

Introduction

This document provides a basic introduction to C++ Classes. It should be used in conjunction with the code files for the C++ example class called **ExampleClass**. For any given topic discussed in this document, there are many additional important details not discussed because this is a basic introduction. The official online resources for C++ can be consulted for these details. Links to some of these resources, specifically Microsoft and C++ Reference, are provided at the end in the **Resources** section. Both are excellent resources, and the Microsoft links are more beginner friendly and easier to understand.

Note: for the purposes of demonstration, the std namespace is being used by including the statement `using namespace std`. This is just to make code easier to read. For example, it allows `string` to be explicitly written instead of `std::string` to refer to a C++ string, and it allows `cout` to be explicitly written instead of `std::cout`. It is okay to use the std namespace for the purposes of learning, teaching, or demonstration, but in real C++ code it is not a good practice.

Table of Contents

Topics	Page
General Setup and File Organization	2
Member Variables	4
Member Functions	5
Resources	8

General Setup and File Organization

Header File

Header file: This can be a ".h" or ".hpp" file. The difference is that ".h" files can be used for C or C++, and ".hpp" files are exclusively for C++. The header file is named "ExampleClass.h" because the class is named ExampleClass. Always include **header guards** to prevent accidental inclusion of a header file twice. The header guard macro is EXAMPLE_CLASS_H because the file is named ExampleClass.h

Header file contents - Within the header file is the **class declaration** which contains declarations for **member variables** and **member functions**. Alternatively, function declarations can be omitted and the function definitions can be directly written inside of the class. These are called **inline functions**. The member variables and member functions together create the **internal representation** of the class.

Access specifiers - The keywords **public**, **private**, and **protected** are called **access specifiers**, and they control access to the internal representation of the class. The **protected** keyword is not discussed here because it is more related to the concept of inheritance which is not discussed in this document. Below are definitions of the public and private access specifiers as well as what parts of the internal representation of the class are conventionally designated as public or private.

- **private:** A member variable or function that is directly accessible only within the class such as in member functions and **friend functions** (discussed later). The parts of the class's internal representation that should be private are:
 - **Member variables** - The data stored by the class. Member variables can be anything including all primitive types, structures, and classes (both C++ built-in and user defined). Note that there are some cases where member variables should be made public (also protected, but again not discussed in this document).
 - **Helper functions** - Functions that are only called inside of the class member functions, and are never outside such as in the main function. They "help" the class do various tasks. Whether or not a class has helper functions is just situationally dependent on their utility. For example, if there was a particular operation that needed to be done frequently throughout many of the member functions, it might be helpful to delegate that task to a helper function.
- **public:** A member variable or function that is directly accessible anywhere outside of the class. The member variables can be accessed and modified anywhere, and the member functions can be called anywhere, such as in the main function. The parts of the class's internal representation that should be public are:
 - **Member functions** - Any and all functions that need to be called outside of the class like in the main function. The member functions act as an intermediary between the internal representation of the class and the world outside of the class. They are used to reveal what values are stored in the member variables as well as set the member variables to new values. They will always protect the member variables from being set to an invalid state, such as setting the size of an array to a

negative number. This protection of data is one of the most important concepts in object-oriented programming.

- **Member variables** - There are cases where member variables should be made public though it is not the majority of the time. A typical example would be a constant, such as a mathematical constant like PI. Since it's a constant, it can't be changed so it's okay to make it public.

Implementation File

Implementation file: This is the ".cpp" file named "ExampleClass.cpp".

Implementation file contents - Within the implementation file are initializations for any static member variables and function definitions for any non-inline member functions.

Main File

Main File: This is the ".cpp" file named "main.cpp" where the **main function** is. The main function is a special function where the program runs. This is where objects of classes are instantiated and used in a program.

The main function is not part of the class, but it is such an integral part of any C++ program that it is included here. Any C++ program at a minimum must have a main function. The most basic C++ program would include one single file - the main.cpp file, and an empty main function like as follows

```
int main(int argc, char* argv[]) {

    return 0;

}
```

The "int argc, char* argv[]" are for using command line arguments. Command line arguments are discussed in another document, and an understanding of them is not necessary for an understanding of C++ classes.

Member Variables

As previously stated, member variables are typically made private and can be any primitive type, structure, and built-in or user-defined class. A built-in class is a class built into the C++ language like `vector` or `string`, and a user-defined class is a class created by a user like `ExampleClass`. Though structures can be included, they are not part of `ExampleClass`.

