Object-*based* programming, instantiation, cxrs, static members, access rights, ref params, etc.

Supplementary notes

CS246 Fall 2012
Mike Godfrey

Note on notes on C++

- These are pretty much all my own examples and opinions, tho it is possible that some may have been "borrowed" from somewhere and the attribution forgotten

- I've tried to follow the topics in the Savitch book, but I've done a little reorganizing when it made sense to me to do so

-- MWG, Fall 2012

Procedural programming

Procedures plus variables (incl. `struct` instances)

i.e., what you did in C in CS136.

It's *not* the same as functional programming (e.g., Scheme/Racket)

- Variables are created in main program and passed as params to procedures (or are global)

- Each `struct` defines a *type* of variable, with variable sub-parts
  - Then create and manipulate instances of them
  - Each instance has own state

Object-based programming

Classes (fields + methods) + instances

- Classes/structs have variable sub-parts *and* procs that act on their sub-parts
  - Procs are called "methods"
  - Subparts are called "fields" or "member variables"
- Each *object* is an instance of a predef'd class/struct
- An object can have a subpart that is also an object (or a ptr to an object)

Object-oriented programming

Classes/instances + inheritance / polymorphism + generics

- Classes can extend other classes (partial blueprints)
- Some (abstract) classes have no instances, exist only to define common shapes of descendants
- Can treat instances of related classes in a similar manner (polymorphism)
- Generics: Can parameterized some classes (e.g., containers and their elements) by a type

  e.g., `List<T>` where `T` might be `string` or `int` or `Figure*`

Caveat

- There is a lot more detail and subtlety to OOP and C++ than we will present here
- I will try to tell you no lies, but I may not tell you the whole truth.
  - You can't handle the truth … yet ☺

Some OO terminology

- A class *member* is a variable or method defined inside a class
- A *member variable* (aka field) is a variable defined inside a class. There are two kinds:
  - Instance variable (one created per object instance, at instantiation)
  - `static` / class variable (only one created ever, period, at beginning of program execution)
- A *member function* (aka method) is a function defined inside a class, usually for operating on its member variables. There are two kinds:
  - Instance methods (can reference instance and static vars)
  - `static` / class methods (can reference `static` but NOT instance vars)

Some OO terminology

- An *object* (aka instance) of a class is an independent data element created with fresh, independent copies of each instance variable peculiar to the instance
  - The construction recipe is specified by special methods called *constructors* (cxrs)
  - In C++, we also have to kill off objects when they are no longer needed; the "cleanup" recipe is specified by a special method called a *destructor* (dxr)
    - Always declare dxrs as `virtual`; we'll explain why later
    - Often, dxrs are trivial
    - Some languages (Java, C#, Python) don't use dxrs; the run-time system collects "dead" objects using *garbage collection*

Creating `class/struct` instances

Instances of classes/structs are called *objects*; there are two ways to create them in C++:

1. Direct instantiation

   `className cObj;`

   – Space is allocated on the run-time stack
     - But instances disappear at the end of their defining scope
     - This is a problem if we want to pass around linked structures
     - Use a period to select field/method:

     ```
     Balloon b("red");
     b.speak();
     ```

Creating `class/struct` instances

2. Dynamic instantiation

   `className* cPtr= new className;`

   - Space is allocated on the heap (tho ptr `cPtr` is on the stack) and persists until explicitly deleted by programmer
   - Must remember to `delete` instances when no longer needed
   - Use an "arrow" (minus-greaterThan) to select field/method

   ```
   Balloon* bPtr = new Balloon ("red");
   bPtr->speak();
   …
   delete bPtr
   ```

```
#include<string>
#include<iostream>
using namespace std;

struct Coord {
    int x, y;
};  // Need ";" at end of struct!

void print (Coord c) {
    cout << "x = " << c.x << " y = " << c.y << endl;
}
```

```
int main () {
    Coord a;
    a.x = 3;
    a.y = 5;
    Coord b = a; // copy
    print (a);
    print (b);
    a.x = 17;
    print (a);
    print (b);

    Coord* p1;
    p1 = new Coord;
    p1->x = 4;
    p1->y = 12;

    Coord* p2; // alias
    p2 = p1;   // ptr copy
    print (*p1);
    p1->x = 42;
    print (*p2);

    Coord* p3 = &a; // evil
    // p3 is an alias
    p3->y = 83;
    print (a);

    delete p1;  // ok
    delete p2;  // error
    delete p3;  // error
    return 0;
}
```

