- **Basic OOP Concepts**

Introduction

- **Goal of OOP: Reduce complexity of software development by keeping details, and especially changes to details, from spreading throughout the entire program.**
- **This presentation assumes "Basic Class Design" presentation.**

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.

Four key concepts of OOP

Abstraction - responsibilities (interface) is different from implementation

- Distinguish between interface and its behavior vs. the implementation and its details
- A class provides some services, takes on some responsibilities, that are defined by the public interface. How it works inside doesn't matter.
- 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, and what the client developers are responsible for.

Inheritance - share interface and/or implementation in related classes

- Express commonalities between classes of objects by arranging them into an inheritance hierarchy. Allows functionality shared between classes to be written/debugged/maintained in one place, the base class, and then reused in descendant classes (shared implementation).
- More importantly, allows client code to interact with classes in the hierarchy through an interface specified in the base class of the hierarchy (shared interface).

Polymorphism - decouple client from exact class structure

- Allows client code to use classes in an class hierarchy in a way that depends only on the public interface of the base class. Derived classes will implement this interface differently, but in ways compatible with the shared public interface.

General Approach for Designing Classes

- **Do the class design work at the level of the public interfaces, not the private implementations.**
  - *Don't get bogged down in implementation details like "I can do with this with a map container and a deque!"*
  - *Think only about what the class responsibilities are and what they do in their public interfaces:*
    - Class X is responsible for …., class Y for ….
    - When an X object needs … it calls the public member function … of the appropriate Y object with … as parameters, which returns …
  - *Try writing pseudo code just for the interactions between class objects through their public interfaces.*
  - *Keep this up until you can't stand it any more, then make implementation choices and write the code.*
- **Continue design thinking until you have thought of at least two reasonable ways to solve each design problem.**
  - *"Reasonable" here means "not obviously stupid."*
  - *Don't just jump on the first design you think of and hack it out.*
  - *All designs are imperfect - they all involve trade-offs. They are good in some ways, bad in others.*
  - *A good design is good in the most important ways, and bad in the less important ways.*
  - *But there might be more than one good design - just different in the specific tradeoffs.*
  - *You can't make an intelligent choice if you have only thought of one design - there could be another, better, simpler one.*

A couple of General DO NOT rules

General Don't: Don't overengineer -

Overengineering - a more complex solution than necessary.

- Do simple things in a simple way.
- Just getting complex without thinking through what the responsibilities of the classes really are results in misdelegating, misassigning responsibilities - the result is simply unnecesssarily complex.

Often a result of anticipating future needs inappropriately.

- "Yes, it is more complex, but if I do it this way, then in the future, it will be easier to do yada-yada." - but will this be needed?
- Problem: the code is harder to work with NOW, and you don't actually know whether you will need to do the future thing.
- "YAGNI" principle - "you aren't going to need it".
- Wisdom of the gurus: If the code is a simple solution that is clear and well-designed, it will be easy to change it in the future if necessary.
- So design and code a current solution well, instead of making a mess trying to anticipate an unknown future.

General Don't: Don't create 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.*

Example: In project 2's restore function, the work of reading and interpreting the data file, and creating the right objects or relationships, was delegated to the Record and Collection classes, which do all the work, and just signal a problem if they can't. Only thing the main module knew is that Record data comes first, then Collection data. If a member variable was added to Record, only the Record class class would need to be changed.

- Contrast with a god-like main module that knows how Records and Collections are structured, and what the details of what data file looks like - it reads the data, validates it, creates objects, and stuffs the data into them from the outside while they just sit there passively. If a member variable is added to Record, both the Record declaration and the main module code would need to be changed.

Guidelines for Designing Individual and Concrete 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.

- Accounts, Customers, Banks, Rooms, Meetings, Persons, Ships, Islands, Resources.

