Intro to OOP

C++

1. Classes (I)

1.1. Introduction

Class: A class is a type of datastructure that can hold data members and member functions.
Object: An instantiation of a class.
Classes are defined using the keyword `class` or `struct`. A typical class implementation may look like the following:

```
class ClassName {
accessSpecifier1:
  member1;
accessSpecifier2:
  member2;
    ...
} objectNames;
```

Access specifiers define access rights to the data stored within a class. They can be one of `private`, `public`, or `protected`. By default, members are set to `private`.

1. `private` members can only be accessed from within the class itself.
2. `public` members can be accessed anywhere.
3. `protected` members can be accessed from within the class and through derived classes.

```
 class Rectangle {
   int width, length;
 public:
   void set_values(int, int);
   int area(void);
} rect;
```

In the code above, we create a Rectangle class and instantiate an object rect. We define a private width and length, and public methods `set_values` and `area`. We are able to access the public members simply as normal function calls:

```
rect.set_values(5, 4);
rect.area();
```

Note that the code above does not implement any behaviour to `set_values` nor `area`. We can either implement behaviour within the class as a normal function, or use the scope operator and define it outside, as follows:

```
#include <iostream>

//create Rectangle Class
class Rectangle {
  int width, length;
public:
  void set_values(int, int);
  int area(void) {
    return width*length;
    }
};

//We can implement behaviour of prototype functions within a class by using the
↪  :: operator.
void Rectangle::set_values(int x, int y){
  width = x;
  length = y;
}

int main(){
  Rectangle rect;
  rect.set_values(3, 4);
  std::cout << rect.area() << std::endl;
  return 0;
}
```

The code above will print the value 12. Note that each instantiation of a class is independent to each other. For example,

```
...
int main(){
  Rect rectA, rectB;
  recta.set_values(3, 4);
  rectb.set_values(5, 6);
  std::cout << "rectA area: " << rectA.area() << std::endl;
  std::cout << "rectB area: " << rectB.area() << std::endl;
  return 0;
}
```

Output:

```
rectA area: 12
rectB area: 30
```

1.2. Constructors

In the code shown above, what would happen if we were to call `area()` before `set_values`? An undetermined result, since we havent initialized width or height. In order to avoid this behaviour, we can introduce a special member, known as a constructor.
Constructor: A constructor is a method that is called automatically when a new object is instantiated.
A constructor is defined by a method that has the exact same name as the class, and no return type; not even `void`. For example:

```cpp
#include <iostream>

class Rectangle {
  int width, height;
public:
  Rectangle(int, int); //prototype constructor - takes the same name as the
   ↪  class, with no return type
  int area(void) {
    return width * height;
  }
};

//constructor definition
Rectangle::Rectangle(int x, int y){
  width = x;
  height = y;
}

int main(){
  //instantiate rect with (3, 4) parameters
  Rectangle rect(3, 4);
  std::cout << rect.area() << std::endl;
  return 0;
}
```

Output:

```
12
```

1.2.1 Overloading Constructors

Constructors also have the ability to be overloaded. This can be useful in cases where you want different instantiation behaviour. For example:

```cpp
#include <iostream>

class Rectangle {
  int width, height;
public:
  //Rectangle constructor with 2 parameters
  Rectangle(int x, int y){
    width = x;
    height = y;
  }

  //Rectangle constructor with 1 parameter
  Rectangle(int x){
    width = x;
    height = x;
  }

  //overload default constructor
  Rectangle(){
    width = 10;
    height = 10;
  }

  int area(void){
    return width * height;
  }
};

int main() {
  Rectangle rect(3, 4);
  Rectangle square(5);
  Rectangle default_rect;
  return 0;
}
```

In this case, since `square` only has one parameter, it will assume the Rectangle constructor that only accepts one parameter. As well, we have overloaded the default constructor, which is called when we pass no parameters. Note that `default_rect()` would NOT work. In this case, it would create a `default_rect()` function declaration, instead of calling the default constructor.

1.2.2 Uniform initialization

The method of calling constructors as shown above, by enclosing the arguments in parenthesis is known as functional form. However, C++ allows for other syntaxes.

