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

Zeke Ulrich

February 3, 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

```cpp
#include <iostream>

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

Listing 2: User Input

```cpp
#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

```cpp
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

```cpp
#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

```cpp
#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

```cpp
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

```cpp
#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

```cpp
#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

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

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

    // other copying logic
}
```

Hashing

Hashing converts input data into a fixed-size string of characters (the "hash") using a hashing algorithm.

Properties

Effective hashing algorithms and systems typically exhibit several important properties:

- **Determinism:** A given input will always produce the same hash output, ensuring consistency across operations.

- **Uniformity:** Hash values are uniformly distributed over the output range. This minimizes the chance of clustering and improves performance in applications like hash tables.

- **Efficiency:** The hashing process should be computationally efficient, making it feasible to hash large volumes of data quickly.

- **Irreversibility:** Especially in cryptographic applications, a good hash function should make it infeasible to reconstruct the original input from the hash value.

- **Collision Resistance:** It should be computationally difficult to find two distinct inputs that produce the same hash output. This property is crucial for maintaining the integrity of the hashing process.

Algorithms

There are many hashing algorithms, each designed with specific use cases and properties in mind. Some common algorithms include:

- **MD5:** Once popular for its speed and simplicity, MD5 produces a 128-bit hash. However, due to vulnerabilities to collision attacks, it is now considered insecure for cryptographic purposes.

- **SHA Family:** The Secure Hash Algorithm (SHA) family, including SHA-1, SHA-256, and SHA-3, offers enhanced security over MD5. SHA-256, for example, produces a 256-bit hash and is widely used in various security applications.

- **CRC (Cyclic Redundancy Check):** Although not used for cryptographic security, CRCs are used for error-checking in data transmission and storage due to their efficiency in detecting accidental changes in data.

Hash Tables

Hash tables (or hash maps) are data structures that utilize hash functions to map keys to values, offering efficient insertion, deletion, and lookup operations. The performance of hash tables heavily depends on the quality of the underlying hash function and the strategies used to handle collisions.

- **Structure:** A hash table typically consists of an array of buckets, where each bucket can store one or more key-value pairs. The key is processed by the hash function to determine the appropriate bucket.

- **Collision Resolution:** Common methods for resolving collisions in hash tables include:

  - **Chaining:** Each bucket holds a list (or another data structure) of entries that hash to the same value. When a collision occurs, the new entry is simply added to the list.

  - **Open Addressing:** When a collision occurs, the algorithm probes for the next available bucket according to a predetermined sequence (e.g., linear probing, quadratic probing, or double hashing).

- **Performance:** In the average case, hash table operations (insertion, deletion, lookup) have a time complexity of $O(1)$. However, performance can degrade if many collisions occur or if the table is poorly sized relative to the number of stored elements.

Collisions

A collision occurs when two different inputs produce the same hash output. While collisions are mathematically inevitable given the fixed size of hash outputs relative to the potentially infinite size of input data, good hash functions minimize their occurrence and make finding collisions computationally difficult.

- **Collision Resistance:** A hash function is considered collision-resistant if it is hard to find any two distinct inputs that hash to the same value. This is essential for cryptographic applications where collision attacks can compromise data integrity and security.

- **Birthday Paradox:** The probability of collisions is often analyzed using the birthday paradox, which shows that the likelihood of a collision increases significantly as more hashes are generated. This paradox emphasizes the need for a sufficiently large hash size.

- **Mitigation Strategies:** To mitigate the risk of collisions, designers use larger hash sizes, more complex algorithms, or even combine multiple hash functions. In non-cryptographic contexts, collision resolution strategies (such as chaining or open addressing) are implemented in data structures like hash tables.

Collision Resolution

When two distinct keys hash to the same bucket, a collision occurs. Collision resolution strategies are essential to maintain efficient operations in a hash table. Two common methods are separate chaining and open addressing. Here, we focus on an in-depth discussion of separate chaining.

Separate chaining is a widely-used collision resolution method where each bucket in the hash table is not limited to storing a single key-value pair. Instead, each bucket contains a secondary data structure (commonly a linked list, but sometimes a dynamic array, tree, or another structure) that holds all the key-value pairs whose keys hash to that particular bucket.

Mechanism:

- **Insertion:** When a new key-value pair is inserted, the hash function computes the bucket index for the key. The new pair is then added to the list (or other container) at that index.

- **Search:** To retrieve a value, the hash function again computes the bucket index from the key. The algorithm then searches through the list in that bucket, comparing the keys until it finds a match.

- **Deletion:** Similarly, to delete a key-value pair, the hash function locates the bucket, and the algorithm traverses the list to find and remove the matching entry.

Variants and Optimizations:

- **Using Balanced Trees:** Some implementations replace the linked list with a balanced tree (such as a red-black tree) when the chain exceeds a certain length. This change ensures that the worst-case search time improves from $O(n)$ to $O(\log n)$.

- **Dynamic Resizing:** To maintain efficient average-case performance, hash tables using separate chaining often resize (rehash) when the load factor (the average number of entries per bucket) exceeds a predefined threshold. Resizing redistributes the keys across a larger array of buckets, thereby reducing the average chain length.

- **Hybrid Approaches:** Some hash table implementations combine separate chaining with open addressing. For example, they may use open addressing for buckets with few collisions but switch to separate chaining when the bucket becomes too full.

The time complexity of separate chaining with a well behaved hash function is $O(\frac{n}{m})$, where m is the number of buckets.