- *Objects that support the other objects.*
  - E.g. String, Ordered_list, map<>, priority_queue<>

- **Design a class by choosing a clear set of responsibilities.**
  - *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.
  - *Symptoms of a lazy class - not being responsible for its own data and computations.*
    - Other code (e.g. main module) has to tell the object what to do and when to do it, or what its variable values should be, when it has the knowledge to do the work itself.
      - E.g. instead of the class updating its own member variables when it does something, other code tells it to do the update.
    - All (or almost all) member variables have both getter and setter functions, which suggests that the client code is doing the work, and the class is just a passive holder of data.
  - *Symptoms of misallocated responsibilities - redundant or misplaced computations.*
    - The class makes decisions or does computations that other code also does.
      - E.g. client makes decisions, then class also makes similar decisions internally.
        - Why doesn't just one of them do it?
    - The class has to figure out what work it is supposed to do when the client code could be more specific.
      - E.g. look at a member function parameter and branch to different code depending on it.
        - Why not separate member functions and the client decides which to call instead of passing a code?
    - The client code is responsible for passing information between objects that don't communicate directly.
      - Might be a "god" module.

- **Principle: Avoid heavy-weight, bloated, or "god" classes - prefer clear limited responsibilities.**
  - *If a class does everything, it is probably a bad design. You may have misunderstood the problem domain. You have probably combined things that should be delegated to derived classes or peer classes, or even to separate objects from a simpler class.*

- *Symptoms of a heavy-weight or bloated class:*
  - A large number of unrelated member functions or variables that don't correspond to any simple concept of the classes responsibility.
  - An inability to summarize the class's responsibility in a short English description.
  - Only one of them gets created, because it does multiple things that would be separate objects if the class was simpler.
  - When a member function is called, you have to include an additional parameter that signals the object what kind of thing to do.
  - If the object is a container of data of some sort, it includes several functions that are so specialized that it couldn't possibly be used in another project that needs that same general kind of functionality.
- *Symptoms of a "god" class:*
  - Doing things to or for other classes that they could be doing for themselves.
    - Common example: Stuffing another object with its data instead of letting it initialize itself from the data source.
  - Everybody's friend: "god" class has to mess around inside everybody else, so they declare it to be a friend.
  - Complete accessors to private members provided for benefit of the "god" class to allow it to see and control everything.
    - Control should stem from the interaction of objects with clear responsibilities, not one object in charge of everything.

- **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 and no protected member variables.*
  - *Beware of get_ functions that return a non-const pointer or reference to a private member variable - breaks the encapsulation!*
  - *Beware of functions that return an iterator to the contents of a container -*
    - Either breaks encapsulation by allowing modifications of private data
    - Suggests that client is doing work that class should be doing - it should be working with its data, not somebody else!
- **Put in the public interface only the functions that clients can meaningfully use.**
  - *Reserve the rest for protected interface or private helpers.*
  - *Resist the temptation to provide getters/setters for everything.*
    - Leads to lazy classes, or "god" classes.

- **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 only for those occasions where the implementation needs to change a strictly internal implementation state that is invisible to other classes - never as a way to fake constness.*
- **Explicitly decide whether the compiler-supplied "special member functions" (the destructor and the copy/move functions) are correct, and let the compiler supply them if so.**
  - *Do not write code that the compiler will supply.*
  - *Unnecessary code is an unnecessary source of bugs.*
- **Try to follow the "Rule of five or zero" - either explicitly declare all five of the special member functions, or declare none of them and let the compiler supply them automatically.**
  - *"Declare" here means to either declare and define your own version of the functions, or declare what you want the compiler to do with =default or =delete.*
  - *If you have to write even one of these functions for some reason, explicitly declare the status of the rest of them to avoid confusion or possible undesired behavior.*
  - *If you have to write your own destructor function to manage a resource (like memory), you almost certainly have to either write your own copy/move functions or tell the compiler not to supply them (with =delete).*
  - *In writing a copy constructor, remember to copy over all member variables - a common error.*
- **If copy or move operations are not meaningful for a class, explicitly prevent the compiler from supplying them.**
  - *For example, to enforce the concept that objects in the domain are unique.*
  - *In C++11, disable compiler-supplied copy and move functions with the =delete syntax.*

