C++ Programming

Advanced Concepts, Templates, and More

Advanced Concepts, Templates, and More

- Templates
- Overloading
- Polymorphism
- The this Pointer
- Static Members
- Default Parameters

Templates

- Templates are a generic way of writing functions and classes that are capable of operating on more than one data type without having to duplicate the code
  - In a way, the data type becomes a parameter in the definition of the templated function or class
- In the end, templates can match hand-written, less general code in terms of run-time and space efficiency, but are more general and only require a single implementation to be defined

Function Templates

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

template <typename Type>
Type maximum(Type a, Type b) {
   if (a > b)
      return a;
   else
      return b;
}

int main(){
   std::cout << maximum(2, 4) << endl;
   std::cout << maximum(2.0, 4.0) << endl;

}
```

A function template defines a generic function (`maximum`) that can handle parameters of different types, while the code stays the same. (Here, finding the maximum of two numbers).

At call, different types of parameters are passed (e.g., integer and real numbers), giving us two different functions in the end.

Class Templates

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

template <class Type> class calc
{
   public:
     Type multiply(Type x, Type y);
     Type add(Type x, Type y);
};

template <class Type> Type calc<Type>::multiply(Type x, Type y) {
   return x*y;
}

template <class Type> Type calc<Type>::add(Type x, Type y) {
   return x+y;
}

int main() {
   calc<int> c;
   cout << c.add(10,20) << endl;
}
```

We are declaring a class template …

We tell the member functions to use the type provided by the class

When we declare an instance of the templated class, we specify the type we want it to operate on; in this case we are making an integer variant of the class

Final Notes on Defining Templates

- You can create templates that take more than one type using
  `template<typename T1, typename T2>`
- `template<typename …>` and `template<class …>` syntaxes are interchangeable when defining basic templates
  - That said, the syntax does matter when creating things like template templates (yes, you can do that!)
- Templates can get rather complicated; for more advanced use cases, check out your favourite C++ reference …

Standard Template Library (STL)

- The C++ Standard Template Library (or STL) provides a collection of general-purpose templated classes and functions that implement many commonly used algorithms and data structures
- The STL is standardized across C++ implementations; for portability and maintainability of code, the STL should be used wherever feasible instead of re-implementing things unnecessarily
- At the core of the STL are:
  - Containers, algorithms, and iterators

Standard Template Library (STL)

Component	Description
Containers	Used to manage collections of objects of a certain kind; for example: vector, list, map, queue, stack, and so on
Algorithms	Act on containers; for example: perform initialization, sorting, searching, and transforming of the contents of containers
Iterators	Step through the elements of collections of objects

Standard Template Library (STL)

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

int main() {
   vector<int> numbers;

   for (int count=0; count < 10; count++) {
      numbers.push_back(count);
      cout << numbers[count] << " is in the vector!" << endl;
   }

   cout << "Added " << numbers.size() << " numbers to the vector!" << endl;
}
```

We have created a vector of integers … it starts empty, but we can add to it quite readily …

Standard Template Library (STL)

- There are too many goodies in the STL to go through them all here

- Please consult your favourite C++ reference for a complete listing, and API documentation for this
  - www.cplusplus.com and www.cppreference.com are both fairly decent in this regard

Overloading

- C++ allows you to have more than one definition for a function name or operator in the same scope; this is referred to as overloading
- While they have the same name, overloaded functions and operators will have different argument types and, naturally, different implementations
- When you call an overloaded function or operator, the compiler will determine the most appropriate definition to use based on the types that you are using at the time

Function Overloading

- As noted above, you can have multiple definitions for the same function name in the same scope; this applies to methods or member functions of classes as well
- The definition of the function must differ from each other by the types and/or the number of arguments in the argument list
- You cannot overload function declarations that differ only by return type as the compiler might not be able to determine which version of the function you are attempting to call

Function Overloading

```
void print(int i) {
  cout << "Printing int: " << i << endl;
}

void print(double d) {
  cout << "Printing float: " << d << endl;
}

void print(string s) {
  cout << "Printing text: " << s << endl;
}
```

We have three functions named the same but each taking a different type

Function Overloading

```
#include<iostream>
using namespace


int main() {
    print(5);
    print(10.0);
    print("This is some text!");
}
```

When compiling, the compiler will determine which function is called in each instance based on the type of data passed in; this will call the integer, double, and string versions in sequence.

Function Overloading

