Classes (II)

OOP in C++

1. Overloading Operators

In C++, different types have different interactions with operators. For example,

```
#include <string>

int a, b, c;
a = b + c; //arithmetic addition

std::string str1, str2, str3;
str1 = str2 + str3; //string concatenation
```

Therefore we sometimes have the need to define interactions with operators, as the operation to be performed can be ambiguous. Consider:

```
#include <string>

class MyClass {
  std::string str;
  float number;
} a, b, c;

a = b + c;
```

Here, it is not obvious what the + operator should do. In fact, this would cause a compilation error, as we have not explicitly defined behaviour for +. The good news is that C++ allows for most operators to be overloaded so that their behvaiour can be defined for almost any type, including classes. We overload opertors by using `operator` functions, which take on the form

```
type operator sign (parameters) {body}
```

For example, consider cartesian vector addition. We have two vectors (a, b) and (x, y), and their sum is defined as (a+x, b+y).

```
#include <iostream>
class CVector {
public:
  int x, y;
  CVector (int a, int b) : x(a), y(b) {}
  CVector operator+ (const CVector&); //+ operator prototype
  //Note that we have to pass a const lvalue reference of CVector, to allow
  ↪ rvalue references. In other words, this setup ensures that an expression
  ↪ such as a = b + CVector(1, 2) is valid. This is discussed in further
  ↪ detail in upcoming sections.
};

CVector CVector::operator+ (const CVector& b) {
```

```
  CVector result;
  result.x = x + b.x;
  result.y = y + b.y;
  return result;
}

int main() {
  CVector foo(3, 1);
  CVector bar(5, 2);

  CVector result;
  result = foo + bar;

  cout << result.x << ','<< result.y << std::endl;
  return 0;
}
```

Output:

```
8,3
```

Note that we can also call the operator explicitly:

```
result = foo.operator+ (bar); //8,3
```

We can also overload operators as non-member functions, as below:

```
#include <iostream>
class CVector {
public:
  int x, y;
  CVector (int a, int b) : x(a), y(b) {}
  CVector operator+ (const CVector&); //+ operator prototype
};

CVector operator+ (const CVector& lhs, const CVector& rhs) {
  CVector result;
  result.x = lhs.x + rhs.x;
  result.y = lhs.y + rhs.y;
  return result;
}

int main() {
  CVector foo(3, 1);
  CVector bar(5, 2);

  CVector result;
  result = foo + bar;

  cout << result.x << ','<< result.y << std::endl;
  return 0;
}
```

Output:

```
8,3
```

Notice that when in this case, we must pass an instance of the class as a first argument.

2. "This"

The keyword `this` is a pointer to the the object itself. It is analogous to the keyword `self` in Python, however it is implicit in C++.

One of its uses can be to check if a parameter passed to a member function is the object itself. For example:

```
#include <iostream>

class Dummy {
public:
  bool isitme(Dummy& param);
};

bool Dummy::isitme(Dummy& param) {
  if (&param == this)
    return true;
  return false;
}

int main() {
  Dummy a;
  Dummy* b = &a;

  if (b->isitme(a))
    cout << "&a is b" << std::endl;
  return 0;
}
```

Output:

```
&a is b
```

3. Static Members

A class can contain static data or functions. A static data member is known as a class variable. Static data members are constant amongst all object instantiations of the class.

```
#include <iostream>

class Dummy {
public:
  static int n;
  Dummy() {n++;} //increment n each time we create an instantiation of Dummy.
};

int Dummy::n = 0;
```

```
10
11 int main() {
12   Dummy a;
13   Dummy b[5];
14   std::cout << a.n << std::endl;
15   Dummy *c = new Dummy;
16   std::cout << Dummy::n << std::endl;
17   delete c;
18   return 0;
19 }
```

Output:

```
6
7
```

In general, it is good practice to initialize static members outside of the class definition, using the scope operator. This is to avoid reassignment of the static member whenever a new object is created. Because it is a common variable amongst all objects, it can be referred to as a member of any object of that class or directly by the class name, by scoping.

