Classes in C++

Picked from Lippman 5th edition

Basic Syntax and Concepts

- Member functions are defined similar to regular functions, but the declarations must occur inside the class definition.
- The functions may be defined outside, and those defined inside the class body are implicitly `inline`.
- Member functions called inside other member functions have this access through the `this` pointer.
- It is illegal to define a parameter or a vriable named `this`.
- The type of `this` is a const pointer to the nonconst version of the class type.
- This means that in general, we cannot call an ordinary member function on a const object.
- The way to indicate that the `this` pointer needs to be const is adding `const` in between the parameter list and the function body. For eg:

```
double Sales_data::avg_price() const {
    ...
}
```

- For defining a class member function outsude the class, you need to validate it using the scope operator, `class_name::func()`.
- To return the object itself, you should `return *this`.
- One can also return an lvalue reference to the object.
- Non-member functions that are part of the interface are usually declared in the same header.
- Class object initialization is controlled through constructors. Constructors have no return type, and have the same name as the class.
- Constructors may not be declared const, as they invariably need to change the object.
- A default constructor is one that takes no arguments.
- The compiler supplies a special default constructor, the synthesized constructor, which is defined implictly if no constructor is defined explicitly by us. It works as:
  1. If there is an in-class initializer, use it.
  2. Otherwise, deault-initialize the member.
- If any constructor is defined explicitly, the synthesized constructor flies out.
- If a class has a member which is another class, which does not have any default constructor, the compiler cannot synthesize a constructor implicitly.
- If we want to define a default constructor which is the same as the synthe-

sized one, and more constructors, we can write `Sales_data() = default;`. This can appear with the declarations inside or as a definition outside.
- The constructor can explicitly initialize the members instead of just assigning them, though the constructor initializer list.
- The constructor initializer list is a list of member names each followed by its initialized value in parantheses, separated by commas. It appears between the parameter list and the body. For eg:

```
Sales_data(const std:string &s): bookNo(s) {}
```

- The constructor's initializer list wins over te in-class initializer if both are present.
- It is usually better to use in-class initializer instead.
- Classes also define what happens on copy, assignment and destruction.
- Classes that deal with pointers, or dynamic memory, would need to redefine all of the above, and probably cannot use the synthesized versions.
- C++ uses access control specifiers to enforce encapsulation.
- struct deafults to public where as class defaults to private.
- A class can allow another class or nonmember function to access its nonpublic members by declaring them friends, basically by writing out the prototype preceded by the keyword `friend` in the class definition. For eg:

```
friend Sales_data add(const Sales_data &, const Sales_data &);
```

- The friendship is not affected by the access modifier of where it is declared.
- We also need to declare the function outside separately, a friend declaration is not a declaration.
- A class can also define local names for types, using `typedef`. They should appear before they are used.
- For a function defined outside, we can declare it inline either during the declaration or during the definition outside.
- It is better to define inline functions in the corresponding class header.
- Member functions can also be overloaded.
- Some datamembers must be allowed to change, even inside a const object. Such datamembers need to be declared `mutable`.
- In-class initializers could also be brace initialized in nature.
- Functions that return a reference to `*this` allow for chaining of functions on them one after the other.
- We can overload a function based on just the constness of the `this` pointer. This will resolve by const objects calling one version and nonconst ones the other.
- Two classes with the same contents still define two different unrelated types.
- We can declare a class without defining it, by just writing `class Screen;`.
- This makes `Screen` an incomplete type. We can only define pointers or references to such types, and declare functions that use these.
- Data members of a class type need to have that class defined before they themselves are specified.