```
include<iostream>
using namespace std;

int area(int length, int width) {
   return length*width;
}

float area(int radius) {
   return 3.14*radius*radius;
}

int main() {
   cout << area(5,5) << endl;
   cout << area(5) << endl;
}
```

We have two area functions this time, both operating on integer data, but with a different number of arguments in each case

Because the calls have differing numbers of arguments, the compiler still knows which function you want to use

Operator Overloading

- Most of the built-in operators in C++ can be redefined or overloaded
- Because of this, a programmer can also add and use operators with user-defined types, including classes
- As an overloaded operator is, to at least a certain extent, treated like a function behind the scenes, it is considered to have an argument list and a return type and follows the same general rules as an overloaded function

Operator Overloading

- For example, if we would like to be able to add or combine objects from one of our classes together, we could overload `operator+` and then bring them together using `object1 + object2`
- This is a powerful mechanism in C++, but can be confusing to other people if abused, overused, or used improperly
  - There is nothing stopping you, for example, for using `operator+` for other things; syntactically, your program would still make sense, but it would be much more difficult to understand from a semantic perspective

Operator Overloading

```
class Complex {

public:
   Complex(double re,double im) {
      real = re;
      imag = im;
   };
   Complex operator+(const Complex& other);

private:
   double real;
   double imag;
};

Complex Complex::operator+(const Complex&  other) {
   double result_real = real + other.real;
   double result_imaginary = imag + other.imag;
   return Complex(result_real, result_imaginary );
}

int main() {
   Complex c1(1,1);
   Complex c2(2,2);
   Complex c3 = c1 + c2;
}
```

We define a new addition operator to allow us to add two complex number objects together …

By doing this way, we can add our complex numbers in an intuitive and natural fashion

Operator Overloading

- As another interesting example, we can overload stream operators << and >> if we wanted to stream our own user-defined types to and from files
- For example, if we had a Rectangle class and an object from this class:

```
Rectangle r;
```

It would be nice to be able to output this by writing:

```
cout << r;
```

Operator Overloading

```
#include <iostream>
using namespace std;
class Rectangle {
   int width, height;
public:
   Rectangle(int w, int h) {
      width = w; height = h;

   }
   friend ostream& operator<<(ostream& os, const Rectangle& rect);
};

ostream& operator<<(ostream& os, const Rectangle& rect) {
   os << rect.width << "x" << rect.height;
   return os;
}

int main() {
   Rectangle r(10, 20);
   cout << r << endl;
}
```

We declare this as a friend so that our overloaded operator can access private data in our `Rectangle`.

The operator returns the output stream given as a parameter to allow us to chain operations together.

Operator Overloading

- The following operators can be overloaded:

+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

Operator Overloading

- These operators, however, can’t be overloaded:

::	.*	.	?:

Polymorphism

- Polymorphism means “many forms”
- In C++, we are usually referring to sub-type polymorphism, in which we can make use of derived classes through base class pointers and references
- Let’s look at an example …

Polymorphism

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

class Person {
public:
   void sayHello() {
      cout << "I am a Person." << endl;
   }
};

class Student: public Person {
public:
   void sayHello() {
      cout << "I am a Student." << endl;
   }
};

int main() {
   Person *p;
   Student *s = new Student();

   s->sayHello();
   p = s;
   p->sayHello();
}
```

Executing this code gives us:

```
I am a Student.
I am a Person.
```

Polymorphism

- Notice that the method invoked depends on the type of the pointer, not what the object actually is
- This is typically referred to as static dispatching, as it is determined by the C++ compiler when code is compiled
- This is the default for efficiency

Polymorphism

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

class Person {
public:
   virtual void sayHello() {
      cout << "I am a Person." << endl;
   }
};

class Student: public Person {
public:
   virtual void sayHello() {
      cout << "I am a Student." << endl;
   }
};

int main() {
   Person *p;
   Student *s = new Student();

   s->sayHello();
   p = s;
   p->sayHello();
}
```

Executing this code gives us:

```
I am a Student.
I am a Student.
```

Polymorphism

- The `virtual` keyword enabled dynamic dispatching
- In this case, the compiler does not know which method to invoke at compile time and instead the program needs to determine this at run-time as a derived class might override the method
- This gives flexibility, but costs in performance
- So, use `virtual` if a method may potentially be overridden

Polymorphism

- `virtual` is specified in the header file, not in the implementation
- A method declared as `virtual` stays virtual in all descendent classes
- For readability, convention is to continue adding `virtual` as a reminder, even though it is not strictly necessary

Polymorphism

- It is also possible to make use of pure virtual methods (also referred to as abstract methods in other languages)
- These are methods without an implementation
- These are used to force derived classes to implement particular methods
- These are declared as in this example:

