DSOOP Midterm Review

April 15, 2019

1 Classes

A ***class*** is a collection of related functions centered around an *object*, and operates on *member functions* , which operate on items within that object.

Each class has to have an *access specifier*, which denotes how the member functions of the class can be accessed by other member functions. Functions and variables declared as `public` can be accessed from anywhere else in the program; from `main` for example.

A special member function, called the *constructor*, serves to initialize values every time a new instance of the object is started. It is the first function called when the object iniitializes. There can be several different types of constructors, able to initialize the values in different ways.

Members labelled `private` are not accessible to anywhere else in the program other than the class definitions (and friend functions). Usually, data members (variables) are labelled private and member functions are labelled `public`. Labelling everything as `public` can lead to other parts of the program acessing our class's data members, which we don't usually want changed.

We can define the member functions outside the main class declaration. Common practice is to have another file called `foo.cpp`, where we include the header file and define the functions. When we do this, we have to declare the class name followed by two colons '::', to indicate that the current function belongs to the specific class. Otherwise, the compiler will treat these funcitons as regular, non-member functions.

Programming around classes is useful because we can separate interface from implementation, that is, the user does not need to know how the function works in order to use it.

The compiler will only store copies of the data members for each function, the classes will all use the same copies of the function declarations to improve performance.

We use a dot operator(.) to access member functions and values, the arrow operator(-¿) is used when we have a pointer and we want to access member values.

It is usually not best practice to define member functions in their declarations. Clients of the class will be able to see the implementation and will have to

recompile the entire program if something changes. The simplest and functions unlikely to change mat be declared in the header.

At the beginning of the header files, we should put *include guards*, or pieces fo code that ensure that the header files are not included twice in our program. Because multiple source files may include the same header files, if our program includes the same header file twice , it will cause a compilation error because it already say the function declaration.

A common type of function found in classes are *access functions* and *predicate functions*. Access functions allow us to retrieve the values of data members. Predicate functions allow us to test the truth or falsity of conditions. For example, a function like `getValue()` retrieves a value, and function `isEmpty()` might be a boolean function to ckeck if a data object is empty.

Destructors are called byt the program whenever it has to destroy an object. They do not have a type and do not return anything. For local variables declared in regular functions and data members, the destructor is called when the executin reaches the end of that block, i.e. when the function ends executing.

By decalring data members as private, we forbid the user from changing the values directly (maybe by accident). However, we can decalre *set* and *get* member functions to interact with the values directly,

Returning a reference to a private data member

A very dangerous practice is returning a reference to a private data member. Returning a such reference may cause the value in the class to be changed, because the value may serve on the left side of the equation. Thus returning a reference to a private data member may break the encapsulation and cause a private data member to be changed from outside the class implementation,

To pass an object to a function, the default usage is to pass by value, where the compiler will create a copy of the object to pass. This will create a copy of the object every time we want to pass it to a function. Passing by reference will increase performance, but it might lead to the passed object being changed in the function. A safe alternative is passing by const reference, where we pass an object as a reference but declare it const. This provides the performance benefit of passing by value, but elminates the risk of changing the object.

2 Classes: A Deeper Look, Part 2

2.1 Const Member Functions and Objects

When we want to declare a const function, we use the `const` keyword to specify that it is not possible to change the value. We can't call for `const` objects unless the function itself is declared as const. Const declarations may also increase performance, since the compiler may perform its special optimizations on const objects. Const and non-const versions of functions may be overloaded and the compiler will determine which version to call depending on whether the object is const or not.

Constant object will not be able to call non constant functions! Else, there will be a compiler error because const objects nee to be called by const functions. When declaring a const object or variable, we cannot set it equal to any other value, so we must initialize it at the beginning. We then will not be able to change its value,

2.2 Composition: Objects as Members of Classes

We can declare some objects to be members of other classes (like the vector columns of matrices in the homework), and this will give us access to the composed objects in the greater class. Composition is often referred to as a *has-a relationship*, because there is one part which is composed of another but they are not equal.

For example, we can have the constructor accept a parameter of a type that's another object. Then, when we initialize the objects in the constructor, the constructor for the included class will be called as well.

2.3 Friend Functions

Friend functions are functions that are not a part of the class itself; they are not member functions; however, they can still access the public and private members of the class in which they are declared.

Classes can also be declared as friends. In this case, the friend class can freely have access to the other class and its member values and functions.

2.4 The `this` pointer

Every object has a pointer which points to its own address. This means we always have a pointer such that

```
Class_name *ptr = Calling_Class_Obj
```

This means that we always have a way of addressing our own values if we are dealing with values that have the same name.

That's nice and all (noice), but what else can we do with the `this` pointer? We can allow for **cascaded function calls**, which allow us to perform multiple

funciton calls at once. This took me a while to fully understand, so here it goes: When we return a reference to the same object, we are in a sense making it so that the next function call will be declared on the same object. The thing calling the next object will be an object of the same type (actually it will be a reference, but it will still modify the original value).

2.5 static class members

The keyword `static` usually refers to a a variable we only want to keep one instance of. For example, if we want to have a varaible that counts the amount of object instances, we could decalre a static variable in the class declaration. This will make the varaible be kept in the heap, where other static variables are held.

Static member values and member variables can be referenced even when there are no objects that have been declared.

3 Operator Overloading

For custom defined objects, we can define the functions of several operators to perform a certain way. For user-defined objects, we can also specify how the compiler is supposed to treat a call to an operator when it sees one, depending on the parameters we wish to use.

To mimic the regular functionalty of the operator should be a no-brainer when overloading the operators.

