- **Idioms & Design Patterns Creational**
- **Introduction to Design Patterns**
  - Patterns and idioms can be grouped roughly into:
    - Creational Patterns and idioms
      - Singleton, Factory Method, Abstract Factory, Named Constructor
    - Structural patterns and idioms
      - Composite, Facade, Adapter, Compiler Firewall
    - Behavioral patterns and idioms
      - Observer, MVC, Double-dispatch
    - Idioms are small-scale patterns, in this context
    - Design patterns come in two basic flavors:
      - Using one or more base classes to hide details from the client
        - One thing about most of the design patterns: presented in a very general form
          - E.g. normally everything has an abstract base class that defines the working interfaces between the parts of the pattern
          - But an actuall implementation might not need this - the abstract base can be collapsed into a single concrete class
      - Other clever ideas using encapsulation, interfaces, class responsibilities to hide details from the client.
        - e.g. Singleton
  - Key concepts for using design patterns:
    - Take the class relationships and the details seriously!
      - You aren't taking advantage of the design pattern just by having some classes with the "buzzword" names organized kinda like the pattern. The exact way in which the classes relate to each other, and how key details are handled, is where the real power of the pattern is!
    - The goal of many of the patterns is to achieve easy extensibility, at the expense of some verbosity and some run-time overhead.
      - This means that a special case solution to a design problem will take less code, and maybe run faster, but it will be much harder to generalize when new features and capabilities get added to the code.
      - The need to extend a program is very common, even if it wasn't planned for.

- Either use a design pattern from the beginning, to allow for future extensibility, or be ready to refactor the special case solution to make use of a design pattern so that future extensions will go more smoothly.

▼ *So the goal of the patterns is not short code, or fast code, but easy-to-extend code.*

- *Don't wreck the pattern to make the code shorter or more efficient - you're missing the point!*

- **Singleton: Unique, globally-accessible Objects**
  - Sometimes you need exactly one object of a class and it should be globally accessible, but only in certain ways, relevant everywhere in the system, but there should be only one of them, and its creation, initialization, and destruction must be clear and well defined.
  - Common use: This object provides resources - e.g. data that is needed widely, gets updated, has to be kept consistent - so easiest if in exactly one place.
    - Datum * get_datum_ptr(const std::string& name);
    - can we put this into an object somewhere so that it will be available everywhere, and easily kept accurate?
  - What's wrong with global variables?
    - Access is open - anybody can change it in any way desired.
    - When is its initial value determined?
    - What if initial value requires run-time computation?
    - Common temptation to use global variables:
      - there is a single repository or set of services in the program.
      - If there really is only one, can't we treat this stuff as a global variable?
      - But what's to stop code from accidently modifying something directly?
      - Also, how do we make sure it is always only one variable for each kind of data - no inadvertent copies to get out of synch?
      - Need the encapsulation, access control of classes, but global access
  - Singleton pattern solves these problems
    - Define a class that has the data members and services that you want
      - map<string, Datum *> // a map container of datum names to pointers
void add_Datum_ptr(Datum * ptr);
bool is_datum_present(const std::string& name);
Datum * get_datum_ptr(const std::string& name);
void remove_datum_ptr(const std::string& name);
void remove_datum_ptr(Datum * ptr);
    - Ensure that only one object of that class can be created.
    - Ensure that this one object gets created when it needs to be created.
    - Provide a way to globally access the object's services, but control access as required.
  - Basic pattern:
    - file Singleton.h

- class Singleton {
  public:
  static Singleton * get_Instance();
  // services member functions
  // prevent copy/move construction and assignment
  Singleton(const Singleton&) = delete;
  Singleton& operator= (const Singleton&) = delete;
  Singleton(const Singleton&&) = delete;
  Singleton& operator= (const Singleton&&) = delete;
  private:
  // constructor is private nobody else can create one
  Singleton();
  static Singleton * ptr; // only one, class-wide

  // services data members initialized by ctor

  // prevent accidental deletion
  ~Singleton();
  };

- file Singleton.cpp
  Singleton * Singleton::ptr = 0; // initialized during or immediately after loading
  // create the object if it doesn't already exist
  Singleton * Singleton::get_Instance()
  {
  if (!ptr)
  ptr = new Singleton();
  return ptr;
  }
  // code for ctor, dtor (if needed), service functions

- Usage:
  class Data // the singleton class
  Datum * find_datum(const string& name);
  void add_datum(Datum *);
  void remove_datum(Datum *)
  etc
  // in other modules
  #include "Data.h"
  ...
  cin >> name;
  Datum * p = Data::get_Instance()->find_datum(name);

▼ Refinements:

▼ have get_Instance return a reference instead of a pointer

- syntactically more convenient, less error prone
- Singleton& Singleton::getInstance()
  {
  if (!ptr)
  ptr = new Singleton();
  return *ptr;
  }

  Datum * p = Data::getInstance().find_datum(name);

▼ How does it get destroyed?

- Singletons usually endure for the length of the program run; most OSs will recover memory and most other resources when a program terminates, so there is an argument for not worrying about this.
- But in this course, and more generally, we will want to follow the rule of cleaning up at program termination. How to do this with a Singleton?

