Object-oriented Programming

Week 9 | Lecture 1

Generic Functions

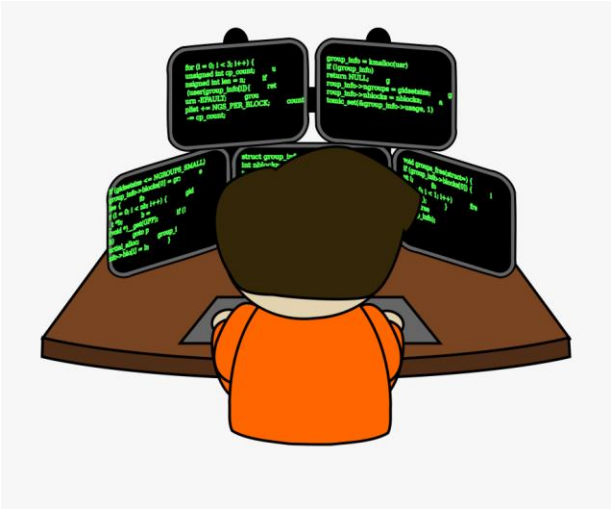
- A generic function defines a general set of operations that will be applied to various types of data. The type of data that the function will operate upon is passed to it as a parameter

- 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*



Generic Functions

- The general form of a template function definition is:

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
- We can use the keyword ***typename*** in place of ***class*** if we want

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 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';
}
```

Explicitly Overloading a Generic Function

- 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***

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;
}
```

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)";
}

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

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

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
- etc…

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 *
}
```

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

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 { //… };
```

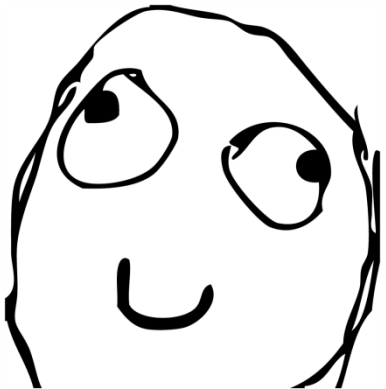
Example (cont.)

```
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> { //… };
```

Common Applications

- Stack
- Queue
- Other data structures