However, opeartor overlaoding should be done with caution, especially in the case that dynamic memory is being used or other types of pointers. We can only modify the behaviors of the operators with certain objects; we cannot change the fundamental procedure.

Interesting: Any overloaded operator requires that the call to it includes a user defined type. This is a continuation of the idea that the operator cannot change the fundamental procedure on fundamental types.

If we want to declare a global overloading function, for performance reasons we could declare it as a `friend` function. If we wanted to do so, we would have to declare the function as a friend function in the class/object declaration.

Careful: If the first argument to the operator function is an object of another class, then the funciton must be declared as a friend function, and the object itself must be the argument. Why? If we wanted to represent the overloaded function call, it woule be, for example:

```
operator<<(obj.name, cout);
```

This would require the programmer to write:

```
Obj.name<<cout;
```

While a declaration like this is possible, it would require the programmer to pay extra attention to this specific type of operator. For convencience, we would declare input and output functions as friends.

3.1 Overloading Stream Insertion and Extraction Operators

To overload `<<` and `>>` operators, we need to declare the overloaded functions as friend function. However, if we want to enable cascading, we should return a reference to either `istream` or `ostream`, depening on the function call. Therefore, a prototypical output function might look like

```
ostream &operator<<(ostream& os, const &Obj.name ){
/*print whatever you want bro it's your life*/
return os;
}
```

This will ensure that we can call multiple input or output functions one after another, since the type returned will be the same type required to call the next function, or a reference to input or output stream.

Here, we are returning a reference, but why is it not dangerous? The reference to `istream` and `ostream` are long-lived, so we are not really returning a reference to an automatic variable.

Const vs. Non Const Reference: If we have an operator overloading function, we need to decide if we want to return a const reference or not. Usually, for subscript operators, we will need to define two different functions: one for when the object is an rvalue and one where the returned value is to serve as an lvalue. If the value returned is to serve as an rvalue, then we would be able to use the value as an assignment to another value or variable. Else, we could change the value in our function because we could reassign the value the reference points to.

3.2 Dynamic Memory Management

With C++ (and C, but differently) we can create pointers to values or objects and arrays at runtime, depending on how we need the values. These values are set with the keywords `new` and `delete`.

The new keyword will return a pointer to the object we have just returned.

Then, whenever we don't need to use te memory anymore, we can deallocate the memory with `delete`. Since the memory is located on the heap, there is only a limited amount of memory available. If memory was not succesfully allocated for the object, an exception is thrown

If the memory released, there will be a *memory leak*, which one time crashed my desktop :(.

To allocate dynamic memory for fundamental types and arrays, the declaration would look something like:

```
double *ptr = new double(...);
int *arr = new int[size];
delete ptr;
delete arr[];
```

The last two lines show the statement for deleting dynamically allocated memory.

Remember to always pass the object by reference to the copy constructor, otherwise the assignment will result in infinite recursion. (My favorite type of recursion though)

If our class contains pointers to dynamically allocated memory, we really need to define a copy and assignment operator, because we would need to create new dynamically allocated values for our current class.

3.3 Converting Between Two Types

Sometimes, we would need to declare an operator to cast an object from one type to another; however, we would need to define the operator for this specific conversion.

3.4 Increasing INcrement and Decrement Operators

If we want to define the postfix and prefix increment operators, by differentiating their signatures we could tell them apart. **The prefix versions of the increment operators are overloaded in exactly the same way!**.

With the postfix operator, we would have to pass another int argument, usually set to 0 and called a *dummy variable* .

An implicit conversion would be one where we convert from one type to another and we want to carry out the assignment in that manner; if we use the keyword `explicit` this is *explicitly* not allowed :).

4 Object-Oriented Programming: Inheritance

Suppose we have two classes. If one class is contained in another class (for example a shape contains a circle), we can specify one class to inherit the functionality of the other class. Unlike composition, which establishes a has-a relationship, we now establish a "is-a relationship." For example, a circle *is* a shape, and so any characteristic that a shape posseses, a circle should also posses.

One class can also inherit from multiple classes. Derived-class objects *are* base class objects; however base class objects are not base-class objects.

Whenever we have a class inherit from another class, we have to specifically call the base-class constructor in our derived class constructor, since our derived object should also possess the functionality and values of the base class.

Protected Inheritance will allow our derived class objects to access base class objects. However, to encourage proper software practice, we should use get and set functions and leave the optimizations to the copmiler.

5 Object-Oriented Programming: Polymorphism

As the name implies, a polymorphic function can have multiple implementations depednding on the type of object. Polymorphism can be implemented mostly through base class pointers to derived class objects and a special type of function declaration in the base class.

When we have a base class pointer pointing at a derived class object, and we want to execute a function exclusive to the derived class, there will be an error. This is because in this situation (i.e. without virtual functions), ,we would be calling the base class version of the function using the derived class data; however, since there is no such function in the base class, there will be an error when trying to call the function.

To avoid this error, we can *downcast* the base class pointer to an object of the derived class.

virtual functions: If in the base class we declare a function to ve `virtual`, then we can oberride its functionality in the derived class declarations, so that the derived class implementation executes.

5.1 Abstract Classes and Pure Virtual Functions

Abstract Classes are classes in which we don't really instantiate any object, but we use it as a base class to derive classes from. If we declare one or more pure virtual functions, then the class is an abstract or pure virtual class.

We declare a pure virtual function by adding a "=0" *pure specifier*.

```
virtual int foo() =0;
```

Then what is the difference between a regular virtual and a pure virtual function? Well with a regular virtual function, we give each derived class the option of overriding the function declared in the base class; however, with a pure virtual function every derived class is required to override the function, since it doesn't have a "default" base-class implemenatation in any case.