▼ Auxiliary Destructor Class

▼ Can have an auxiliary static object SingletonDestroyer, from a friend class whose destructor reaches in and does a delete ptr; of the singleton object - for example:

- class Singleton {
  ...
  friend class Singleton_destroyer;
  };
  class Singleton_destroyer {
  public:
  ~Singleton_destroyer()
  {
  delete Singleton::ptr; // friend status, so access ptr directly
  }
  };
  // somewhere in code (Singleton.cpp is probably the best place)
  // create a global static Singleton_destroyer object
  Singleton_destoyer the_destroyer;

- When program starts, the destroyer object will be created (does nothing).
- When program terminates, all global/static objects are destroyed by the runtime library shutdown code (inserted by the linker), so the_destroyer will be destroyed; its destructor will delete the Singleton, running its destructor.

▼ The Meyers singleton: Neat solution contributed by C++ Wizard Scott Meyers: make the actual object a static local variable in the getInstance function.

▼ just define the get_Instance function differently:

```
static Singleton& Singleton::get_Instance ()
{
    static Singleton s;
    return s;
}
```

- Compiler automatically builds code that creates 's' first time through the declaration, not thereafter, and then deletes the static object at program termination

- But can get into very difficult situation if objects rely on each other at the time of termination - when does the Singleton disappear relative to other objects? Can read further for various ideas. But for simple applications, this works fine.

▼ Idiomatic use of the Singleton: Always call get_instance() function to access the Singleton - do not attempt to save a reference or pointer to it for more convenient use later.

- Why? A key feature is that it is guaranteed to be there if you need it provided you call the get_instance() function. If you rely on a stored pointer/reference to it, now if your code changes and you need it in a different place or time, the pointer or reference may not be available or valid.

- Note that get_instance is simple and can be defined in the class declaration and get inlined, so no significant function call overhead. Can also give it a shorter name, though get_instance() is the customary name for the function.

▼ Why not simply use global/static variables to solve the problem, or just have static members for every member variable in the singleton?

▼ Answer:

- If a bunch of naked static variables, no control over how they're accessed - just global variables.

- The static members will have their ctor's called during program startup, and before main() gets executed, but no guarantee on when or what order ACROSS translation units. Can get into strange and difficult situations if initializing a static member variable somewhere else requires the object, but its static member hasn't been initialized yet.

- But built-in types like pointers that are static and have constant initial values are always initialized before any initializations that involve function calls! Might even happen during loading (constant initial value might actually be part of program text in memory image loaded from disk).

- Therefore, Singleton using a static pointer intialized to zero or the Meyers singleton will always be well defined no matter when other code makes use of it! - it gets created the first time it is used, no matter what.

- **Named constructor idiom**
  - Control how an object is created
  - common case - only on the heap, so e.g. refer to with a pointer or shared_ptr.
    - make the constructors private
    - use a static "create" function that takes constructor parameters, constructs and returns the object
      - class Thing {
        public:
        static shared_ptr <Thing> create(int i);
        ~Thing{};
        private:
        Thing(int i = 0) : i_(i) {}

        int i_;
        };

        shared_ptr <Thing> Thing::create(int i)
        {
        shared_ptr <Thing> p (new Thing(i));
        return p;
        }

- **Virtual constructor idiom**
  - Not really a virtual constructor - but has that sort of effect - construct an object based on the run-time type of an existing object
  - Given an object that has some derived type, create another object like it, and return a Base class pointer to it
  - Create an object of a type without having to know what exact type it is
    - class Base {
      public:
      virtual Base * create() = 0;

      class Derived : public Base {
      public:
      Base * create() override {return new Derived;}

      .....
      Base * p = some object address;
      Base * new_p = p->create(); // another object of the same type, whatever it is
  - Related: virtual clone function - return a copy of the object
    - class Base {
      public:
      virtual Base * clone() = 0;

      class Derived : public Base {
      public:
      Base * clone() override {return new Derived(*this);}

      .....
      Base * p = some object address;
      Base * new_p = p->clone(); // a copy of the object, whatever type it is

- **Common problem: create an object whose type and initial values are specified by run-time data**
  - Suppose we have a class hierarchy of polymorphic objects:
    - Ship
      - Warship
        - Battleship
        - Torpedo_boat
      - Merchant
        - Trawler
        - Freighter
    - We usually refer to all of the objects by base class pointers, Ship *
  - Normally, to create an object of a particular type, you have to write explicit code in which the type explicitly appears

```
list<Ship *> ships;
// put some ships into the list
ships.push_back(new Torpedo_boat(data));
ships.push_back(new Battleship(data));
ships.push_back(new Trawler(data));
```

  - but suppose information and data about which objects is coming in from outside ..
    - E.g. create a new ship from a user command:
      - Enter command: create PT_boat PT109
      - type and name of object not known until program is running.
    - E.g. create a bunch of ships from data in a file:
      - BB Indefatigable 23000 12 15 24 18
      - PT 109 5 4 0.5 4
      - MF City_of_Peoria 10000
      - ...
  - Two problems: who decides what kind of object to create, and how do we get the objects initialized with the proper values?
    - Reminder: Constructors can have other parameters beside values for member variables, and can get those values from some other place besides the supplied parameters.

