates

A function template starts with the keyword **template** followed by template parameter/s inside `< >` which is followed by function declaration.

```
template <class T>
```

```
T someFunction(T arg)
```

```
{
```

```
    ... ..
```

```
}
```


Generic Functions/Function Templates

You can write both statements in a single line:

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

T is a placeholder that the compiler will automatically replace with an actual data type

You can also use keyword **typename** instead of class in the above example.

```
template <typename T>
```

```
T someFunction(T arg)
```

```
{
```

```
    ... ..
```

```
}
```

How to call a function template?

We can call the template function a couple of ways. Firstly, we can call it by explicitly specifying the type like so

//Explicit type parametrizing.

```
int myint = 5;  
someFunction<int>(myint);  
double mydouble = 99.9;  
someFunction<double>(mydouble);
```

Or

```
someFunction<int>(5);  
someFunction<double>(99.9);
```

Implicit type parametrizing

However with template function the compiler can determine the parametrizing types when we don't provide them, hence we can also call `someFunction()` like so

```
int myint = 5;
```

```
someFunction() <>(myint);
```

Implicit type parametrizing

```
double mydouble = 99.9;
```

```
someFunction(mydouble);
```

The first call with empty angle brackets tells the compiler that we are calling a template function and the second call leaves it up to the compiler to infer.

Example

```
template <class X>
//template <typename T>
void SimplePrint (X a){
    cout << "Parameter is: " << a;
}
int main(){
    int i = 20; char c = 'M'; float f = 5.5;
    SimplePrint ( i );
    SimplePrint ( c );
    SimplePrint ( f );
}
```

Example

```
template <class T>
void swapargs(T &a, T &b)
{
    T 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';

    swapargs(i, j); // swap integers
    swapargs(x, y); // swap floats
    swapargs(a, b); // swap chars
}
```

Output

```
fist:20  Second:10  
fist:23.3  Second:10.1  
fist:z  Second:x  
  
-----  
Process exited after 0.9158 seconds with return value 0  
Press any key to continue . . .
```


Syntax

The line:

template <class X> void swapargs(X &a, X &b)

can also be written in two consecutive lines as:

***template <class X>
void swapargs(X &a, X &b) { \\ function body }***

Note: *But no other statement can occur between the two lines*

Function template with more than one type parameter

In above example you can only use single data type what if you want to use combination of data types

// 2 type parameters:

```
template<class T1, class T2>
```

```
void someFunc(T1 var1, T2 var2 )
```

```
{
```

```
// some code in here...
```

```
}
```

Function with Two Generic Types

You can define more than one generic data type in the template statement by using a comma-separated list

```
template <class T1, class T2>
void myfunc(T1 a, T2 b)
{
    cout << a << " & " << b << '\n';
}
```

Can you have unused type parameters?

No, you may not. If you declare a template parameter then you absolutely must use it inside of your function definition otherwise the compiler will complain. So, in the example above, you would have to use both T1 and T2, or you will get a compiler error.

Output?

```
#include <iostream>

using namespace std;

template <class T1, class T2>
void myfunc(T1 a){
    cout << a << '\n';}

int main () {
    myfunc(2);
    return 0;
}
```

Output?

Compiler (5) <input type="checkbox"/> Resources <input checked="" type="checkbox"/> Compile Log <input type="checkbox"/> Debug <input type="checkbox"/> Find Results <input type="checkbox"/> Console <input type="checkbox"/> Close				
Line	Col	File	Message	
		C:\SPRING 2024\Object Oriented Program...	In function 'int main()':	
47	9	C:\SPRING 2024\Object Oriented Programming\...	[Error] no matching function for call to 'myfunc(int)'	
44	6	C:\SPRING 2024\Object Oriented Programming\...	[Note] candidate: 'template<class T1, class T2> void myfunc(T1)'	
44	6	C:\SPRING 2024\Object Oriented Programming\...	[Note] template argument deduction/substitution failed:	
Line:		50 Col:	3 Sel:	0 Lines: 50 Length: 915 Insert Done parsing in 0 seconds

Output?

```
#include <iostream>

using namespace std;

template <class T1, class T2>
void myfunc(){
    }

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

Output?

Compiler (5)					Resources	Compile Log	Debug	Find Results	Console	Close
Line	Col	File	Message							
		C:\SPRING 2024\Object Oriented Program...	In function 'int main()':							
57	8	C:\SPRING 2024\Object Oriented Programming\...	[Error] no matching function for call to 'myfunc()'							
54	6	C:\SPRING 2024\Object Oriented Programming\...	[Note] candidate: 'template<class T1, class T2> void myfunc()'							
54	6	C:\SPRING 2024\Object Oriented Programming\...	[Note] template argument deduction/substitution failed:							
Line:		57 Col:	8 Sel:	0 Lines:	60 Length:	1053	Insert	Done parsing in 0 seconds		

Output?