- We can declare an entire class as a friend, allowing all member functions of that class access to all of our members.
- We can also make a member function of another class a friend of ours, by scoping the function during the friend declaration.
- This requires a careful ordering:
  1. Define the other class and declare the function.
  2. Put a friend declaration for that member function in this class.
  3. Define the member function outside.
- Each class defines a scope, called the class scope.
- During a function definition outside the class, the return type comes before the function name, so if the return type is of a type defined by the class, you need to scope the return type also. This need not be done for the parameters.
- Compilers look at classes in two phases. First all declarations are compiled, then all definitions (function bodies).
- Names used in declarations still need to occur before they are used.
- Name lookup thus proceeds as:
  1. the part of the function code seen yet.
  2. The whole code of the class definition.
  3. Surrounding scopes.
- We can override 1 and get to 2 by using scoping.
- The constructor initializer is needed for types which need to be initialized distinct from assignment. This is true for const types and reference types.
- The order of member initialization is not the same as that specified in the constructor initializer list, but rather follows the order in which they appear in the class definition.
- This becomes important when one datamember is initialized in terms of others.
- A constructor with default arguments for all parameters also ends up defining the default constructor.
- You could also allow constructors to delegate initializations to other constructors, by syntax like `Sales_data() : Sales_data("", 0, 0) {}`. This calls another constructor for the initialization.
- The code of the delegated-to constructor also runs, not just the initializer list.
- Default initialization happens when:
  1. nonstatic variables or arrays are defined without initializers at block scope
  2. when a class that itself has a class type member uses synthetic constructor
  3. when members of class type are not initialized in a constructor initializer list
- Value initialization happens when:
  1. Array initialization provides too few initializers
  2. When we define a local static without an initializer
  3. When we explicitly request so by writing `T()`.

- Note that `Sales_data obj()` defines a function and not an object.
- A constructor with a single argument also defines a conversion from that type.
- During function calls, only one class-type conversion is allowed.
- We can suppress these implicit conversions by adding `explicit` in front of the function declaration inside the class.
- We can still use these functions for conversions if we force the conversion explicitly.
- An aggregate class is one which has all data-members public, does not define any constructors, has no in-class initializers and has no base class or virtual functions.
- You can initialize objects of an aggregate class using brace initialization, the initializers must appear in order of declaration of the data members.
- A literal class is one whose objects can be used in `constexpr` type expressions.
- Such classes may have member functions that are `constexpr`.
- An aggregate class with all datamembers of literal types is a literal class. A nonaggregate class can be literal if:
  1. The datamembers have literal type.
  2. The in-class initializers must be a constant expression.
  3. The class must have atleast one `constexpr` constructor (which is usually empty).
  4. The class must use the default destructor.
- A datamember associated with the class itself and not inidividual objects is called static.
- To declare this add keyword `static` in front of the datamember declaration.
- We can also have static member functions, which will not have access to the `this` pointer. For the same reason they cannot be declared `const`. We can still refer to static datamembers inside the body.
- Static member functions can be defined like normal member functions. But static datamembers need to be defined outside the class explicitly, like how class member functions defined outside the class look like. For eg.

```
double Account::interestRate = 5;
```

- The definition can occur only once, so it is best to put this line in the file with the definitions of the member functions.
- Constant integral static members can have in-class initializers, and may skip the definition, but this is not good practice. If an in-class initializr is given, the definition must not have any initializer: `const int Account::period;`
- A static member has lesser restrictions, as in it can have an incomplete type and it can be used as the default argument in member functions.

Copy Control

- Consists of 5 elements:

1. Copy constructor
2. Move constructor
3. Copy assignment
4. Move assignment
5. Destructor

- The copy constructor is used to initialize another object of the same class. It has the prototype `T(const T &)`, although the const is not really needed.
- The synthesized copy constructor memberwise copies the object. Member of class type are copied through thier copy constructor, built-in types are copied directly.
- Arrays are copied element wise.
- Copy initialization happens at:
  1. defining variables using the `=` syntax.
  2. passing an object as argument to nonreference type parameter.
  3. return from a nonreference type function.
  4. brace initialization of the members of an array/aggregate class.
- Also used inside vector and such containers when using vaue initialization.
- The reason that the parameter of the copy constructor must be a reference is because otherwise the copying there would call the copy constructor again (recursion).
- Note that in cases like `T a = b` where `b` is of type `S`, the copy constructor will work, but we also need the constructor of type `T` from type `S`, and it should not be `explicit`.
- The overloaded assignment operator has the prototype `T& operator=(const T &)`, where again the const is not really needed.
- The this pointer binds to the lhs in this scenario.
- The synthesized copy assignment does assignment on each member.
- The destructor has the prototype `~T();`, and in the destructor the function body is executed first and then the members are destroyed in reverse order of their initialziation.
- Note that destructor of the pointer type does not call delete on it.
- Destructors are called when:
  1. variables are destroyed or go out of scope.
  2. members of a larger object are being destroyed as part of that destructor.
  3. elements when the container is destroyed.
  4. delete on dynamically allocated object.
  5. temporary objects at the end of the expression when they got created.
- The destructor does not run when the pointer goes out of scope.
- The synthesized destructor essentially has an empty body.
- The rule of three/five states that if you need one of these different from synthesized, you probably need them all.
- You can use the `=default` for any of the copy control members.
- If we want to explicitly avoid copies, simply not defining does not work, synthesized versions still exist.
- We can instead write out the function prototype and then follow it with

`=delete`.

