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

Zeke Ulrich

April 4, 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 `std`s 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
}
```

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.

Templates

A *template* is useful for when you don't want to prescribe a specific data type for some class.

For example, a template to create a pair of any data types can be defined as

Listing 10: -> Operator

```
template <class T1, class T2>
// can use 'typename' instead, like <typename T1 ...
class MyPair{
        private:
        T1 a;
        T2 b;
        public:
        T1 first ();
        T2 second ();
        MyPair( const T1 &mfirst,
        const T2 &msecond);
    };

// Instantiated as
MyPair(1, "hello");
```

The convention followed in this class is that template definitions go into `.h` files while the implementations go into `.hpp` files, which must include the `.h` files.

Inheritance

The *has-a* relationship describes a class C using an object of class H to hold information about class C. For instance, consider a `Car` class that has a string `model` within it. `Car` has `String`.

The *is-a* relationship is when an object of one class is also an object of another class. For instance, a `Student` instance may also be a `Person` class. We say that `Student` *inherits* or *extends* `Person`.

Listing 11: Inheritance

```
#include <iostream>

// Base class
class Animal {
public:
    void eat() {
        std::cout << "This␣animal␣eats." << std::endl;
    }
};

// Derived class inheriting from Animal
class Dog : public Animal {
public:
    void bark() {
        std::cout << "The␣dog␣barks." << std::endl;
    }
};

int main() {
    Dog myDog;
    myDog.eat();  // Inherited from Animal
    myDog.bark(); // Defined in Dog
    return 0;
}
```

Access Levels

Public

Public members are accessible from any part of the program where the object is visible. They form the interface of the class. In inheritance, public members remain public when using public inheritance, though they can become private under private inheritance.

```
class MyClass {
public:
    int publicVar;
    void publicMethod();
};

int main() {
    MyClass obj;
    obj.publicVar = 42;    // Accessible from anywhere
    obj.publicMethod();    // Accessible from anywhere
    return 0;
}
```

Protected

Protected members are accessible within the class that declares them, as well as in classes derived from it and friend classes. In public inheritance, protected members remain protected in the derived class. However, they cannot be accessed from outside the class hierarchy.

```
class Base {
protected:
    int protectedVar;
public:
    Base() : protectedVar(0) {}
};

class Derived : public Base {
public:
    void modify() {
        protectedVar = 10;  // Accessible here in the derived class
    }
};

int main() {
    Derived d;
    // d.protectedVar = 5; // Error: protectedVar is not accessible outside Base or Derive
```

```
    return 0;
}
```

Private

Private members are only accessible within the class that declares them and by its friend functions or classes. They are not directly accessible in derived classes, although they do exist as part of the derived object. If a derived class needs to interact with these members, it must do so through public or protected member functions of the base class.

```
class Base {
private:
    int privateVar;
protected:
    int protectedVar;
public:
    Base() : privateVar(0), protectedVar(0) {}
};

class Derived : public Base {
public:
    void tryAccess() {
        // privateVar = 5;  // Error: privateVar is not accessible here
        protectedVar = 10;  // OK: protectedVar is accessible in the derived class
    }
};

int main() {
    Base b;
    // b.privateVar = 5;  // Error: cannot access private member
    return 0;
}
```

Generic Programming

Generic programming is a style of computer programming in which algorithms are written in terms of types to-be-specified-later that are then instantiated when needed for specific types provided as parameters. This approach allows for code reusability and type safety. Generic programming is primarily achieved through the use of templates. Templates allow functions and classes to operate with generic types, which makes it possible to create a function or class to work with any data type.

A function template works by defining a pattern for a function that can operate on any data type. Here is an example of a simple function template that returns the maximum of two values:

```
template <typename T>
T max(T a, T b) {
    return (a > b) ? a : b;
}
```

Class templates allow you to define a class that can operate with any data type. Here is an example of a simple class template for a pair of values:

```
template <typename T1, typename T2>
class Pair {
public:
    Pair(T1 first, T2 second) : first_(first), second_(second) {}
    T1 first() const { return first_; }
    T2 second() const { return second_; }
private:
    T1 first_;
    T2 second_;
};
```

Function Pointers

Function pointers are variables that store memory addresses of functions. They allow functions to be passed as arguments to other functions, returned from functions, and stored in data structures. In the context of generic programming, function pointers enable writing algorithms that can work with different behaviors specified at runtime.

The syntax for declaring a function pointer involves specifying the return type and parameter types:

```
// Declaring a function pointer
return_type (*pointer_name)(parameter_types);
```

Here's a basic example of using function pointers:

```
#include <iostream>

int add(int a, int b) {
    return a + b;
}

int subtract(int a, int b) {
    return a - b;
}
int main() {
    // Declare function pointer
    int (*operation)(int, int);