Private Member Variables - These are private to protect them from getting modified in an undesirable way.

```
private:
    // Built-in class and primitive type
    string word;
    double x;

    // User-defined vector of integers (not using C++ vector)
    int *vec;
    int size;
    int capacity;
```

The following would be invalid in the main function

```
ExampleClass ec;
ec.size = -1;           // invalid, size is private
```

Public Member Variables - These are public because they don't need data protection. `PI` is public because it's a constant meaning its value can't get changed. Therefore, it can never be set to an invalid state and doesn't need the protection of the private access specifier.

```
const static int PI = 3.14;
```

Non-Static vs. Static Variables

- Static

- A single copy of the static variable is shared amongst every object of the class.
- The static variable can be referred to without instantiating an object of the class
- *Constant static vs. non-constant static variables.*
 - Constant static variables can be declared and defined simultaneously in the class declaration. This is `PI` in `ExampleClass`.
 - Non-constant static variables can be declared in the class declaration but must be defined outside of the class declaration like in the implementation file. This is `y` in `ExampleClass`.

- Non-static

- Each object has its own unique copy of the variable.
- The variable cannot be referred to without instantiating an object of the class.
- This is `word`, `x`, `vec`, `size`, and `capacity` in `ExampleClass`.
- In the following example, assume that `word` were public, not private.

```
cout << ExampleClass::PI;           // valid, prints the value in PI (3.14)
cout << ExampleClass::y;           // valid, prints the value in y (10)
cout << ExampleClass::word;        // invalid, word is not static
```

Member Functions

As previously stated, member functions are typically made public and are most commonly used to see what data is stored in the member variables and set the data in the member variables to new values. There are a number of different kinds of member functions all described below.

Constructors: Special functions that are called automatically when an object gets instantiated. They give the object an initial state meaning they initialize all of the member variables with certain values. Constructors have no return value, and their name is the same name as the class. Note that a naming convention is to prepend an underscore to each parameter. This helps identify which parameter is for which variable while simultaneously avoiding a naming collision. Also, note that constructors often use **member initializer lists** which is when member variables get initialized before the body of the constructor executes. This is the preferred way of initializing member variables as opposed to directly assigning them values in the body of the constructor.

- **Default Constructor:** A constructor that either has no parameters or all of its parameters are default values. Regardless, it initializes an object to some default State. If no user-defined constructors exist, then the compiler automatically generates an inline default constructor. This compiler generated default constructor should not be relied on - if a default constructor is to be used, always create a user-defined default constructor.

`ExampleClass();`

- **N-Parameter Constructor:** A constructor that takes N parameters (3 in ExampleClass) and instantiates a new object to some customized state based on those parameters.

`ExampleClass(string _word, double _x, int _capacity);`

1-Parameter constructors should always have the `explicit` keyword.

`explicit ExampleClass(int _capacity);`

- **Copy Constructor:** A constructor that takes another object (specifically an lvalue reference) of the same class as an argument, and instantiates a new object that is a complete and independent copy of that object. If no user-defined copy constructor is provided, the compiler automatically generates one. This compiler generated copy constructor should not be relied on - if a copy constructor is to be used, always create a user-defined copy constructor. The `const` keyword means that the internal representation of `ec` will not be changed - `ec` is being copied, but its not itself being modified.

`ExampleClass(const ExampleClass& ec);`

- **Move Constructor:** A constructor that takes another object of the same class as an argument (specifically an rvalue reference), and instantiates a new object whose data is not copied from but rather transferred from that object. Move constructors are part of **move semantics** and are a more advanced C++ feature not meant for discussion in an introduction for C++ classes. They are not demonstrated in the example code.

`ExampleClass(ExampleClass&& ec) noexcept;`

Destructor: A special function that gets called automatically when an object goes out of scope, and it deallocates any and all memory associated with an object. Therefore, it is the opposite of a constructor. The destructor has no return value, is the same name as the class, and starts with the `~` character. A user-defined destructor must be created only if the class contains any member variables that were allocated on the heap with the `new` keyword. In this case, within the destructor their memory must be deallocated and returned to the **heap** using the `delete` keyword.

```
~ExampleClass();
```

In `ExampleClass`, there is a member variable allocated on the heap named `vec`. Therefore, a user-defined destructor had to be created which deallocated the memory for `vec`. If no member variables were dynamically allocated, then no user-defined destructor would need to be created. In C++, `new` and `delete` are to `malloc` and `free` in C.

Copy Assignment Operator: The copy assignment operator is a function that falls under a completely separate topic in C++ called **operator overloading**. However, it is such an important feature of a class it is being mentioned here. It takes two already existing objects, `ec1` and `ec2`, and creates a complete and independent copy of `ec2` and stores it in `ec1`. Any data previously stored in `ec1` gets replaced.

```
ExampleClass& operator=(const ExampleClass& ec);
```

The copy constructor creates a copy of an object and stores it in a brand new object, whereas the copy assignment operator creates a copy of an object and stores it in an already existing object

Get / Getter Functions: Functions that get i.e. return the values stored in the member variables of the object. This allows those values to be seen, but not modified, outside of the class .