Guidelines for Class Hierarchies

- **The base class must represent a meaningful abstraction in the domain.**
  - *At least the interface must be shared by the derived class objects.*
  - *Vehicles - move from place to place, need fuel, etc.*
- **Make base classes abstract - corresponding to the abstraction in the application domain.**
  - *Clearly defines their roles as the root of a class hierarchy (or subhierarchy)*
  - *Clearly shows that the class corresponds to an abstraction in the problem domain.*
  - *Avoids some inheritance oddities from saying is-a with things that aren't really is-a.*

- E.g. If you can instantiate an animal, that means it has the same status as a cat - which can't be true.

▼ *Can approximate making a base class abstract by making its constructors protected instead of public.*

- Derived class constructors can still invoke the base class constructors, but client code can't.

▼ **Most-derived ("leaf") classes should represent concrete objects in the domain.**

- *Car, airplane (fighter plane, cargo plane, etc), ship (warship, cargo ship), submarine*

▼ **Intermediate classes should also represent meaningful abstractions in the domain.**

- *Have at least interface in common*
- *Land vehicles, flying vehicles, floating vehicles, etc.*
- *Make abstract if possible (e.g. with protected constructors if no pure virtual functions).*

▼ **Inherit publicly only to represent the "is-a" (substitutability) relationship.**

- *A derived class can be used in the same way as the base class, because the derived object "is a" base object.*

▼ **Distinguish between the public and protected interface.**

▼ *Public is for client code*

- Functions provide services for client code

▼ *Protected is for derived classes.*

- Functions provide services for derived classes

▼ **Put common functionality as high up in the class hierarchy as it is meaningful to do so.**

▼ *If same code and members appears in two classes, they probably should be in a class they both inherit from.*

▼ Common error: Taking the same member variables alone as indicating common functionality to be placed in a base class.

- Member variables by themselves do not contribute to "common functionality."
- Note that each object has its own copy of all (non-static) member variables, including those in the base classes. So there is no "savings" in memory footprint from having member variables in a base class than in derived classes.
- Symptom of the problem: Strong temptation to make the base class member variables protected, or to provide protected write accessors for them.
- Solution: Focus on the work done by the member functions - place functions (and their related member variables) as high up in the class hierarchy as it is meaningful to do so. If necesary, refactor to support this.

▼ *The derived classes should all make use of the base class functionality; do not move the functionality up any higher than this.*

- Common error: ***Bloated base class.*** A Base class that has functionality that some derived classes do not need. Factor the functionality into additional intermediate classes.

▼ **Let each class in a hierarchy be responsible for itself.**

▼ *Initializes and de-initializes itself.*

- Base or Derived classes, or client, is not responsible for setting up an object if its own constructor can do it.
- Supported by constructor/destructor call sequence performed by the compiler
- Enforced by making the member variables in each class private.

▼ *Does all computations related to its own member variables.*

- Protected or public member functions make them happen, supply outside data.
- Enforced by making the member variables in each class private.
- Derived classes can call functions in Base classes to do relevant work.
- Overriding can allow each derived class to put its own variation on the base class operations,

▼ **To re-use code that is in a class, prefer using a member variable of that type (aggregation, has-a relationship) instead of private inheritance (implemented-with relationship).**

- *Usually produces fewer conceptual and programming problems.*

▼ **If the class might be used as a base class, declare a virtual destructor.**

- *Otherwise, might not be able to use derived class objects polymorphically and have them properly destroyed.*
- *A class hierarchy might have classes at intermediate levels used as base classes in different contexts, so their destructors should be virtual also.*
- *This is easy because if the root base class destructor is declared virtual, then all derived class destructors are automatically virtual as well and do not need to be separately declared.*

▼ **If you want leaf objects to be copied or moved, let the compiler supply the copy/move functions if they are correct. If not, and you have to write your own, they must call the base class copy/move functions to ensure that the base class member variables are correctly copied or moved.**