Classes vs. structs

- We are going to use `class`es for OO code and `struct`s for non-OO code in this class
  - In practice, they are almost the same thing in C++
  - We suggest you follow the practice of using `struct`s when all you want is structure, and using `class`es when you want methods, inheritance, or generics

- So let's begin with some balloons!

- Wouldn't it be nice
  - If we could initialize the colour when we declare an instance?
  - if we could somehow tie operations on Balloons to the structs tightly?
  - If we could restrict access to the internal parts sometimes, so that only "official" procedures could operate on them?

Nit: Using the default cxr

```
Balloon b;
Balloon *pb = new Balloon;
Balloon rb ("red");
Balloon *prb = new Balloon ("red");
// Below is old style; it's still legal but don't do this
Balloon *pb2 = new Balloon();
Balloon b2();
```

- In each case (except last), appropriate cxr is called automatically
  - #1, 2, and 5 call cxr of no args, have "transparent" colour
  - #3, 4 call other cxr, have "red" colour

- First five are legal, last one is an error
  - The compiler will think you're declaring a function called Balloon whose definition will be given elsewhere

Defining methods

- Need to provide implementations (definitions) of each method (incl. con/destructors) after declaration
  - `Balloon::Balloon()` means the `Balloon` method (constructor) of no args of the class `Balloon`
  - Similarly for `void Balloon::speak()`
  - "`speak()`" by itself is not a procedure, it's called `Balloon::speak()`

Constructors

- A *constructor* is a special kind of procedure / method that is used to create a new instance
  - It has the same name as the class.
  - It specifies what needs to be done to the sub-parts to create a new instance
  - For any given instance, it is called exactly once, at the beginning of its lifetime
    - It is called *implicitly*, when the object is instantiated
  - There is no declared return type (but you get a new instance of that class)

Constructors

- There may be several cxrs for a class, but differ in the parameters they take
  - This is called *overloading*
  - These are alternatives for creation, for flexibility
    - You have to pick one of them ☺
- Usually, but not always, we define at least the *default* cxr
  - i.e., the one with no arguments
  - If you don't provide a default cxr, the compiler does NOT automatically create one for you if you have defined any other cxrs (unlike Java)

Inside methods

- A method can be called by an instance of the class

```
Balloon b;  // will use cxr of no args
b.speak();  // not "speak(b);"
```

- Inside method body, can access subparts of object
  - Directly or via the special pointer to myself: `this`

```
cout << "I'm a " << colour << " balloon!\n";
cout << "I'm a " << this->colour << " balloon!\n";
```

- A method is invoked on an instance using its own fields in the method body
  - So `b1.speak()` and `b2.speak()` will have different outputs if objects `b1` and `b2` have different colours
  - [draw a picture]

What's `this`?

- Resolving var. names inside a method body:
  e.g., "What does 'colour' refer to?"
  - First look at params, then at member variables.
  - So for `Balloon` constructor with string, you need to say
    `this -> colour = colour;`
    (Or else use a parameter name that is different from the field name)

- `this` is a pointer that, inside a method definition, points to the "object under consideration"
  - Don't need to use this most of the time, only when name conflicts are possible or if you just want to be absolutely clear which bits are part of the object

Constructors and initializers

- Constructors have two parts:
  - Initializer list (not in Java)
  - Constructor body

- Strong advice: Do as much as you can using initializers
  - The constructor body is for the awkward bits that you can't do via an initializer for some reason
  - Any sub-part that is not set using an initializer will be constructed automatically using the default constructor for its type, even if you reset the value in the body
    - So in our previous definition of `Balloon::Balloon(string)`, the `colour` field is first created and then given the empty string value before being reset to the provided parameter value in the cxr body

Initializers

- Format is: `partName (value)`
  - No `this` needed for part name

- Initializers are used for two purposes:
  - Calling cxr of parent class when using *inheritance* (we'll discuss this later)
  - Initializing member variables