4. Const Member Functions

When an object of a class is defined using `const`, it restricts the object as if all members were read only. THIS ALSO INCLUDES CONST REFERENCES!!!!

```
1  #include <iostream>
2
3  class MyClass {
4  public:
5    int x;
6    MyClass(int val) : x(val) {}
7    int get() {
8      return x;
9    }
10 };
11
12 int main() {
13   const MyClass foo(10);
14   //foo.x = 15;  //Compile Error: Cannot modify read only
15   std::cout << foo.x << std::endl;
16   return 0;
17 }
```

Output:

```
10
```

We are unable to modify `foo.x`, as `foo` was declared as `const`. However, notice that we are able to initialize `x = 10`.Member functions of a `const` object can only be called if they are also specified as `const`. To specify this, we add the `const` keyword after the parameters:

```
int get() const {return x;}
```

Note that `const` in this context is different to returning a `const` type; here, only `const` members can be called for `const` objects.

```
int get() const {return x;} //const member function
const int& get() {return x;} //nonconst member function returning a const&
const int& get() const {return x;} //const member function returning a const&
```

We are also able to overload "constness":

```
#include <iostream>

class MyClass {
  int x;
public:
  MyClass(int val) : x(val) {}
  //const
  const int& get() const {return x;}

  //nonconst
  int& get() {return x;}
};

int main() {
  MyClass foo(10);
  const MyClass bar(20);

  foo.get() = 15; //dispatched to nonconst
  //bar.get() = 20; //NOT allowed: return type of bar.get() is const int&, which
  ↪  cannot be modified.

  std::cout << foo.get() << std::endl;
  std:: cout << bar.get() << std::endl;

  return 0;
}
```

Output:

```
15
20
```

5. Class Templates

Class templates allow us to store data of any type. Consider:

```
template <class T>
class MyPair {
  T values[2];
public:
  MyPair(const T (&arr)[2]) : values{arr[0], arr[1]} {}
  }
};
```

This code will store two elements of any type, so long as they are both the same. For example, the following examples are all valid:

```
MyPair<int> myInt(115, 36);
MyPair<double> myFloats(3.0, 2.18);
MyPair<std::string> myStrings("hello", "world");
```

In order to define member functions outside of the class definition, ensure that it is preceded with the `template <...>` prefix:

```
#include <iostream>

template <class T>
class MyPair{
  T values[2];
public:
  MyPair(T first, T second) {
    a = first; b = second;
  }

  T getmax();
};

template <class T>
T MyPair<T>::getmax() {
  T returnValue;
  returnValue = a > b ? a : b;
  return returnValue;
}

int main() {
  MyPair<int> myObject(100,75);
  std::cout << myObject.getmax() << std::endl;
  return 0;
}
```

Output:

```
100
```

6. Template Specialization

We can define different behaviour for a template depending on the type that is passed. For example, suppose we have a class `MyContainer` that has a `increase` member function, which just increments its value. However, if we were to pass a `char`, we would want different implementation, say, `uppercase`.

```
#include <iostream>

//class template:
template <class T>
class MyContainer {
  T element;
```

```
public:
  MyContainer(T param) : element(param) {}
  T increase () {
    return ++element;
  }
};

//class template specialization:
template <>
class MyContainer <char> {
  char element;
public:
  MyContainer (char param) : element(param) {}
  char uppercase (){
    if ((element>='a') && (element<='z'))
      element += 'A'-'a';
    return element;
  }
};

int main() {
  MyContainer<int> myInt(7);
  MyContainer<char> myChar('j');
  std::cout << myInt.increase() << std::endl;
  std::cout << myChar.uppercase() << std::endl;
  return 0;
}
```

Output:

```
8
J
```

The syntax used for class template specialization:

```
template <> class myContainer <char> {...};
```