```
virtual void sayHello() = 0;
```

Polymorphism

- The use of pure virtual methods gives rise to what are referred to as abstract classes
- These are classes that have one or more pure virtual methods
- These classes cannot be instantiated and are used to serve as base classes for other derived classes
- A derived class is concrete if and only if all inherited pure virtual methods are implemented

Polymorphism

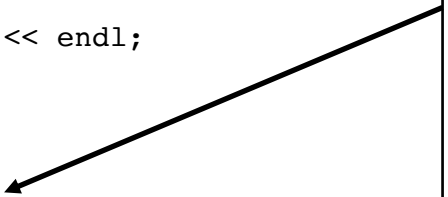
```
#include <iostream>
using namespace std;

class Person {
public:
   virtual void sayHello() = 0;
};

class Employee: public Person {
};

class Student: public Person {
public:
   virtual void sayHello() {
      cout << "I am a Student." << endl;
   }
};

int main() {
   Person *p = new Person();
   Employee *e = new Employee();
   Student *s = new Student();
}
```



Only the last statement here is acceptable. `Person` is an abstract class and so we cannot instantiate it. `Employee` is derived from `Person` but is still abstract, as it did not implement the `sayHello()` method. So, it cannot be instantiated either.

The this Pointer

- Every object in C++ has access to its own address through the use of a pointer called `this`
- The `this` pointer is an implicit parameter to all member functions, and so inside a member function, the `this` pointer can be used to refer to the invoking object

The this Pointer

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

class Rectangle {
   int width, height;
public:
   Rectangle(int width, int height) {
      width = width;
      height = height;
   }
};

int main() {
   Rectangle r(10, 20);
}
```

What happens here?

The this Pointer

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

class Rectangle {
   int width, height;
public:
   Rectangle(int width, int height) {
      this->width = width;
      this->height = height;
   }
};

int main() {
   Rectangle r(10, 20);
}
```

Much better!

Static Members

- By declaring a class member as static, it is possible to have that member belong to the class itself, as opposed to individual members of the class
- Such static members, in a way, are essentially shared among all objects of the class
- Static members can also be accessed without requiring the instantiation of an object in advance

Static Members

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

class Person {
private:
   static int counter;
public:
   Person() {
      counter++;
   }
   int getPersonCount() {
      return counter;
   }
};
int Person::counter = 0;

int main() {
   Person p1;
   Person p2;
   cout << p2.getPersonCount() << endl;
}
```

Our counter variable goes up every time we create an object of the class and we can access it to see how many we have made so far!

Static Members

- When we declare a member variable as `static`, we are simply telling the class that it exists
- As it is not attached to any object, no storage is allocated for it from only its class declaration
- As a result, `static` member variables need to also be declared separately to instantiate them and, optionally, initialize them

Static Members

- We can also have `static` member functions as part of our classes
- Again, these can be used without having to instantiate objects
- As these functions are not attached to particular objects, they do not have a `this` pointer

Static Members

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

class Person {
private:
   static int counter;
public:
   Person() {
      counter++;
   }
   static int getPersonCount() {
      return counter;
   }
};
int Person::counter = 0;

int main() {
   Person p1;
   Person p2;
   cout << Person::getPersonCount() << endl;
}
```

Because our accessor method to retrieve the count of objects created is now `static`, we no longer need to use one of our instances to access it …

Static Members

- There are some caveats to creating static members that should be kept in mind whenever using them
  - Static member variables are more-or-less the same as having global variables, albeit potentially with some access restrictions
  - Because static member variables are essentially shared between all objects of a class, they represent a threat to thread-safeness and access to them needs to be regulated like other globally shared data

Default Parameters

- Default parameters allow functions to be called without providing one or more trailing parameters
- Instead of explicitly providing such a parameter, a default value is substituted in its place
- This default value is specified in the declaration of the function
- Both regular functions and member functions of classes can use default parameters

Default Parameters

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

void printNumber(int i=0) {
    cout << i << endl;
}

int main() {
    printNumber(2);
    printNumber(1);
    printNumber();
}
```

As no parameter is given to our `printNumber()` function on the last call to it, it will use the default value of 0 for that call of the function.

Default Parameters

- Once a default parameter is specified in a parameter list for a function, all subsequent parameters must also have default values
- As another important note, default parameters can cause some attempts to overload a function to fail because it creates ambiguity between the two functions …

Default Parameters

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

void printNumber(int i=0) {
   cout << i << endl;
}

void printNumber() {
   cout << "?" << endl;
}

int main() {
   printNumber(2);
   printNumber(1);
   printNumber();
}
```

How would the compiler know which version of `printNumber()` to use? The one with no parameter or the one with a default parameter? It wouldn't know what to do, so this would not be allowed …