1 Who is this for?

If you are taking this class and are not familiar with some of the features of C++, then this guide is for you. In other words, if any of these words are unfamiliar to you or you are unsure of how to use them:

- Inheritance
- Polymorphism
- Copy and Move Semantics
- Shared, Weak, and Unique Pointers

then you should read this document. You should already be familiar with the basics of classes. However, this guide is meant to be brief and give you a basic understanding of the concepts so that they are not foreign to you. This document does not complain to be thorough or give you a complete picture on any of these concepts. You are encouraged to Google and do more research on your own.

1.1 Who is this NOT for?

If you do not know C++ at all, this document will not attempt to teach you it. Please consult Google or take CS106B before taking this class. Obviously, if you’re familiar with these concepts, feel free to skip this document.

2 Object-Oriented Programming

In a nutshell, object-oriented programming is the art of viewing the world as “objects” and modeling the relationship between these different types of objects (classes) via inheritance and composition.

2.1 Inheritance

Inheritance is the concept of modeling a is-a relationship. For example, a “cow” is a “mammal” which is an “animal.” To give another example, a “square” is a “rectangle” which is a “shape.” This is-a relationship is modeled with inheritance in the object-oriented programming world. Let’s take the cow example and see how public inheritance works in C++.

Let’s say we have a base class called animal:

```cpp
class Animal {
public:
    void Quack();

protected:
    bool has_ears;

private:
    int health_points;
```
Creating a class that publically inherits from another class is simple (note that “public Animal” part):

```cpp
class Mammal: public Animal
{
public:
    void Moo();
};
```

So what has happened here? The Mammal class now publically inherits from the Animal class. This has a few implications:

- All of Animal’s public functions and variables are now also public in Mammal and in all child classes.
- All of Animal’s protected functions and variables are now also protected in Mammal and in all child classes.
- All of Animal’s private functions and variables are NOT accessible Mammal and in all child classes.
  But this DOES NOT mean that those functions and variables have “disappeared.”
- Anyone can know of the relationship between Mammal and Animal.

As a result, this code can compile:

```cpp
Animal* cow = new Mammal();
cow->Quack();
```

Why? Since a Mammal publically is-a Animal, we can create an Animal pointer that points to a Mammal object. Furthermore, we can call the “Quack” function because the Mammal class inherits the public function of Animal. However, since an Animal is not a Mammal, the following code would not work:

```cpp
Mammal* cow = new Animal();
```

Let’s define the “Moo” function as follows:

```cpp
void Mammal::Moo() {
    std::cout << "Has Ears: " << has_ears << std::endl;
}
```

This compiles fine because protected variables “has_ears” stay protected (meaning the variable can be accessed in all child classes) in public inheritance.

```cpp
void Mammal::Moo() {
    std::cout << "Health: " << health_points << std::endl;
}
```

This does not compile because private variables stay private in public inheritance. Only the class in which the private variable is defined can access the private variable. This, however, does not mean the variable is gone! The Mammal class still has “health_points,” it just can not access it. To do so, you would have to add a public/protected function to Animal and call it from the Mammal class:

```cpp
int Animal::GetHealthPoints() {
    return health_points;
}
```

The following would then compile:

```cpp
void Mammal::Moo() {
    std::cout << "Health: " << GetHealthPoints() << std::endl;
}
Why bother with protected and private variables and functions then? If you are curious, I would suggest reading “Effective C++” by Scott Meyers—Item 22 gives a much better explanation than I could ever give. If you are not, just be aware that this exists in the code and you will be dealing with this in the future (Assignment 3 to be exact).

You will notice that I stressed the public part of the inheritance. This is because protected and private inheritance also exist in C++. They are very rarely used (and not used at all in the framework code) so if you are interested, please consult your nearest Google.

2.2 Polymorphism

In the previous section, you will notice that the Mammal’s “Quack” function will do the same thing as the Animal’s “Quack” function. But what if you wanted the Mammal to “Quack” differently from other Animals? A simple solution is to just write a new Quack function:

```cpp
class Mammal: public Animal
{
public:
    void Quack() {
        std::cout << "Ain't nobody can tell me how to quack!" << std::endl;
    }
    void Moo();
};
```

Then this piece of code will call the Mammal’s “Quack”:

```cpp
Mammal* cow = new Mammal();
cow->Quack();
```

However, this piece of code will call the Animal’s “Quack”:

```cpp
Animal* cow = new Mammal();
cow->Quack();
```

What if you wanted the second case to also call Mammal’s “Quack”? This is where polymorphism comes in. Effectively, this is the art of getting a base class pointer that points to a derived class object to call the overridden derived class’s functions instead of the base class’s. This is very useful! C++ accomplishes this with virtual functions. Let’s change our Animal and Mammal setup.

```cpp
class Animal
{
public:
    virtual void Quack() {
        std::cout << "Quack" << std::endl;
    }
};
class Mammal: public Animal
{
public:
    void Quack() override {
        std::cout << "Moo" << std::endl;
    }
};
```

Notice the use of the virtual and override keywords. The virtual keyword lets the compiler know that this function can potentially be overridden by child classes and the child class’s overridden function should be called EVEN when a pointer to the base class is used. On the other hand, the override function just lets
the compiler know that the function overrides a virtual function. This is helpful as it lets the compiler catch errors where you write a function that you should overrides a virtual function but does not really (for example when you have a typo). Now, both of the following situations will call Mammal’s “Quack” and output Moo.