- The `=default` can occur outside the class body, but the `=delete` needs to occue=r on the first declaration.
- Never try to delete the destructor, then you want to be able to define objects or call `delete` on dynamically allocated versions of that object.
- The compiler might delete some of the copy control members instead of synthesizing them, if that does not make sense:
  1. if a member's destructor is inaccessible, the synthesized destructor is deleted.
  2. if a member's destructor or copy constructor is inaccessible, the synthesized copy constructor is deleted.
  3. same for the assignment, and it is also deleted if a member is const or reference type.
  4. the default constructor also gets deleted if any member's destructor is deleted, or a reference member has no in-class initializer or has a const member whose type does not define a default constructor and does not have in-class initializer.
- The same behavior as `=delete` could be achieved by making the members as private and then not defining them.
- Many library functions would assume the presence of a swap function, which usually looks like:

```
class T {
    friend void swap(T &, T &);
};
inline void swap(T &lhs, T &rhs) {
    ...
};
```

- Remember that for using `swap` functions inside `swap`, it is better to put a `using std::swap` at the top and then use just `swap`. This makes sure the library `swap` is not used if you defined your own version.
- A good idea often used is to write

```
T& T::operator=(T rhs) { // rhs passed by value here
    swap(*this, rhs);
    return *this;
}
```

Move semantics

- Some objects are deleted right after copying them, so we need a semantics to allow "moving" objects.
- To support this, the standard came up with rvalue references.
- Lvalue references cannot be bound to objects that require a conversion, literals, and rvalue expressions. Rvalue references can bind to all these but not lvalues.

- The syntax for declaring an rvalue reference is `int &&r = ...`.
- Subscript, dereference, assignment, prefix inc/dec and lvalue return function calls all result in lvalue expressions.
- Since rvalues are ephemeral in nature, we know they are about to be deleted.
- All variables, including rvalue references themselves, are lvalues as expressions.
- Calling `std::move()` on an lvalue tells the compiler to treat it as if it were an rvalue.
- After the move, we can only use the variable to assign a new value to it (or perhaps destroy it).
- To allow classes to use advantage of this, we need to define the move constructor and the move assignment operator.
- The move constructor and assignment operator need to ensure that the moved-from object is safe to be destroyed.
- The library does extra work if our move constructor (or assignment operator) can throw, so if it cannot, we should declare it as `noexcept`, by writing the keyword after the parameter list (before the constructor initializer list if any).
- We should specify `noexcept` on both declaration and definition.
- An example of this fact is that when `push_back` is called on vector and it runs out of memory, vector will move the objects only if they are sure to not throw any exception in that process.
- A moved-from object can be in any valid destructible state.
- If any of the destructor, copy constructor and copy assignment operator are defined, the compiler will not synthesize the move constructor and assignment operator.
- The synthesized move constructor and assignment operator are synthesized only when none of the copy-control methods are defined and all data members can be move constructed or assigned respectively.
- The move constructor or move assignment operator are usually not synthesized as deleted, to allow fallback onto copy construction and assignment. However if you explicitly demand through `=default`, then they will be synthesized as deleted.
- If you define the move constructor or assignment operator, the corresponding copy operations are synthesized by default as deleted.
- In general, rvalues are moved and lvalues are copied. If move operations are not defined, function matching will fallback to the copy format for rvalues.
- Interestingly, the copy-and-swap mechanics of the assignment operator can also handle the move semantics and copy semantics in one go.
- Other member functions can also benefit from rvalue and lvalue versions.
- We can also version functions based on wether the object on which they are called are lvalue or rvalue, in the same way we do for const. For eg.

```
Foo sorted() const &;
```

```
Foo sorted() &&;
```

- There is no default in these, which means if you add a reference ualifier in one version, add a qualifier in all versions of that function.