- *In other words, copy/move of base class members does not happen automatically when you define copy/move for a derived class.*

▼ **If all leaf objects in a class hierarchy should not be copied or moved, prevent the compiler from generating copy/move functions by simply deleting them in the root base class.**

- *It is then unnecessary to delete the copy/move functions in the derived classes - do not clutter derived class declarations with unnecessary deletionsl*
- *This works because if the compiler-supplied default copy or move functions for derived classes will invoke the copy or move function for the base class; if these have been deleted, then the compiler will not create the derived class functions.*

The fundamental OOP technique: A polymorphic class hierarchy.

Fundamental: Use virtual functions instead of "switch on type" logic!

- *"Switch on type" logic in C++ is usually a design failure!*
- *Occasionally it is unavoidable, but almost always is a result of bad design.*
  - Somebody else screwed it up, now you have to deal with it.
- *Virtual functions should be doing the work as much as possible!*

Use inheritance and polymorphism to hide details and changes.

- *Basic concept: Hide present and future differences under a base class, and use polymorphism to get the different behavior controlled by the same client code.*
  - Slogan: Figure out what will vary and hide it from the client!
  - Base class declares virtual functions
    - Base class is usually abstract - only leaf classes should be instantiatable as objects.
    - Each Base virtual function is either pure virtual or provides a "default" definition that Derived classes can use if they wish.
  - Client code refers to all objects with a Base * pointer, calls virtual functions through the Base * pointer to get polymorphic behavior of the objects.
  - Benefits
    - By overriding virtual functions as desired, each Derived class can have a different behavior, in any desired pattern.
    - Client code does not need to know what Derived class the objects belong too - the virtual function calls automagically route the call to the correct version of the virtual function defined by the Class that the object belongs to.
    - Using virtual functions, the client code can use objects whose exact type it doesn't know.
      - Client code needs to know only the Base class declaration - this provides the interface for all of the classes in the hierarchy.
      - Client just #includes Base.h - needs nothing else.
    - Means that Derived classes can be added, removed, modified, without changing the client code that class the interface.
      - If done properly, client code does not even need to be recompiled.

If some functions are implemented only in derived classes, choose a way to access them.

- *"The fat interface problem" - no single good label for the problem.*
  - If the classes in an inheritance hierarchy are not uniform in what functions it makes sense for them to have, you can't choose a set of base class virtual functions that are equally applicable to all derived classes.

- Example: A ship simulation: Have various kinds of ships, etc.
  - All can be told to move on certain course and speed, all have some fuel capacity, maximum speed, etc. Good choices of base class members.
  - Different kinds of warships might respond differently to command to "attack", different kinds of cargo ships to command to load and deliver cargo. But:
    - Should warships be told to load and deliver cargo?
    - Should cargo ships be told to attack?
  - Worst: A submarine is a kind of ship, but has submerge capability. Should there be a "submerge" function in the base class of Ship?
    - Makes no sense for all other kinds of Ships?
- So if some functions are meaningful for only some derived classes, how do we enable the Client to access them?

- *Four solutions:*
  - **1. "Fat interface"**
    - For a virtual function call to reach a function defined in a Derived class, all possible virtual functions must be declared in the base class.
      - What if you have a function declared and defined only in one Derived class? How can you call it given only a Base * pointer?
        - Base declares vf1, vf2 as virtual
          - Derived declares vf1 (for example)
            - DerivedDerived declares vfx
        - using Base * Bp, can call p->vf1(); p->vf2();
        - how can we call vfx? vfx is not known as a member of Base, so call through Bp will fail.
        - Example: Navy simulation
          - Ship class - has navigation functions
            - Warships - have weapons, etc
              - Submarine - can submerge
    - Go ahead and declare vfx virtual in Base and supply a "default" definition there (e.g. does nothing). Base class now has an interface to everything all of the derived classes are able to do - a "fat" interface.
    - No other classes declare vfx. Now Bp->vfx(); works to call DerivedDerived vfx if Bp points to a DerivedDerived object, does "default" if not.
      - E.g. Ship has virtual void submerge() {} function.