```
#include <iostream>
using namespace std;
template <class T1, class T2>
void myfunc(T1 a, T1 b)
{
    cout << a << " & " << b << '\n';
}

int main () {
myfunc(3,4.5);
return 0;
}
```

Output?

Compiler (5)				Resources	Compile Log	Debug	Find Results	Console	Close
Line	Col	File	Message						
		C:\SPRING 2024\Object Oriented Program...	In function 'int main()':						
71	13	C:\SPRING 2024\Object Oriented Programming\...	[Error] no matching function for call to 'myfunc(int, double)'						
65	6	C:\SPRING 2024\Object Oriented Programming\...	[Note] candidate: 'template<class T1, class T2> void myfunc(T1, T1)'						
65	6	C:\SPRING 2024\Object Oriented Programming\...	[Note] template argument deduction/substitution failed:						
Line:		71 Col:	13 Sel:	0 Lines:	74 Length:	1252	Insert		Done parsing in 0 seconds

Output?

```
#include <iostream>
using namespace std;
template <class T1, class T2>
void myfunc(T1 a, T2 b)
{
    cout << a << " & " << b << '\n';
}

int main () {
myfunc(3,4.1);
return 0;
}
```

Output?

```
3 & 4.1
```

```
-----
```

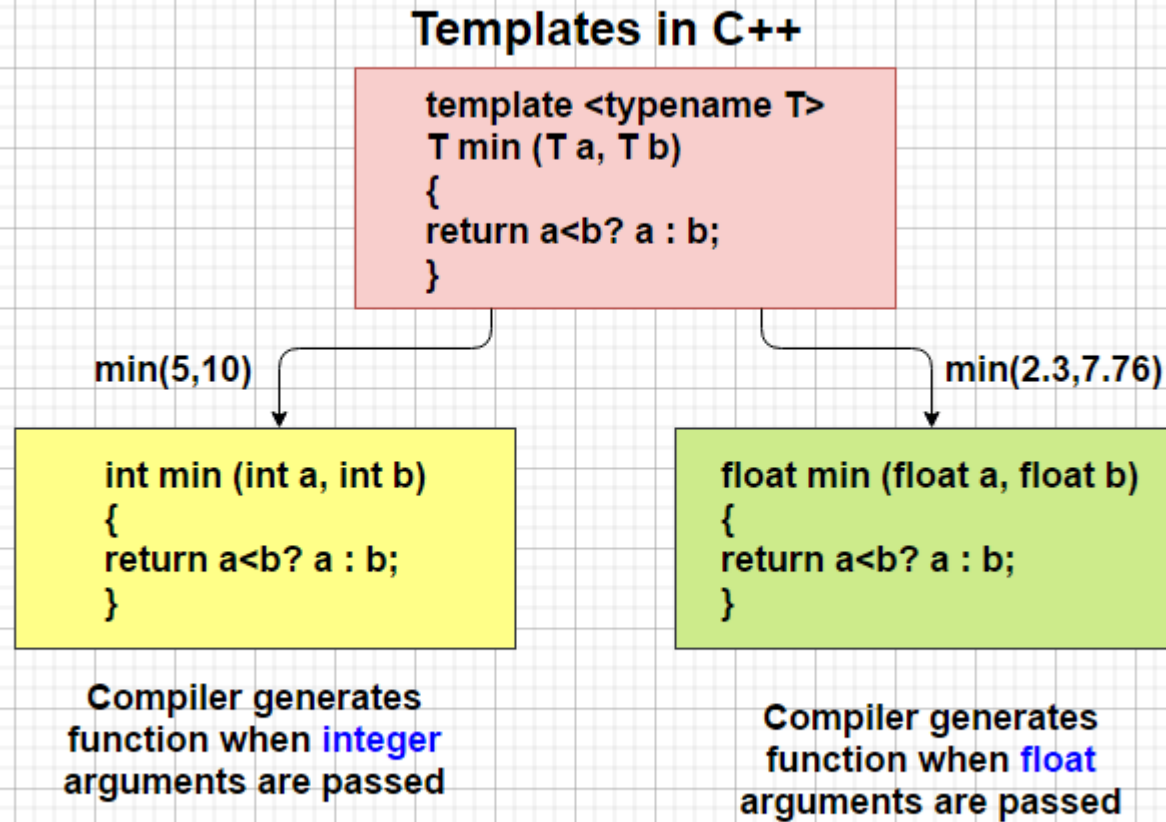
```
Process exited after 10.06 seconds with return value 0
```

