C++ Programming

Overloading and Polymorphism

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

but they must have same return type.

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

i.e differ on input param.

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

in fact, it is close to three different methods.

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

so make sure be clear what ur doing!

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;
}
```

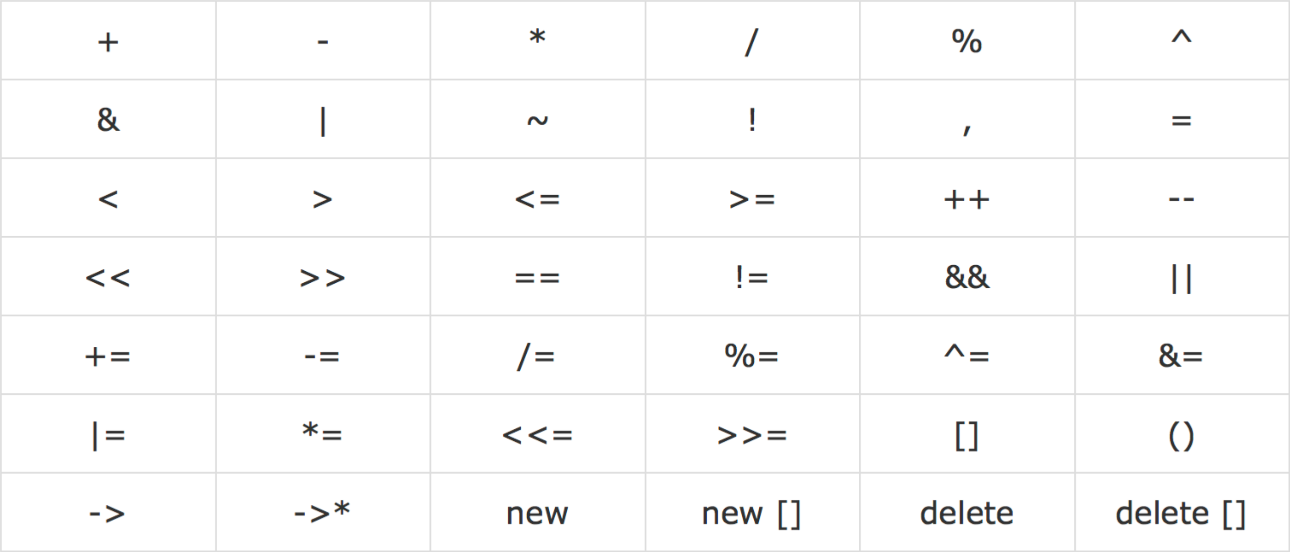
then we would chain out bunch of the functions

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.
```

But we cannot do reverse operation.

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();
}
```

← by default

<person> s -> sayHello()

=> I'm a person.

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)

in parent class, there's no implementation.

- 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;
```

enforce a consistent interface to all its child classes.

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();
}
```

student would be initiation since it has sayHello().

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.

it does not have sayHello() so, it would crash on calling.