

Object-oriented Programming

Week 11 | Lecture 1

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
- A single general procedure can be applied to a wide range of data

Generic Functions

- A generic function is created using the keyword **template**
- A generic function is also called a *template function*

Templates in C++

- **Templates** are the foundation of generic programming, which involves writing code in a way that is independent of any particular type.
- **Templates** are powerful features of C++ which allows you to write **generic** programs.
A template is a **blueprint** or **formula** for creating a **generic class** or a **function**.
- In simple terms, you can create a single function or a class to work with different data types using templates.
- Templates are often used in larger **codebase** for the purpose of code **reusability** and **flexibility** of the programs.

What is the difference between function overloading and templates?

- Both are used to achieve Compile time Polymorphism
- Both function overloading and templates are examples of polymorphism feature of OOP. Function overloading is used when multiple functions do similar operations, templates are used when multiple functions do identical operations. You can use overloading when you want to apply different operations depending on the type. Templates provide an advantage when you want to perform the same action on types that can be different

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.

Types

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

Generic Functions/Function Templates

- A function template works in a similar to a normal function, with one key difference.
- A single function template can work with different data types at once but, a single normal function can only work with one set of data types.
- Normally, if you need to perform identical operations on two or more types of data, you use function overloading to create two functions with the required function declaration.

- 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);`

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> 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
}
```


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?

In function 'int main()':

[Error] no matching function for call to 'myfunc()'

[Note] candidate is:

[Note] template<class T1, class T2> void myfunc()

Output?

- `#include <iostream>`
- `using namespace std;`
- `template <class T1, class T2>`
- `void myfunc(){`
- `}`
- `int main () {`
- `myfunc();`
- `return 0;`
- `}`

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);`
- `return 0;`
- `}`

Output?

In function 'int main()':

[Error] no matching function for call to 'myfunc(int, int)'

[Note] candidate is:

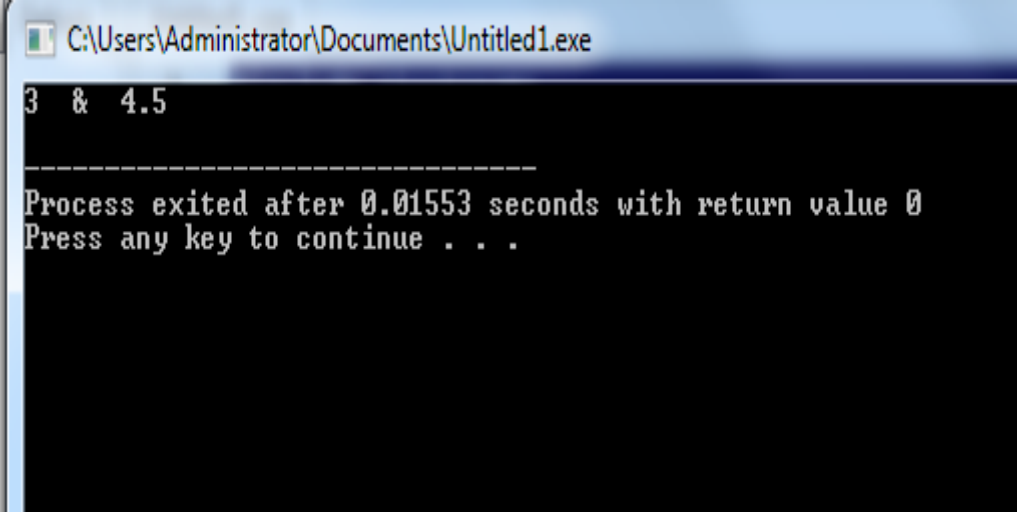
[Note] template<class T1, class T2> void myfunc(T1, T1)

Output?

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

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

Output?



```
C:\Users\Administrator\Documents\Untitled1.exe
3 & 4.5
-----
Process exited after 0.01553 seconds with return value 0
Press any key to continue . . .
```

Explicitly Overloading a Generic Function/Specializing templates

- We can explicitly overload a generic function
- If you overload a generic function, that overloaded function "*hides*" the generic function relative to that specific version
- This is formally called ***explicit specialization***

Explicitly Overloading a Generic Function/Specializing templates

- Normally when we write a template class or function we want to use it with many different types, however sometimes we want to code a function or class to make use of a particular type more efficiently this is when we use a template specialization. To declare a template specialization we still use the template keyword and angle brackets <> but leave out the parameters like so.
- `template<>`

Example

```
template <class X> void func (X a)
{
    cout << "Hello every data type: " << a;
}
```

*// Following version hides generic version if
parameter is int*

```
void func (int a)
{
    cout << "Hello integers: " << a;
}
```

Alternate Syntax

- A new-style syntax can also be used to denote the *explicit specialization* of a function:

```
template < > void func <int> (int a)
{
    cout << "Hello integers: " << a;
}
```

Alternate Syntax

- **template < > void func (int a)**
{
 cout << "Hello integers: " << a;
}

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

Example

// First version of f() template

```
template <class X> void f(X a)
{
    cout << "Inside f(X a)";
}
```

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

// Second version of f() template

```
template <class X, class Y> void f(X a, Y b)
{
    cout << "Inside f(X a, Y b)";
}
```

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

Generic Classes

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

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

Generic Classes

- If necessary, we can define more than one generic data type using a comma-separated list
- We create a specific instance of that class using the following general form:

class-name <type> ob;

Example

```
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; }
};
```

Example (cont.)

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

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

Example Program of Class Template

- `template <class T>`
- `class mypair {`
- `T a, b;`
- `public:`
- `mypair (T first, T second)`
- `{a=first; b=second;}`
- `T getmax ();};`
-
- `template <class T>`
- `T mypair<T>::getmax (){`
- `T retval;`
- `retval = a>b? a : b;`
- `return retval;}`
- `int main () {`
- `mypair <int> myobject (100, 75);`
- `cout << myobject.getmax();`
- `return 0;}`

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;`
- `}`

Example (cont.)

```
int main()
{
    myclass <100> intArray;
    myclass <double> doubleArray;
    myclass <> defArray;
}
```

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:

Common Applications

- Stack
- Queue
- Other data structures

Next Lecture

- STL!