```
Press any key to continue . . .
```

How templates work?

Templates are expanded at compiler time. This is like macros. The difference is, compiler does type checking before template expansion. The idea is simple, source code contains only function/class, but compiled code may contain multiple copies of same function/class.

How templates work?



```

// template function
template <class T>
T Large(T n1, T n2)
{
    return (n1 > n2) ? n1 : n2;
}

int main()
{
    int i1, i2;
    float f1, f2;
    char c1, c2;

    cout << "Enter two integers:\n";
    cin >> i1 >> i2;
    cout << Large(i1, i2) << " is larger." << endl;

    cout << "\nEnter two floating-point numbers:\n";
    cin >> f1 >> f2;
    cout << Large(f1, f2) << " is larger." << endl;

    cout << "\nEnter two characters:\n";
    cin >> c1 >> c2;
    cout << Large(c1, c2) << " has larger ASCII value.";

    return 0;
}

```

Enter two integers:

5

6

6 is larger.

Enter two floating-point numbers:

2.5

33.2

33.2 is larger.

Enter two characters:

x

a

x has larger ASCII value.

Process exited after 17.94 seconds with return value 0

Press any key to continue . . .

Overloading a Generic Function

- In addition to creating explicit, overloaded versions of a generic function, you can also overload the template specification itself
- To do so, simply create another version of the template that differs from any others in its parameter list

Overloading a Generic Function

Example

```
#include <iostream>
using namespace std;
// First version of f() template

template <class X>
void f(X a)
{
    cout << "Inside f(X a)";
    cout<<endl;
}
// Second version of f() template
template <class X>
void f(X a, X b)
{
    cout << "Inside f(X a, Y b)";
}
int main()
{
    f(10);           // calls f(X)
    f(10, 20);       // calls f(X, Y)
}
```

Overloading a Generic Function

Output:

```
Inside f(X a)
Inside f(X a, Y b)
-----
Process exited after 0.86 seconds with return value 0
Press any key to continue . . .
```

Using Normal Parameters in Generic Functions

- You can mix *non-generic parameters* with *generic parameters* in a template function:

```
template<class X> void func(X a, int b)
{
    cout << "General Data: " << a;
    cout << "Integer Data: " << b;
}
```

Use of Generic Functions

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

Common Applications

Sorting

Compacting an array

Searching

Generic Classes

- In addition to generic functions, you can also define a *generic class*
- The actual type of the data being used (in class) will be specified as a parameter when objects of that class are created
- Generic classes are useful when a class uses logic that can be generalized e.g. Stacks, Queues

Class Templates-Generic Class

- 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 in 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?

```
template <class T>
class className
{
    ... ..
public:
    T var;
    T someOperation(T arg);
    ... ..
};
```

In the above declaration, T is the template argument which is a placeholder for the data type used.

Inside the class body, a member variable var and a member function someOperation() are both of type T.

How to create a class template object?

```
className<dataType> classObject;
```

Example

```
className<int> classObject;  
className<float> classObject;  
className<string> classObject;
```

Generic Classes

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

```
template <class T> class class-name
```

```
{
```

```
    ...
```

```
}
```

Example

```
#include <iostream>
using namespace std;
template <class T1, class T2> class myclass
{
    T1 i;
    T2 j;
public:
    myclass (T1 a, T2 b) { i = a; j = b; }
    void show( ) { cout << i << " & " << j; }
};

int main()
{
    myclass<int, double>
    ob1(10, 0.23);
    myclass<char, char *>
    ob2('X', "Hello");

    ob1.show(); // show int, double
    cout<<endl;
    ob2.show(); // show char, char *
}
```

```
10 & 0.23
X & Hello
```

```
-----
Process exited after 2.823 seconds with return va
Press any key to continue . . .
```

Example

```
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
```

```
-----
Process exited after 0.9359 seconds with return value 0
Press any key to continue . . .
```

Using Non-Type Arguments with Generic Classes

In a generic class, we can also specify non-type arguments:

```
template <class T, int size> class MyClass
{
    T arr[size]; // length of array is passed in size
    // rest of the code in class
}
```

Example (cont.)