First, if a constructor only takes a single parameter, it can be called implicitly with variable assignment:

```
ClassName objectName = initializationValue;
```

The code above is equivalent to

```
ClassName objectName = initializationValue;
```

Added in C++11, uniform initialization is essentially the same as functional form, but instead uses braces.

```
ClassName objectName { value1, value2, ... }
```

The benefit of uniform initialization is that it strictly enforces types, which protects you from accidently losing precision.

```
int x{3.14}; //ERROR: Narrowing conversion
int y(3.14); //Allowed, however y becomes 3.
```

For example:

```
#include <iostream>
const double PI = 3.14159265

class Circle {
  double radius;
public:
  Circle(double r) {
    radius = r;
  }

  double circum() {
    return 2 * radius * PI
  }
};

int main() {
  Circle foo(10.0); //Functional initialization
  Circle bar = 20.0; //Assignment initialization
  Circle baz {40.0}; //uniform initialization
  return 0;
}
```

Whilst this choice of constructors is largely just a mater of style, many newer style guides suggest to use uniform initialization.

1.2.3 Member Initialization in Constructors

Many times in a constructor, we need only to initialize other members. This can be done by using a colon and a list of initializations. For example,

```
class Rectangle {
  int width, length;
public:
  Rectangle(int, int);
  int area() {
    return width*length;
  }
}
```

Here, the constructor can be defined as shown before as

```
Rectangle::Rectangle(int x, int y) {width = x; length = y;}
```

Or, using member initialization:

```
Rectangle::Rectangle(int x, int y) : width(x), length(y) {}
```

Notice that we have an empty constructor body as well. This is because we want the constructor to only initialize the members, and have no added functionality.

While default constructing may just seem like a convenient shorthand, it does have some applications where it is necessary. Consider the following:

```
#include <iostream>
const double PI = 3.14159265

class Circle {
  double radius;
public:
  Circle(double r) : radius(r) {}
  double area() {
    return radius * radius * PI;
  }
```

```cpp
};

class Cylinder {
  Circle base;
  double height;
public:
  Cylinder(double r, double h) : base(r), height(h) {}
  double volume() {
    return base.area() * height;
  }
};

int main() {
  Cylinder foo(10, 20);

  cout << "Volume: " << foo.volume() << std::endl;
  return 0;
}
```

In this example, the class `Cylinder` has a member of type `Circle`. Because objects of class `Circle` can only be initialized with a parameter, `Cylinder` needs to invoke `Circle`'s constructor. This is only possible through a member initialization list.
Note that initialization lists can also be created using uniform initialization.

```cpp
Cylinder::Cylinder(double r, double h) : base{r}, height{h} {} //with {} syntax
```

1.3. Pointers to Classes

Since classes are just an object type, we can define a pointer that points at a class type. For example,

```cpp
Rectangle *pRect;
```

Creates a pointer to an object of class `Rectangle`.
Much like with built in datastructures, we can access members of an object directly from a pointer by using the arrow operator.

```cpp
#include <iostream>

class Rectangle {
  int width, length;
public:
  Rectangle(int x, int y) : width(x), length(y) {}
  int area(void) {
    return width * height;
  }
};

int main() {
  Rectangle obj(3, 4); //object instantiation
  Rectangle *foo, *bar, *baz;
  foo = &obj;
  bar = new Rectangle(5, 6);
  baz = new Rectangle[2]{{2, 5}, {3, 6}};

  cout << "obj area: " << obj.area() << std::endl;
  cout << "*foo area: " << foo->area() << std::endl;
  cout << "*bar area: " << (*bar).area() << std::endl;
  cout << "baz[0] area: " << baz[0].area() << std::endl;
  cout << "baz[1] area: " << *(baz + 1).area() << std::endl;

  delete bar;
  delete[] baz;
  return 0;
}
```

1.4. Classes Defined with Struct and Union

Classes also be defined using the keyword `struct` and `union`.

The keyword `struct` can also be used to declare classes that have member functions, which have the exact same syntax as `class`, however members have `public` access by default.

The concept of unoins, however, is quite different to that of `class` and `struct`. Unions can only have a single data member active at a time, but they can also hold multiple member functions. That is, members in a union will share a common memory address, with the size allocated being the largest member. The default access for union is `public`.

2. Classes (II)

2.1. 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 explicity defined behavaiour 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 opeartors 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.
};

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.

2.2. "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 it's 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 Dumy::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
```

2.3. 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;

int main() {
  Dummy a;
  Dummy b[5];
  std::cout << a.n << std::endl;
  Dummy *c = new Dummy;
  std::cout << Dummy::n << std::endl;
  delete c;
  return 0;
}
```

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 reffered to as a member of any object of that class or directly by the class name, by scoping.