- Pros: simple, easy, uses fast and general virtual function mechanism throughout. Compiler and linker guarantee correct behavior of call.
- Cons: clutters base with conceptually odd declarations - vfx might not be meaningful for the concept represented by Base. Isn't attempting to do vfx on the wrong kind of object a fundamental error?
  - Seems strange to have the code saying that it is meaningful to tell all Ships to submerge!
  - But the default action could be to signal that we were trying to call vfx on the wrong kind of object - what's wrong with that?
- Recommendation: A good solution if the fat interface is not very big and it is easy to maintain.

2. Use separate containers to keep track of objects by their types.

- If the structure of the program permits it, keep separate track of the pointers to Derived objects in such a way that they are known to be valid.
- A list of Base * for all of the objects. A separate list of pointers to Derived objects. Arrange code so that if vfx needs to be done, only the pointers to Derived objects are considered.
- E.g. Ship example. Use list of Ship * pointers to operate on all ships. Have a separate list of pointers for Submarines. If want to tell a Ship to submerge, look for its pointer in the Submarine list, and then use it to call submerge() function.
  - The Submarine list can contain either Submarine * or Ship * with a static cast to Submarine done for the call.
- Pro - clean design, uncluttered Base interface, no potentially slow or difficult to maintain RTTI usage. Can naturally flow from the design if kept in mind initially.
- Con - tends to weaken value of Polymorphic Hierarchy - have to keep separate containers of pointers, client code contaminated by the class hierarchy structure. Can be messy, difficult, or impractical.
- Recommendation: A good solution if the number of separate containers is small and unlikely to expand. Example: Ships versus Islands - things with lots of commands versus things with few or none, and need to distinguish in the command language anyhow.

3. Downcast safely.

- Determine whether it is valid to downcast the Base * pointer to Derived *, and if so, do the cast and the call.
- Use RTTI, especially dynamic_cast for this - its what it is for - avoid do-it-yourself type identification.
  - ```
    if(Derived * Dp = dynamic_cast< Derived *>(Bp) )
        Dp->vfx();
    else
        // do something else
    ```
  - ```
    /* here if command is "submerge" */
    if(Submarine * subp = dynamic_cast<Submarine * >(Ship_ptr) )
        subp->submerge();
    else
        throw Error("This ship cannot submerge!");
    ```
  - Using static_cast for this purpose is dangerous - can cause undefined behavior if pointed-to object is not really a Derived object.

- but if you use dynamic_cast, you have to test the result to get any benefit!
- If the dynamic_cast fails, it signals that we were trying to do vfx on an improper object - we can either do nothing, or treat it as an error of some sort.

▼ Pros:

- Keeps base class interface uncluttered, avoid conceptual difficulties

▼ Cons

- If done more than a little, tends to lead to switch-on-type logic, with its poor performance and difficulties for extension and maintenance. Some performance overhead, since dynamic_cast is a run-time operation that is generally slower than a virtual function call because it is actually very general and powerful (e.g. it knows how to navigate a multiple-inheritance class structure).

- Recommendation: Satisfactory if a failed cast means an error, as opposed to "try the next cast" - in other words, branch-on-type is still a bad idea even if done with dynamic_cast.

▼ How much dynamic_cast is OK?

▼ If a failed dynamic_cast is an error condition of some sort, then using it is not a problem.

- We're supposed to be able to do this, but let's check to be sure, and its an error if we can't.

▼ If a failed dynamic_cast means to try a different dynamic_cast, and continue until the code finds the "right" cast, then you have switch on type logic.

- Are you an X? No? Well, are you a Y? No, OK, how about a Z? - yuch!

- Sometimes switch-on-type can't be avoided, or actually represents the best approach, but this is rare - don't start with it, start with virtual functions instead!

