DSOOP Midterm Review

April 13, 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.