2.4. 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!!!!

```
#include <iostream>

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

int main() {
  const MyClass foo(10);
  //foo.x = 15;  //Compile Error: Cannot modify read only
  std::cout << foo.x << std::endl;
  return 0;
}
```

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
  ↪  cannnot be modified.

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

```

```cpp
  return 0;
}
```

Output:

```
15
20
```

2.5. Class Templates

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

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

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

```cpp
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:

```cpp
#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
```

2.6. 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`.

```cpp
#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> {...};
```

3. Special Members

Special member functions are member functions that are, by default, implicitly defined as member functions (under specific circumstances). There are six:

1. **Default constructor**: `C::C();`
2. **Destructor**: `C::~C();`
3. **Copy constructor**: `C::C(const C&);`
4. **Coppy assignment**: `C& operator=(const C&);`
5. **Move constructor**: `C::C(C&&);`
6. **Move assignment**: `C& operator=(C&&);`

3.1. Default Constructor

If a class is not explicity defined with a constructor, the compiler assumes a default constructor. Therefore, the following class:

```
class Example{
public:
  int total;
  void accumulate(int x) {
    total += x;
  }
};
```

is assumed to have a default constructor. Therefore, we can initialize an object by declaring them with no parameters:

```
Example myObject;
```

However, once we explicitly define a constructor, the compiler will no longer provide the defualt constructor therefore:

```
class Example2 {
public:
  int total;
  Example2 (int initial) : total(initial) {}
  void accumulate(int x) {
    total += x;
  }
}
```

We must provide an argument when we instantiate the class.

```
Example2 myObj(100); //Okay: implicitly calls constructor
Example2 myObj = Example2(100) //Okay: explicitly call constructor
Example2 ex; //Not valid: no default constructor provided.
```

3.2. Destructor

Destructors do the opposite of constructors: they are called when an object is destroyed, or when it goes out of scope. Up untill now, we have not had to allocate any memory when creating objects, therefore the need for a destructor was not necessary. However, imagine that the class dynamically allocates memory to store a string. In this case, it would be quite useful to have a function that is automatically called whenever the object goes out of scope, so that we can release it's memory. We can create this by using a destructor. They are created in a similar fashion to constructors, where they have no return value, but they take no arguments and the class name is preceded with a tilde ( ).

```
#include <string>
#include <iostream>

class Example3 {
  std::string *ptr;
public:
  Example3(): ptr(new std::string) {} //default constructor
  Example (const std::string& str) : ptr(new std::string(str)) {}

  //destructor:
  ~Example3() {delete ptr;}

  //access content(Read only)
  const std::string& content() const {return *ptr;}
};

int main() {
  {Example3 foo;}
  Example3 bar ("Example");
  std::cout << "bar: " << bar.content() << std::endl;
  return 0;
}
```

Output:

```
bar: Example
```

3.3. Copy Constructor

When an object is passed a **named** object of its own type as an argument, its copy constructor is invoked to copy the object. A copy constructor is a constructor who's first parameter is a const reference to the class itself (Why? A copy should never change the original). Thus a copy constructor must have the following signature:

```
MyClass::MyClass(const MyClass&)
```

If a class has no custom copy nor move constructors defined, an implicit copy constructor is provided. This copy performs a *shallow copy*. For example,

```
class MyClass {
public:
  int a, b;
  string c;
}
```

The implicit copy constructor would look something like:

```
MyClass::MyClass(const MyClass& x): a(x.a), b(x.b), c(x.c) {}
```

In most use cases, the default copy constructor is sufficient. However, shallow copies only copy the member of the class. Consider `Example3`. When we copy `Example3`, this would create a copy of only the pointer value, and not the content; thus, both pointers would point to the same value, and would share a single `string` object. At some point for destruction, both objects would attempt to delete the same value, and would (in most cases) cause a classic `segfault`. This can be solved by creating a custom copy constructor that performs a deep copy:

```
#include <iostream>
#include <string>

class Example4 {
  std::string *ptr;
public:
  //constructor
  Example4(const std::string& str) : ptr(new std::string(str)) {}
  //destructor
  ~Example4() {delete ptr;}
  //copy constructor:
  Example4(const Example4& other) : ptr(new std::string(other.content())) {}
  //access content:
  //NEEDS TO BE CONST METHOD!!!! a const method (the copy constructor) is
  ↪  invoking this method, thus it also needs to be const.
  const std::string& content() const {
    return *ptr;
  }
};

int main() {
  Example4 foo("Example");
  Example4 bar = foo;

  std::cout << "bar: " << bar.content() << std::endl;
  return 0;
}
```