- e.g., a file can provide a source of the parameter data
  - ifstream infile;
  - ...
  - ships.push_back(new Battleship(infile));
    - constructor parses the data in the file, uses it to initialize the object.
  - ships.push_back(new Merchant_ship(infile));
    - constructor parses the data in the file, uses it to initialize the object.

- So, once we know what kind of object to build, we can delegate to each class the responsibility of handling the details, instead of in a single "know-it-all" place in the code.

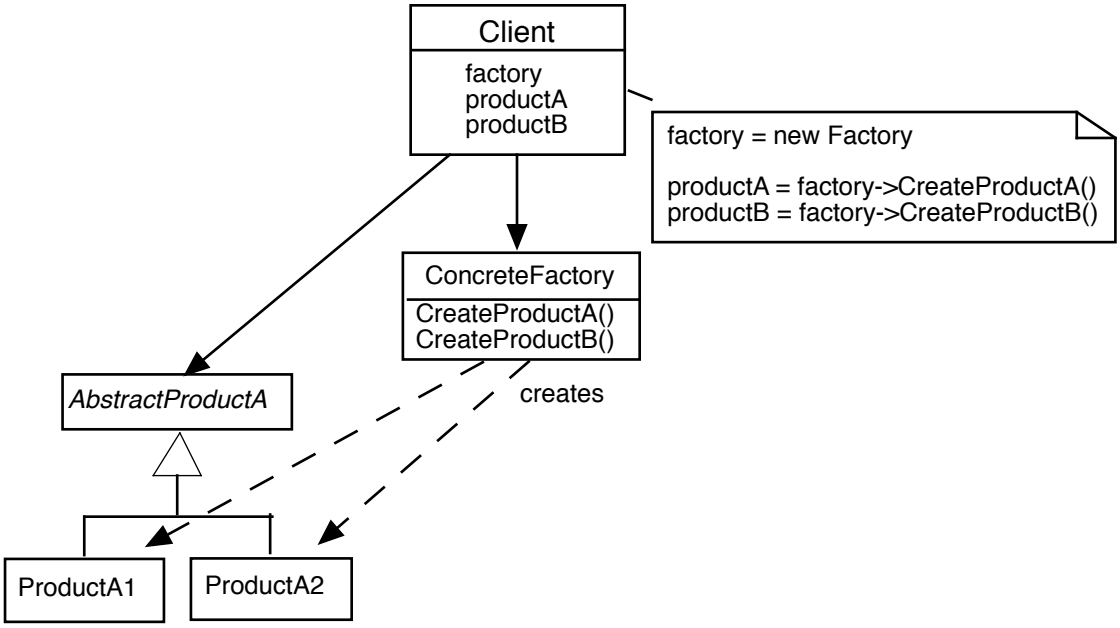
- **Factories: create objects for a client who doesn't need to know their actual types**
  - Simplest idea: A parameterized factory function - given information, create and return the desire kind of object
    - simple factory function - stand-alone function like in Project 4
  - Elementary factory *method* concept: A static member of the relevant class interprets the data and decides what kind of object to create, and makes the initialization data available to it.
    - class Ship: {
      static Ship * build_ship(int type_code, const string& name, int num_guns);
    - Or can give the functions different names to allow client to specify the type needed, and specify the parameters better:
      - class Ship: {
        static Ship * build_PT_boat(const string& name);
        static Ship * build_submarine(const string& name);
        static Ship * build_battleship(const string& name, int gun_size, int num_guns, int armor_thickness);
        static Ship * build_cargoship(const string& name, int capacity, int max_speed);
    - Can combine a static factory method with constructors that initialize from a data source, so that each subtype of Ship knows how to read its own initialization data
      - Ship * Ship::build_ship(ifstream& infile)

```
{
    string ship_kind;
    infile >> ship_kind;
    if (ship_kind == "BB")
        return new Battleship(infile);
    else if (ship_kind == "PT")
        return new Torpedo_boat(infile);
    ...
    else
        throw Error("Unknown kind of ship");
}
```

    - Client can then construct ships from a file:
      - while(infile)
        ships.push_back(Ship:: build_ship(infile));
    - DISADVANTAGE of factory method: Ship.cpp has to #include alll the headers for the derived types, meaning lots of coupling between the base and derived
  - Instead of returning a pointer to the new object, the factory could simply supply the pointer directly to a "centralized data bank" that keeps it for rest of program to use.

- A possible use of a Singleton - the centralized keeper of all the object pointers.

Concrete Factory/ Factory Method

- More sophisticated, a function for each type of object (possibly with parameters also), grouped into a concrete factory class - so-called Factory Method pattern, Concrete Factory version
- Concrete Factory
- 

Abstract Factory

- Another problem - what if at run time we have different families of objects to create?
  - e.g. use either this GUI library or that GUI library - client decides which to use, creates the corresponding factory object.
  - uses the factory to create the objects it needs
  - generally, the products from different families will share interfaces
- Abstract Factory
-