- For example, the class has a `double` value called `x`, and the `get` function for `x` simply returns `x` thus allowing it to be seen what number is currently stored in `x`. The `const` keyword is always used with getter functions, and it means the function will not modify the internal representation of the object.

```
double getX() const {
    return x;
}
```

Set / Setter / Mutator Functions: Functions that set the member variables in the object to new values. If necessary, they should employ error checking. They do not include the `const` keyword since they are in fact modifying the internal representation of the class.

```
void setX(double newX) {
    x = newX;
}
```

The above example did not employ error checking because there were no limits on what `x` could be. It's just a variable that can be set to any floating point number. Error checking is situationally dependent, and an example of where error checking would come into play is if there were a class that represented a date and it contained three integers to represent the day, month, and year. In that case, the `setDay`, `setMonth`, and `setYear` functions would all employ error checking. The year could never be set to a negative number, the month could never be set to a number outside of the range `[1, 12]`, and the day could only be set to numbers within the range `[1, 30]`, `[1, 31]`, `[1, 29]`, and `[1, 28]` for leap year. As an example, the `setMonth` function would look like the following:

```
void setMonth(int newMonth) {
    if (newMonth >= 1 && newMonth <= 12) {
        month = newMonth;
    }
}
```

```
}

```

Note how if `newMonth` were outside of the valid range, `month` would remain unchanged. Thus is an example of data protection - `month` is private, so it cannot be modified directly. The only way it can be modified is via the function `setMonth`, and because `setMonth` prevents the `month` from ever being set to an invalid value, there is no way in a program where `month` could ever be set to an invalid value.

Other Functions: Any function that does not fall directly under the hierarchy of one of the special categories of functions.

- For example, `ExampleClass` includes a vector, and a function to print out all of the values in the vector could be written. These functions may or may not include the keyword `const`; it's situation dependent.

```
void printVec( ) const;
```

Friend Functions: A function that is not a member function of the class but has direct access to the private member variables and functions in objects of that class. In this sense, friend functions violate the principle of information hiding and encapsulation in object-oriented programming, and as a result of this there is a debate about whether or not they are good practice or should be used. Regardless, they are a feature in C++. In the example below, the private member variables and functions in `ec` can be directly accessed meaning the value stored in `x` within `ec` can be seen as well as set by directly using `ec.x`. If `printWord` were not a friend function, then `ec.getX()` and `ec.setX()` would have to be used to get the value of `x` and set the value of `x` within `ec`.

```
friend void printWord(const ExampleClass& ec);
```

Note the difference between friend and non-friend member functions in regards to the placement of their function definitions.

Static Functions: A function that can be called without instantiating an object just like with static member variables. Inside of the function definition of a static function, the only components of the class it can access are static member variables and other static functions.

```
static void printPI( );
```

The above function simply prints out the constant static variable in the class called `PI`. It can do this because `PI` is a static member variable. If `PI` were not a static member variable, then it couldn't access `PI`.

Resources

Classes

- Microsoft: <https://docs.microsoft.com/en-us/cpp/cpp/classes-and-structs-cpp?view=msvc-160>
- C++ Reference: <https://en.cppreference.com/w/cpp/language/classes>

Access Specifiers

- Microsoft: <https://docs.microsoft.com/en-us/cpp/cpp/member-access-control-cpp?view=msvc-160>
- C++ Reference: <https://en.cppreference.com/w/cpp/language/access>

Static Variables and Functions

- Microsoft: <https://docs.microsoft.com/en-us/cpp/cpp/static-members-cpp?view=msvc-160>
- C++ Reference: <https://en.cppreference.com/w/cpp/language/static>

Constructors:

- Microsoft: <https://docs.microsoft.com/en-us/cpp/cpp/constructors-cpp?view=msvc-160>
- C++ Reference: <https://en.cppreference.com/w/cpp/language/constructor>

Destructors

- Microsoft: <https://docs.microsoft.com/en-us/cpp/cpp/destructors-cpp?view=msvc-160>
- C++ Reference: <https://en.cppreference.com/w/cpp/language/destructor>

Copy Assignment Operator

- Microsoft: <https://docs.microsoft.com/en-us/cpp/cpp/copy-constructors-and-copy-assignment-operators-cpp?view=msvc-160>
- C++ Reference: https://en.cppreference.com/w/cpp/language/copy_assignment

Friend Functions

- Microsoft: <https://docs.microsoft.com/en-us/cpp/cpp/friend-cpp?view=msvc-160>
- C++ Reference: <https://en.cppreference.com/w/cpp/language/friend>