Output:

```
bar: Example
```

3.4. Copy Assignment

Seen above, we are calling the copy contructor when declaring a new variable /cppinlinebar. However, copying can also be done through assignment.

```
MyClass foo;
MyClass bar(foo); //object declaration: copy constructor invoked
MyClass baz = foo; //object declaration: copy constructor invoked
```

```
foo = bar; //object assignment: copy assignment invoked
```

The copy assignment operator is an overload of `operator=` which takes a reference of the class as a parameter. The value is generally a reference to `this*`. For example, for `MyClass`, a copy assignment could have a signiture that looks like:

```
MyClass& operator=(Const MyClass&);
```

Implicitly, the copy assignment is defined as a shallow copy, much like the copy constructor. However, we can implement a deep copy in a similar fashion:

```
//inside Example4
Example4& operator=(const Example4& other) {
  *ptr = x.content() //ptr is not const
  return *this;
}
```

3.5. Move Constructor and Assignment

Move semantics can be one of the most confusing parts of learning C++. After all, the definition for `std::move` on cppreference for many can cause even more confusion:

> `std::move` produces an xvalue expression that identifies its arguement `t`. It is exactly equivalent to a `static_cast` to an rvalue reference type.
> Returns `static_cast<typename std::remove_reference<T>::type&&>(t)`

Doesn't really tell us that much. Simply put, moving an object involves "transferring" ownership of content from one object to another. The source loses it's content, which is taken over by the destination. This move can only occur iff the source is an unnamed object.
Unnamed object: Unnamed objects are objects that are temporary in nature, and thus don't have a name. Often times these are rvalues: values which don't have a spot in memory. Other examples may include return values of function or type-casts.
Using the value of a temporary object to initialize another object/assign it value does not require a copy; after all, there is no address to copy from. Instead, we move the value of the source into the destination. This invokes a different kind of method: move constructors and move assignments.

```
MyClass fn(); //function returning a MyClass object
MyClass foo; //default constructor
MyClass bar = foo; //copy constructor
MyClass baz = fn(); //move constructor
foo = bar; //copy assignment
baz = MyClass; //move assignment; equal to baz = MyClass temp{};
```

The move constructor and assignment are members that take an rvalue reference of the class. Syntactically, this is done with a double ampersand.

```
MyClass (MyClass&&); //move constructor: takes a rvalue reference
MyClass& operator=(MyClass&&); move assignment
```

In classes that must dynamically allocate memory, move constructors and assignments are extremely important. We don't want to create a copy of the class if we are just passing a temporary instance of the class.

```
#include <iostream>
#include <string>

class Example5 {
  std::string *ptr;
public:
  Example5 (const std::string& ptr) : ptr(new std::string(str)) {}
  ~Example5 () {delete ptr;}

  //move constructor
  Example5 (Example5&& other) {
    this->ptr = other.ptr; //copies the pointer to this
    other.ptr = nullptr; //deassigns other pointer
  }

  //move assignment
  Example5& operator=(Example5&& other) {
    ptr = other.ptr;
    x.ptr = nullptr;
    return *this;
  }

  //access content
  const std::string& content() const {return *ptr;}
  //addition
  Example5 operator+(const Example5& rhs) {
    return Example5(content() + rhs.content());
  }
};

int main() {
  Example5 foo("Exam");
  Example5 bar = Example5("ple"); //move constructor
  foo = foo + bar; //move assignment
  std::cout << "foo: " << foo.content() << std::endl;
}
```

Output:

```
foo: Example
```

We can also invoke the move constructor using a pre-existing object, and transfer ownership using `std::move`.

```
Example5 foo("Example");
Example5 bar = std::move(foo) //move constructor invoked: foo is valid but
↪ unspecified, and bar contains "Example"
```

Maybe now the definition of `std::move` makes more sense. All it does is it simply takes an object and casts it to an rvalue, and nothing else. Since we are then defining `bar` with an rvalue, it invokes the move constructor.