Notes for ECE 39595 - Object-Oriented Programming with C++

Zeke Ulrich

January 27, 2025

Contents

Course Description

This course teaches C++ and the principles of object oriented programming. It covers the basics of the C++ language, including inheritance, virtual function calls and the mechanisms that support virtual function calls. Design patterns and general principles of programming will be covered.

Introduction

Let's examine some basic C++ programs.

Listing 1: Hello World

```
#include <iostream>

int main() {
    std::cout << "Hello␣World!";
    return 0;
}
```

Listing 2: User Input

```
#include <iostream>

int main() {
    double n;
    int i;

    std::cout << "Enter␣float:␣";
    std::cin >> n;

    std::cout << "Enter␣integer:␣";
    std::cin >> i;

    return 0;
}
```

The `stds` you're seeing all over the place refer not to a frat party but the standard namespace. It holds useful objects like standard in ("`cin`") and standard out ("`cout`").

In C, we use `malloc` and `free` to allocate memory. In C++, the equivalent operations are `new` and `delete`.

Listing 3: Free and Delete

```
void CorrectUsage(){
    int *ptr = new int[3];
    int *ptr1 = new int;
    ptr[0] = 1;
    ptr[1] = 2;
    ptr[2] = 3;
    *ptr1 = 5;
    delete ptr1;
    delete [] ptr;
```

The reason for this is using `new` and `free` calls an object's constructor and destructors, which are defined to properly delete the object. Technically, `malloc` and `free` are both present in C++, but they won't trigger the constructors and destructors and should be avoided. The choice to leave these functions in was made to improve compatibility with C, which is a theme the reader may notice in C++'s many possible ways to do the same thing.

C++ filenames are terminated with a `.cpp` or `.cc` extension, like `example.cpp` or `example.cc`

Objects and Classes

A class stores functions (methods) and values for a bunch of objects you may want to create that follow its blueprint, like characters in a video game or car models.

Listing 4: Objects and Classes

```
#include <iostream>

class Dog {
        public:
        std::string breed;
        std::string name;

        void bark() {
                std::cout << "Woof!" << std::endl;
            }
    };

int main() {
        Dog my_dog;
        my_dog.breed = "Labrador";
        my_dog.name = "Buddy";

        Dog your_dog;
        your_dog.breed = "Poodle";
        your_dog.name = "Coco";

        std::cout << my_dog.breed << std::endl;
        my_dog.bark();

        return 0;
    }
```

There are many ways to create a new object in C++.

Listing 5: Alternative Instantiations

```
#include "Dog.h"
int main(int argc, char* argv[ ]) {
    Dog* myDog = new Dog("Sam");
    Dog myDog("Sanjay");
    Dog myDog = Dog("Jenna");
    ...
}
```

There are also many ways to access values in an object. One potentially novel way is through the -> operator. It's shorthand for dereferencing a pointer and then accessing a member of the object it points to.

Listing 6: -> Operator

```
class Example {
public:
    int value;
    void show() {
        std::cout << "Value: " << value << std::endl;
    }
};

int main() {
    Example obj;
    obj.value = 10;

    Example* ptr = &obj;  // Pointer to the object

    // Accessing members using the pointer
    ptr->value = 20;      // Equivalent to (*ptr).value = 20;
    ptr->show();          // Equivalent to (*ptr).show();

    return 0;
}
```

In C++, the `private` and `public` keywords define access control for members within a class. They determine how and where those members can be accessed outside the class. `private` members are accessible only within the class in which they're declared. `public` members are accessible from anywhere a member of the class is visible.1

References

Just like C, C++ passes parameters by reference. Primitive variables like `int`, `float`, `char`, `pointer` as passed as a copy in parameters. When an object variable is passed as a parameter, it's also passed as a copy. Since each object can be arbitrarily large, this could incur a large cost. The way we get around this is by passing a reference (address) to the object instead. When a reference to an object is passed as an argument, only a copy of the reference is passed.

Listing 7: Passing Objects by Value and Reference

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

class MyClass {
public:
    int value;
    MyClass(int v) : value(v) {}
};

// Function to modify object passed by value
void modifyByValue(MyClass obj) {
    obj.value = 42; // Modifies the copy, not the original
}

// Function to modify object passed by reference
void modifyByReference(MyClass& obj) {
    obj.value = 42; // Modifies the original object
}

int main() {
    MyClass obj(10);

    cout << "Original value: " << obj.value << endl;

    // Pass object by value
    modifyByValue(obj);
    cout << "After modifyByValue: " << obj.value << endl;

    // Pass object by reference
    modifyByReference(obj);
    cout << "After modifyByReference: " << obj.value << endl;

    return 0;
```

```
}
```

Copy Constructor

The copy constructor allows you to copy the values of one object to another. If you have any pass-by-value functions in your class, then you'll need to create a copy constructor since pass-by-value creates a copy of the object to manipulate within the method.

Listing 8: Copy Constructor

```
#include <iostream>
#include <cstring>

class Person {
private:
    char* name;
    int age;

public:
    Person(const char* personName, int personAge) {
        name = new char[strlen(personName) + 1]; // Allocate memory
        strcpy(name, personName); // Copy the string
        age = personAge;
    }

    // Copy constructor
    Person(const Person& other) {
        name = new char[strlen(other.name) + 1]; // Allocate new memory
        strcpy(name, other.name); // Copy the name
        age = other.age; // Copy the age
    }
};

int main() {
    Person person1("Alice", 25); // Create the first object
    Person person2 = person1; // Use the copy constructor

    return 0;
}
```

We must be careful in the case of self-assignment (`x = x`), especially if anything is being deleted in the copy constructor. It's wise to include a check for self assignment and to simply return `*this` if true.

Listing 9: Self Assignment Check

```
    const MyClass& MyClass::operator=(const MyClass& copyFrom) {
```

```
    if(this == &copyFrom) { return *this; }

    // other copying logic
}
```