Basic Class Design

Goal of OOP: Reduce complexity of software development by keeping details, and especially changes to details, from spreading throughout the entire program.

Definitions

- Client Code - the code that *uses* the classes under discussion.
- Coupling - code in one module depends on code in another module

  Change in one forces rewrite (horrible!), or recompile (annoying), of code in the other.

Two of the Four key concepts of OOP

- **Abstraction -** responsibilities (interface) is different from implementation

  A class provides some services, takes on some responsibilities, that are defined by the public interface. How it works inside doesn't matter. Distinguish between interface and implementation.

  Client should be able to use just the public interface, and not care about the implementation.

- **Encapsulation -** guarantee responsibilities by protecting implementation from interference

  Developer of a class can guarantee behavior of a class only if the internals are protected from outside interference. Specifying private access for the internals puts a wall around the internals, making a clear distinction between which code the class developer is responsible for.

- **Both are ways of decoupling the client code from the class implementation:**

  Don't need to know about the implementation;

  Can't accidentally interfere with it or depend on it.

Guidelines for Designing Individual Classes

- **"Concrete" classes - no inheritance or polymorphism involved.**

  Objects interact with each other, contain each other, refer to each other.

  Main program causes initial objects to be created, delegates work to objects as needed.

- **Two kinds of classes:**

  Objects that are in the problem domain.
  - Rooms, Meetings, Persons

  Objects that support the other objects.
  - E.g. String, Ordered_list

- **Design a class by choosing a clear set of responsibilities**

  Make classes responsible for working with their own data.
  - Main code should delegate the work down into the classes that have the information.

  Domain classes should be based on actual domain objects.
  - What kinds of objects are in the domain?
    Which classes -
  - What characterizes each domain object?
    Members -
  - How are different kinds of objects related to each other?
    Inclusion versus association -
    Part-of relation versus "using" or "interacts with"
    Relative lifetimes -
    Do they exist independently of each other?

If class responsibilities can't be made clear, then OOP might not be a good solution
- Lots of problems work better in procedural programming than in OOP, so there is no need to force everything into the OO paradigm.
  Making this distinction is critical to understanding the difference between traditional procedural programming and OOP.

Beware of classes that do nothing more than a C struct.
- Is it really a "Plain Old Data" object, like C struct, or did you overlook something?
  If it is a simple bundle of data, define it as a simple struct.
  If there are functions that operate on the data, maybe they should be member functions, and maybe these objects really are responsible for something.

- **Principle: Avoid heavy-weight, bloated, or "god" classes - prefer clear limited responsibilities.**

  If a class does *everything*, it is probably a bad design. Either you have combined things that should be delegated to derived classes or peer classes, or you have misunderstood the domain.

- **Make all member variables private in each class**

  Concept: Programmer of a class has to be able to guarantee that class will fulfill its responsibilities - do what he/she says it will do.
  - encapsulation - making member data private- is the basic step that makes this guarantee possible - prevents other code from tampering with the data.

  No public member variables.

  Beware of get_ functions that return a non-const pointer or reference to a private member variable - breaks the encapsulation!

  Be careful if it is necessary to return a non-const reference or pointer to an item in a container member variable - even if client can't alter the container, might be able to alter the item in a way that violates design intent - or disorder the container!

- **Put in the public interface only the functions that clients can meaningfully use.**

  Reserve the rest for private helpers.

  Resist the temptation to provide getters/setters for everything.

- **Friend classes and functions are part of the public interface, and belong with the class.**

  Friend class or function is part of the same module or component.

  Most clear if declaration and implementation is in the same .h and .cpp files.

  A class developer should declare a class or functions to be a friends only if he/she/they are also responsible for, and have control over, that class or function.

- **Make member functions const if they do not modify the logical state of the object.**

  Use mutable for those occasions where a strictly internal implementation state needs to be changed.

C++ Facilities to Support Class Design

- Compiler will not allow client code to access private members of a class;

  Only member functions or friend are allowed to do so.

  Enforces separation of public interface from private implementation.

- Constructor functions allow a class to be responsible for its initialization.

  Compiler guarantees that constructor will be called when the object comes into being.

Compiler will generate a constructor for you if you don't define one:
- It will call the constructors (if any) for all of the member variables.
- Built-in types have no constructors (or you can say they have a constructor that does nothing).

If you do define a constructor function, the compiler will call the default constructors for all class-type member variables even if you don't explicitly initialize them.
- Use the constructor initializer list to avoid inefficient default initialization followed by assignment in the constructor body.

If construction fails for some reason (e.g. no memory, invalid initialization data) throwing an exception is almost always the correct thing to do.
- You don't want "zombie" objects that exist but aren't really alive.
  Avoid designs in which you have to test an object to see if it really got initialized correctly.
- Throwing the exception means you get out of the context in which the object is being created, meaning that the object automagically does not come into existence.
  "new" operator will automatically deallocate any allocated memory as a result of the throw
  a stack object will automatically be deallocated as we leave the scope of the local variable.

- Destructor functions allow a class to be responsible for cleaning up after itself.

  Compiler guarantees that destructor will be called when object goes out of scope or is deallocated.

  Compiler will generate a destructor for you if you don't define one:
  - It will call the destructors (if any) for all of the member variables.
  - Built-in types have no destructors (or you can say they have a destructor that does nothing).

  If you do define a destructor function, the compiler will still call the destructors for all class-type member variables - you don't have to arrange for their destruction (and normally should not).

- Copy constructor and assignment operator functions allow a class to be responsible for how it is copied.

  Compiler guarantees that they will be called when copy or assignment is done.

  Supports using user-defined types just like built-in types.
  - E.g. as call-by-value function parameters and returned values.
  - Initialization:
    Thing my_new_thing(old_thing);
  - Initialization with "=" as in
    Thing my_new_thing = old_thing; // this is NOT an assignment statement

  Normal declarations:
  - Classname(const Classname&); // copy ctor
  - Classname& operator= (const Classname&); // assignment operator

  Compiler will generate a copy constructor and assignment operator for you if you don't define them.
  - It copies each member variable, using its copy constructor or assignment operator.
  - Note that if member variable is built-in type, the copy is simply of the value of the member variable (or you could say the copy is "dumb" or "shallow").

  Unlike regular constructor and destructor, if you do define the copy constructor or assignment operator, you are responsible for doing the copy or assignment of *all* of the member variables - the compiler will not try to guess which ones you want done.