▼ **4. Use "Capability" base classes and "capability queries."**

- An elaboration of the downcast safely approach.
- Instead of asking "Are you an object from class X?" we ask "Can we talk to you with the X interface?" - Maybe more than one class supports the X interface!

▼ Define a class that represents the special interface - a pure interface class, typically

- - e.g. a Submersible class that declares pure virtual functions for submerge, surface, etc.

```
class Submersible { virtual void submerge() = 0; }
class Transportable {virtual void load() = 0;  virtual void unload() = 0;}
```

- The Submarine class (and potentially other classes - e.g. James Bond's car) inherit from this Submersible interface class as well as the Ship class.

```
class Submarine : public Ship, public Submersible { ... };
```

▼ To access a function in the special class, do a dynamic_cast to the interface class pointer type, and test for success.

- E.g. attempt to dynamic_cast the Ship * pointer to a Submersible * pointer; if successful, then the object in question has the Submersible capabilities.

```
if (Submersible * p = dynamic_cast<Submersible *>(ship_ptr))
        p->submerge();
else
        throw Error("You can't talk to this ship that way!");
```

- Difference from "downcast safely" is that we are not asking "what specific leaf class type are you?" but "can I talk to you using this interface?"
- More general, less likely to turn into switch-on-type, but still has that potential problem.
  - Key is whether there are fewer interfaces than types of object.
    - If there is, then you can cover many kinds of objects with few capability classes
    - If not, the technique degenerates into switch-on-type maintenance problems.
- Recommendation: An good solution if there are a relatively small number of different capabilities and a large number of potential classes that are simply different combinations of capabilities. An elegant design can result. Otherwise, simply multiplies the complexity of the class structure.

- **Best solution depends on details of the situation.**
  - All designs are tradeoffs. What is being traded off here?
    - The extent to which the polymorphic class hierarchy is really a polymorphic class hierarchy versus a collection of unrelated classes.
    - It is unusual to have a bunch of classes that can be treated *exactly* alike - usually they can be treated alike in some ways, but there is something different about them in other ways.
    - We are looking for the "sweet spot" where we take the maximum advantage of how they are alike, and pay the lowest price for how they are different.
    - Each of these solutions will get ugly if in fact the classes have little in common and are large in number.
  - Key to the choice: How much future extension is likely and what will it be?
    - Only a few additional interface functions will be needed for new classes?
    - Only a few basic sub-types have to be treated differently?
    - A single class must be treated differently, but it is a rare special case?
    - More combinations of a set of capabilities?
    - Is talking to an object inappropriately always a user error?
  - Can use more than one approach!
  - Separate containers is either very clean, or extremely awkward. If there are only a few Derived functions that aren't declared virtual in Base, downcasting safely or using a capability class is often good. If the concept is not seriously damaged by the fat interface, it is by far the simplest, clearest, and the best performing.
    - Make the decision by considering how much future extension is going to be needed and what effects it will have.

- Will use of dynamic_cast or capabilities queries risk slipping down the slope into switch-on-typery? Will fat interface get grossly obese?

Additional Techniques for OOP

Decouple client from object creation details with a virtual constructor or factory.

What if client code is responsible for creating all of the Derived objects?

- To create an object of a class, it is necessary to know its class declaration.
- Now client has to #include Derived.h for each leaf Derived class.
- Result: if another Derived class is added to the hierarchy, client code must be modified, a new header possibly included.
- Another problem: If there is a change in a Derived class declaration (e.g. a new member variable), the client code has to be recompiled.
- Symptom of client and classes being coupled - interferes with easy development - change propagates through the system, forces other changes.
- Better to have client code decoupled from the classes.

*Concept: Insulate the client code from the creation process with a creation function that returns a Base * in response to some form of specification.*

- "Virtual constructor", "Factory" idiom/pattern - more later.

Define a function that returns a Base *, and accepts arguments that specify in some way or another the kind of object desired.