- So below is the preferred approach to defining `Balloon` cxrs:

```
Balloon::Balloon () : colour ("transparent") {}
Balloon::Balloon (string colour) : colour (colour) {}
```

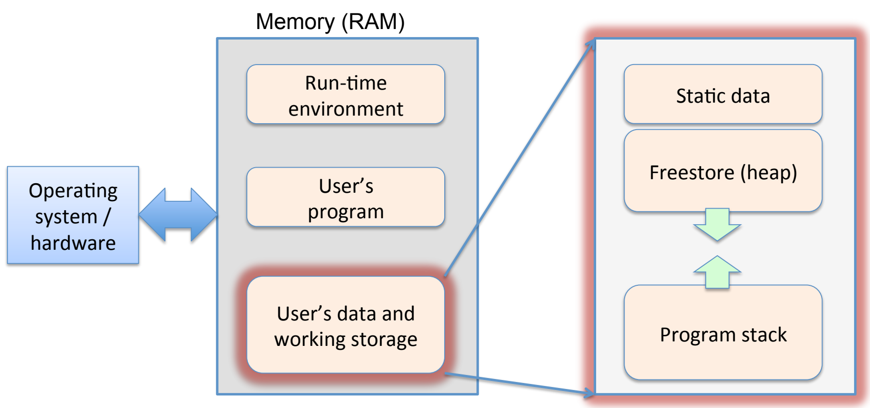
Basic CXR recipe

- When you create an object instance, all subparts need to be created too!

- For a given cxr, first we process the initializer list; if a given instance variable (i.e., sub-part) is not initialized there, then
  - If the part is an object, then we call the default (no arg) cxr for the part's class
  - If the part is a ptr, number, or other basic type, space is allocated for the variable but the initial value is random garbage

- Thus all parts have been constructed *after* processing initializer list but *before* processing cxr body
  - The cxr body can then proceed to do any further special initialization that was impossible to do using an initializer
  - You really do need to initialize those ptrs to `NULL`, `int`s to 0, etc

(Review) The C/C++ memory model



Garbage and destructors

- The amount of free storage in the heap (and stack space) you have for your running program is not unlimited
  – While you can increase the amount at run-time (depending on the underlying OS and language run-time system), it is expensive to do so, and ultimately also limited
- Can't do much about stack-based variables, as they represent a real, ongoing need for the current computation
  – Tho some tricks can help a bit, e.g., `const` ref params

Garbage and destructors

- What shall we do to heap-based variables that we no longer need?
  – The technical term for these variables is *garbage*
- Many newer languages (e.g., Java, C#, Python) use automatic *garbage collection* to reclaims "dead" variable storage
  – You don't need to do anything yourself!
  – The language run-time system periodically runs a little routine in the background that grabs all of the objects not currently being used, like your mother did for you when you were 4 years old
- C/C++ philosophy:
  – You made the mess, you clean it up! And keep track of your own messes too!
  – In C, use `malloc`, `free`, and other routines (it's a headache)
  – In C++, we define (one) destructor for each class; it gives a recipe for how to dispose of an object's subpart

Destructors

- Destructor is called *implicitly* whenever an object's scope is exited (object on stack) or `delete` is called (object on heap)
  – You don't ever call "`~Balloon`" directly
- Dxr says what happens to *heap-based* sub-parts when an object is destroyed.
  – Direct sub-objects are cleaned up automatically
  – Often the dxr is pretty trivial
  – Usually, declare dxr as `virtual`
  – Really good idea:
    - Put logging messages into cxr/dxr to help track what's going on (but take them out when done debugging)

`const` member variables

- Member variables can be constants too
  - In which case their value must be set with an initializer, and may never change later on.
    e.g., a child's name, a balloon's colour
  - Put some thought into which, if any, member variables should be constants, and which should be allowed to change over the object's lifetime

- `const` member variables need to be initialized using an initializer

`static` fields and methods

- A new *instance* variable/field (i.e., the usual kind in a class definition) is created for each object instance
  - And its value can change independently of the other instances over the lifespan of the object
  - An instance method (i.e., the usual kind), may look at or change the instance variables of that object (as well as the `static`/class vars!)