```
int main()
{
    atype<int, 10> intob;
    atype<double, 15> doubleob;
}
```

Using Non-Type Arguments with Generic Classes

- Non-type parameters can only be of type integers, pointers, or references
- The arguments that you pass to a non-type parameter must be an integer constant

Output?

```
template <class T ,int i>
class mypair {
    T a, b;
public:
    mypair ()
    {a=0;
     b=0;}
};
int main () {
    int a=2;
    mypair <int , a> myobject;
    return 0;
}
```

In function 'int main()':

[Error] the value of 'a' is not usable in a constant expression

[Note] 'int a' is not const

[Error] the value of 'a' is not usable in a constant expression

[Note] 'int a' is not const

Using Default Arguments with Template Classes

A template class can be given a default argument:

```
template <class X=int> class myclass { //... };
```

and also like this:

```
template <class X, int size=10> class myclass { //... };
```

Using Default Arguments with Template Classes

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

template <class T=int>
class mypair {
    T a, b;
public:
    mypair ()
    {a=0;
     b=0;}
};

int main () {
    mypair <> myobject;
    return 0;
}
```

Explicit Class Specializations

Just like generic functions, we can also create an *explicit specialization* of a generic class

To do so, use the **template<> construct**

Explicit Class Specializations

For other data types:

```
template <class T> class myclass { //... };
```

For integers:

```
template <> class myclass<int> { //... };
```

Specializing class templates

```
template <> class mycontainer <char> { ... };
```

First of all, notice that we precede the class template name with an empty `template<>` parameter list. This is to explicitly declare it as a template specialization.

But more important than this prefix, is the `<char>` specialization parameter after the class template name. This specialization parameter itself identifies the type for which we are going to declare a template class specialization (char). Notice the differences between the generic class template and the specialization:








```
template <class T>
class Test
{public:
Test(){
    cout << "General template object \n";}
};
template <>
class Test <int>{
public:
Test(){
    cout << "Specialized template object\n";}};
int main(){
    Test<int> a;
    Test<char> b;
    Test<float> c;
    return 0;}
```

```
Specialized template object  
General template object  
General template object
```

```
-----  
Process exited after 0.8443 seconds with return value 0  
Press any key to continue . . . |
```


Explicit specialization of a generic class only takes 1 type

```
#include <iostream>
using namespace std;
template <class T>
class Test
{
public:
    Test(){
        cout << "General template object \n";}
};
template <>
class Test <int,char>{
public:
    Test(){
        cout << "Specialized template object\n";}};
int main(){
    Test<int , char> a;
    Test<float> c;
    return 0;}
```

 Compiler (5)  Resources  Compile Log  Debug  Find Results  Console  Close

Line	Col	File	Message
32	21	C:\SPRING 2024\Object Oriented Programming\...	[Error] wrong number of template arguments (2, should be 1)
25	7	C:\SPRING 2024\Object Oriented Programming\...	[Note] provided for 'template<class T> class Test'
		C:\SPRING 2024\Object Oriented Program...	In function 'int main()':
37	17	C:\SPRING 2024\Object Oriented Programming\...	[Error] wrong number of template arguments (2, should be 1)
25	7	C:\SPRING 2024\Object Oriented Programming\...	[Note] provided for 'template<class T> class Test'

Template specialization for multiple types

```
#include <iostream>
using namespace std;
template <class T , class V> //take 2 placeholder for multiple types
class Test
{
public:
    Test(){
        cout << "General template object \n";}
};

template <>
class Test <int,char>{
public:
    Test(){
        cout << "Specialized template object\n";}};
int main(){
    Test<int , char> a;
    Test<float , short> c;
    return 0;}
```

Common Applications

Stack

Queue

Other data structures