Object-Oriented Programming

WEEK 13

Abeeha Sattar

Generic Programming

- Generic Programming allows us to use different “types”, e.g., int, string, double or even user defined types, as a parameter for functions, classes and interfaces.

- This helps us increase the efficiency of our code, and...
- Write even less code!! How?

Generic Programming

- It helps us less write even less code because…
  - It lets us write generic algorithms that work with all datatypes
  - Thereby eliminating the need for overloading methods for different datatypes.

Advantages of Generic Programming

- Code Reusability
- Avoid Function Overloading
- Can be used multiple times for different data types!

Generics | Templates

- "**Generics**" are things we use to implement Generic Programming
- We can implement Generics in C++ with the help of "**Templates**"

Templates

- Think of templates for different things IRL…
  - Templates for websites
  - Templates for applications
  - Templates for CVs and Resumes

- What do these things have in common?

Templates in C++

- Normally, in order to use templates we use the following statement before our template function or template class:

template <class T>

OR

template <typename T>

- T is a type name;
- it need not be the name of a class

Function Templates

Syntax:

```
template <class T>
returnType functionName(parameter list) {
    // function body
}
```

Example: Function Template [Basic]

```
template <class T>
void SimplePrint(T var) {
    cout << "Parameter is: " << var << endl;
}

int main()
{
    int i = 20;
    char c = 'C';
    float f = 19.99;
    SimplePrint(i);
    SimplePrint(c);
    SimplePrint(f);
}
```

```
template <class T>
void SimplePrint(T var) {
    cout << "Parameter is: " << var << endl;
}
```

Example: Function Template [Swapping]

```
template <class T>
void swapargs(T& a, T& b) {
    T temp;
    temp = a;
    a = b;
    b = temp;
}


int main()
{
    int i = 20, j =10;
    char c = 'C', d = 'D';
    float f = 19.99, g = 10.99;

    swapargs(i,j);
```

Template Functions with More than One Generic Types

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

```
template <class T, class V>
void showDataType(T a, V b) {
    cout << typeid(a).name() << " " << typeid(b).name() << endl;
}
```

Specialized Templates

- Sometimes you don't want all datatypes and classes to share the same functionality.
- In this case we can define "specialized" functionality for a specific datatype or class.

- We do this by using Specialized Templates!

Example: Specialized Template for Strings

```
template<class T>
T add(T a, T b) {
    return a + b;
}

template<>
string add(string a, string b) {
    return a + " " + b;
}


int main()
{
    int i = 20, j =10;
    int k = add(i, j);

    string s = "ssss", t = "tttt";
    string u = add(s, t);
```

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: Template Overloading

```
template<class T>
T add(T a, T b) {
    return a + b;
}
```

```
template<class T, class U>
T add(T a, U b) {
    return a + b;
}
```

Using Normal Parameters in Generic Functions

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

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

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

Syntax: Generic Classes

- This is how we normally declare a generic class:

```
template <class T> class ClassName {
    //class body
};
```

- As with functions, we can have more than one generic type for a class.
- In order to do that, we can use a comma separated list of types in the template declaration

Creating Objects of Generic Classes

- We can create a specific instance of a generic class using the following general form:

ClassName <type> objectName;

Example: Generic Class

```
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 << endl;
    }
};
int main() {
    myclass<int, double> ob1(10, 0.23);
    myclass<char, string> ob2('H', "Hello");
    ob1.show(); // show int, double
```

Using Non-Type Arguments with Generic Classes

- You can also specify non-type arguments for a generic class.

```
template <class T, int size>
class ArrClass {
    T arr[size];

    public:
        ArrClass() {
            cout << "Default constructor called. Array of size " << size << " was
                    made\n";
        }
};
```

Fin.

:)