- e.g. maybe the specifications are in a file
- e.g. maybe a particular pattern of argument values determine the kind of object desired.
- e.g. the specification could be a string identifying the desired object class and initial values.

The function creates the desired kind of object with new and returns the pointer to it.

- in Factory.h - minimal declarations and includes:

```
class Base;

Base * factory(int i);
```

- Note how well-decoupled this is - include the factory creator function with ***nothing else*** coming along!
- in Factory.cpp - include class declarations

```
#include "Factory.h"
#include "Derived1.h"
#include "Derived2.h"
#include "Derived3.h"

Base * factory(int i)
{
        switch(i) {
```

```
                    case 1: return new Derived1;
                    case 2: return new Derived2;
                    case 3: return new Derived3;
                    // etc
                    }
}
```

- Define this function in the .cpp file, in a different module from the client code. Client #includes Base and creator prototype only.
  - #include "Base.h"
  - #include "Factory.h"
- If a new class is added, or a class is changed, only Factory.cpp will need modification or recompilation; client will need modification only to the extent that it has to supply a new specification.
  - e.g. if Factory uses a pattern of variable values, or a file, to get the specifications, client code may not need modification at all!
- One place the Factory function could be declared is as a static member of the Base class.
  - Keeps the Factory associated with the class hierarchy, but couples the .cpp file to rest of hierarchy.
  - But can be elsewhere - e.g. all by itself, helping decouple the base class code from the rest of the class hierarchy code.

- **Consider making the base class an interface class.**
  - *Pure interface class defines only pure virtual functions, has no member variables - strictly an interface to derived types.*
    - Very lightweight; often just needs a .h file, no .cpp
    - No commitments to any implementation details of derived types.
  - *But often useful to have some member variables or functions in the base class for common behavior - while not pure, it still serves a clear role.*
- **Let Derived class supply additional specialized work to the work done by the Base class.**
  - *Technique: a virtual function can have a definition at each level of the hierarchy; the derived version can call its base version before or after doing its own work.*
    - Example: Each class has a print() member function that outputs information about that classes member variables. Each class is responsible for outputting its own information.
      - Derived::print() calls Base::print() to output the Base member variables, then Derived::print() adds the information about its own member variables.
      - Repeat at each level of the hierarchy.
      - Contrast to the most Derived::print() accessing the data members of each Base class and printing it out. Code is highly repetitious, difficult to maintain. E.g. suppose we add another variable to the Base class ....
- **A non-virtual member function can call a virtual member function.**
  - *Use to provide a class-specific piece of functionality in the context of an ordinary member function.*

- Example: non-virtual of(), virtual vf():
  - Base::of()
  - {
    - vf(); // does virtual call if Class1 has Derived classes
  - }
  - Why this works: if a member function calls a member function, the call is through the this pointer:
    - this->vf();
  - if this object's class is a base class, the this has type base *, so a virtual call will be made.

- **Use dynamic_cast appropriately.**
  - *Flakey switch-on-type thinking - avoid if at all possible: What kind of object is it? Try dynamic_casts until one works - yuch!*
    - // use item_ptr to decide what to do with an item in a Dungeons & Dragons game:
      if (dynamic_cast<Apple *> (item_ptr)) {...}
      else if(dynamic_cast<Golden_potion *> (item_ptr)){...}
      else if(dynamic_cast<Iron_armor *> (item_ptr)) {...}
      else if ....
  - *OK: We are supposed to be able to tell the object to do this, but an error might have been made, so check the object type to be sure, and signal an error if it is the wrong type.*
    - perfectly OK to use in situations where if it fails it is due to a programming error, bug, or user input error
    - Dest_type * p = dynamic_cast<Dest_type *>(theBaseptr);
      if(!p)
      throw Error("Illegal command - try again!");
- **Be on the lookout for applications of design patterns**
  - *Many common design problems have been solved before.*
  - *Don't reinvent the wheel - apply previous patterns instead where appropriate and helpful.*
  - *Usually, multiple patterns, each with some customization, are used in a complex program.*