    // Assign function pointer to add
    operation = add;
    std::cout << "Result of add: " << operation(5, 3) << std::endl;

    // Reassign function pointer to subtract
    operation = subtract;
    std::cout << "Result of subtract: " << operation(5, 3) << std::endl;

    return 0;
}
```

Function pointers can be combined with templates for powerful generic programming:

```
template <typename T>
T processValues(T a, T b, T (*operation)(T, T)) {
    return operation(a, b);
}

template <typename T>
T multiply(T a, T b) {
    return a * b;
}
int main() {
    // Use the generic function with different types
    int intResult = processValues<int>(5, 3, multiply);
    double doubleResult = processValues<double>(2.5, 1.5, multiply);

    std::cout << "Int result: " << intResult << std::endl;
    std::cout << "Double result: " << doubleResult << std::endl;
```

```
    return 0;
}
```

Modern C++ provides std::function from the <functional> header, which offers a more flexible alternative to traditional function pointers:

```
#include <iostream>
#include <functional>

template <typename T>
T processValues(T a, T b, std::function<T(T, T)> operation) {
    return operation(a, b);
}

int main() {
    // Using lambda expressions
    auto divide = [](double a, double b) { return a / b; };

    double result = processValues<double>(10.0, 2.0, divide);
    std::cout << "Result: " << result << std::endl;

    return 0;
}
```

Threading

A *process* is an independent program in execution, with its own memory space, while a *thread* is a smaller unit of execution within a process that shares the same memory space as other threads in the same process. Threads are lightweight and allow multiple tasks to run concurrently within a single process.

Threads are commonly used to perform tasks in parallel, such as handling multiple client requests in a server or performing background computations while keeping the main application responsive. However, threads don't return anything, so variables that need modified should be passed by reference.

Creating Threads in C++

C++ provides support for multithreading through the `<thread>` header. A thread can be created by instantiating a `std::thread` object and passing a callable (such as a function, lambda, or functor) to its constructor.

Here is an example of creating and joining threads:

```
#include <iostream>
#include <thread>

// Function to be executed by a thread
void printMessage(const std::string& message) {
    std::cout << "Thread says: " << message << std::endl;
}

int main() {
    // Create a thread and pass a function to it
    std::thread t1(printMessage, "Hello from thread!");

    // Wait for the thread to finish
    t1.join();

    std::cout << "Main thread finished." << std::endl;
    return 0;
}
```

In this example, a thread `t1` is created to execute the `printMessage` function. The `join()` method ensures that the main thread waits for `t1` to complete before proceeding.

Using Lambda Functions with Threads

Lambda functions are often used with threads for simplicity and flexibility. Here's an example:

```
#include <iostream>
#include <thread>

int main() {
    // Create a thread using a lambda function
    std::thread t([]() {
        std::cout << "Hello from a lambda thread!" << std::endl;
    });

    // Wait for the thread to finish
    t.join();

    std::cout << "Main thread finished." << std::endl;
    return 0;
}
```

Thread Synchronization

When multiple threads access shared resources, synchronization is required to avoid race conditions. C++ provides synchronization primitives like `std::mutex` to protect shared data.

Here is an example of using a mutex to synchronize threads:

```
#include <iostream>
#include <thread>
#include <mutex>

std::mutex mtx; // Mutex for synchronization
int sharedCounter = 0;

void incrementCounter() {
    for (int i = 0; i < 1000; ++i) {
        std::lock_guard<std::mutex> lock(mtx); // Lock the mutex
        ++sharedCounter;
    }
}

int main() {
    std::thread t1(incrementCounter);
    std::thread t2(incrementCounter);
```

```
    t1.join();
    t2.join();

    std::cout << "Final counter value: " << sharedCounter << std::endl;
    return 0;
}
```

In this example, `std::mutex` ensures that only one thread modifies `sharedCounter` at a time. `std::lock_guard` automatically locks and unlocks the mutex within its scope.

Detaching Threads

A thread can be detached to run independently of the main thread. However, care must be taken to ensure that the thread does not access resources that may go out of scope.

```
#include <iostream>
#include <thread>
#include <chrono>

void backgroundTask() {
    std::this_thread::sleep_for(std::chrono::seconds(2));
    std::cout << "Background task completed." << std::endl;
}

int main() {
    std::thread t(backgroundTask);
    t.detach(); // Detach the thread

    std::cout << "Main thread continues execution." << std::endl;
    std::this_thread::sleep_for(std::chrono::seconds(3)); // Ensure main thread lives long
    return 0;
}
```

In this example, the `detach()` method allows the thread to run independently. The main thread ensures it lives long enough for the detached thread to complete.