```cpp
Mammal* cow = new Mammal();
cow->Quack();

Animal* cow2 = new Mammal();
cow2->Quack();
```

## 3 Copy and Move Semantics

The differentiation between copy and move semantics is meant to programmers write better performaing code with operations defined within the C++ languages/C++ STL. This is necessary for the implementation of smart pointers discussed in the next section. Let’s take a look at a simple benchmark (note that this benchmark is meant to get a point across not that you would ever do this in real code). This first example uses copy semantics:

```cpp
int main(int argc, char** argv){
    std::vector<std::string> cheese(10000);
    for (int i = 0; i < 10000; ++i) {
        cheese[i] = "Doodles " + std::to_string(i);
    }
    for (int i = 0; i < 1000; ++i) {
        std::vector<std::string> duplicate = cheese;
    }
    return 0;
}
```

This code takes 0.487 seconds to run on my machine. However, if we use move semantics:

```cpp
int main(int argc, char** argv) {
    std::vector<std::string> cheese(10000);
    for (int i = 0; i < 10000; ++i) {
        cheese[i] = "Doodles " + std::to_string(i);
    }
    std::vector<std::string> duplicate;
    for (int i = 0; i < 1000; ++i) {
        if (i % 2 == 0) {
            duplicate = std::move(cheese);
        } else {
            cheese = std::move(duplicate);
        }
    }
    return 0;
}
```

This code takes around 4 milliseconds to complete on my machine.

What is the difference? At a very superficial level, both pieces of code transfer the contents of one vector to another. However, how they do it is different and what is done to the source is different. In a full copy (the first example), after

```cpp
std::vector<std::string> duplicate = cheese;
```
both “duplicate” and “cheese” will contain 10000 strings. However, when you perform a move (the second example), after

```cpp
duplicate = std::move(cheese);
```

only “duplicate” will have 10000 strings while “cheese” will have 0 strings. The code effectively moved the contents of one vector to the other vector (assigning a pointer!). As a result, the code runs a lot faster as it did not need to allocate the memory for 10000 strings at each iteration.

If you are interested, you should look into copy/move operators for classes as well as l-value and r-value references. I will not talk about them as they are not used in the code.

## 4 Smart Pointers

By now, you should be familiar with what a raw pointer is. You can recognize a pointer as it uses an asterisk. For example:

```cpp
int* i_am_a_pointer = new int();
delete i_am_a_pointer;
```

Using a raw pointers can result in many issues if you are not careful. If you forget to free the memory, you have a memory leak. If you free the memory more than once, you will segfault. As a result, smart pointers were introduced to fix this problem and introduce an explicit concept of “ownership” into the C++ language. Then, when noone “owns” the pointer anymore, the memory will automatically be de-allocated for you. However, note that storing both smart pointers AND raw pointers in your code will cause headaches.

### 4.1 Unique Pointers

Unique pointers are great because they maintain the performance of a raw pointer while still giving a clear concept of “ownership.” With a unique pointer, only one entity may own the pointer at any given time. In C++, the “std::unique_ptr” template class provides an implementation of a unique pointer. Therefore, the following code is invalid:

```cpp
std::unique_ptr<int> i_am_a_pointer(new int);
std::unique_ptr<int> another_pointer = i_am_a_pointer;
```

Your compiler will complain because you are trying to make a copy of a unique pointer! But only one entity can own a unique pointer at any given time! So what you need to do instead is to transfer ownership using the move semantics described in the previous section:

```cpp
std::unique_ptr<int> i_am_a_pointer(new int);
std::unique_ptr<int> another_pointer = std::move(i_am_a_pointer);
```

After this code is run, “i_am_a_pointer” will no longer be a valid pointer while the memory pointed to be it will now be pointed to by “another_pointer.” After the pointer goes out of scope, the memory will be automatically de-allocated.

### 4.2 Shared and Weak Pointers

A single-entity ownership concept is not always applicable though and that is where shared pointers come in. Shared pointers are implemented in C++ by the “std::shared_ptr” template class. Unlike unique pointers, multiple people can “own” a shared pointer. Therefore, the following code is valid:

```cpp
std::shared_ptr<int> i_am_a_pointer(new int);
std::shared_ptr<int> another_pointer = i_am_a_pointer;
```
Only when both shared pointers go out of scope will the memory be de-allocated. Great! Let’s use shared pointers everywhere then! Not so fast! How shared pointers accomplish this convenient behavior is that it allocates a counter (among other things) which is then incremented and decremented as shared pointers are created/destroyed. When the counter hits 0, the memory is deleted. This introduces a slight overhead which may or may not be desirable. One way to get around this is to store weak pointers instead. A weak pointer will not “own” the memory and so will not count towards the counter. To get back to the shared pointer (which you must do before you use weak pointer), you must “lock” the weak pointer. If the shared pointer returned by the lock function is invalid, then you know that the memory was already deallocated.

```cpp
std::shared_ptr<int> i_am_a_pointer(new int);
std::weak_ptr<int> a_weak_pointer = i_am_a_pointer;
std::shared_ptr<int> use_me = a_weak_pointer.lock();
if (use_me) {
    // you can use the point use_me here.
}
```

Oftentimes though, it makes more sense to store shared pointers in classes. However, you will want to avoid this unnecessary copying of shared pointers if you pass them to functions. As easy way to get around this is to just pass the shared pointers by const reference! For example,

```cpp
void UseSharedPointer(const std::shared_ptr<int>& pointer) {
    // use pointer here.
}
```

You will see some blatant copying of shared pointers in functions in the framework code and that is a mistake and should be fixed (some day...). Finally, performing move semantics on shared pointers will “move” the data. The source pointer will no longer be valid while the new pointer will have the same data as the old pointer. An advantage here is that when performing a move on a shared pointer, the counter will not be changed.

### 4.3 More on Smart Pointers

Obviously, this is not a complete introduction to smart pointers. If you want to learn more I would suggest reading “Modern Effective C++” by Scott Meyers.

### 5 Feedback

If anything in this document is unclear, please post a question on Piazza under the c++_primer folder!