- A **`static`** variable/field (aka a *class variable*) is different:
  - There is only one of them ever, and it lives in a magical castle away from the grubby object instances
  - A `static` method (aka class method) cannot access instance variables (except via objects that get passed into it); generally, it is defined to manipulate static variables of the class

`static` fields and methods

- Scope-wise, `static` members "live" in the class they are defined in

- Sometimes, we use `static` members to track meta-information about the class usage
  e.g., how many instances created / currently active

- … but in reality, `static` vars/methods aren't used that often
  - They are sometimes useful, esp. for `enum` types and universal constants of the class like `DefaultColour`
    - There's only one universal value for the constant, so why store a copy of it with every object?

Copy constructors

- A *copy constructor* is a constructor that takes a `const` ref to existing object (of the same class) as its argument; it creates a copy of it as a new object. The declaration looks like this:

```
C::C (const C & c);
```

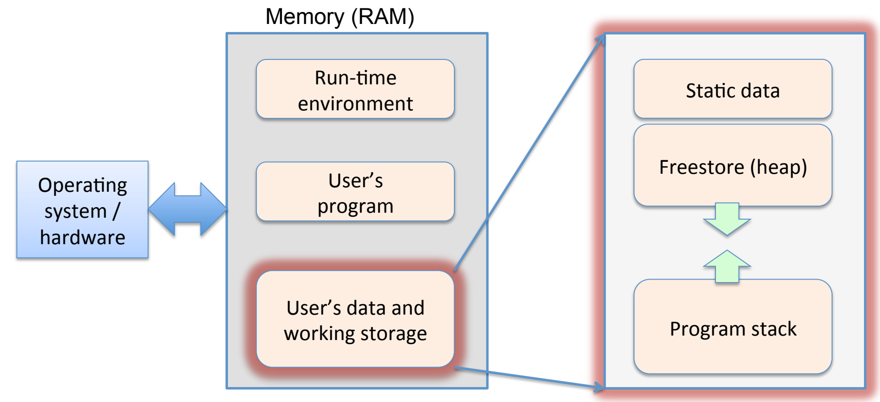
- There is an implicit default copy constructor predefined for every class; it performs a memberwise copy construction. similar to the default implementation of `operator=`
  - For each subpart d of type D, call the (implicit or user-defined) copy constructor **`D::D (const D & d)`**

Copy constructors

- Sometimes, memberwise copy construction is not the appropriate recipe; then you need to define your own customized copy cxr
  e.g., if you have ptrs to external objects that may be shared

[We will return to copy cxrs later]

(Review) The C/C++ memory model




```
class Child {
    public :
      Child ();
      Child (string name);
      Child (string name, string bColour);
      virtual ~Child();
      void speak();
      Balloon* pBalloon;  // Bad idea to be public!
       // Good idea to be ptr as balloons pop
    private :
      const string name;
};
Child::Child() : name ("Les Doe"), pBalloon(NULL) {}
Child::Child(string name):name(name),pBalloon(NULL){}
Child::Child(string name, string bColour)
    :name (name), pBalloon(new Balloon (bColour)) {}
```

Design questions

- Look at the three cxrs together … why is this a bad design?
  - Because two of three cxrs don't create a balloon but the third does.
  - This is inconsistent behaviour among overloaded methods; it would be better to create a balloon each time or never at cxr time.
- Also, does it make sense to create a `Child` without specifying a name? In this case, might be a good idea not to have a default cxr!
  - A good trick: Make (only) the default cxr `private`, then no clients can call it

```
// Inconsistent cxr behaviour: When do we get a Balloon?
class Child {
  public :
    Child ();
    Child (string name);
    Child (string name, string bColour);
    virtual ~Child();
    void speak();
    Balloon* pBalloon; // Bad idea for inst var to be public!
                       // Good idea to use ptr as balloons pop
  private :
     const string name;
};

// Unclear if first cxr makes sense to define
Child::Child() : name ("Les Doe"), pBalloon(NULL) {}
Child::Child(string name) : name(name), pBalloon(NULL) {}
Child::Child(string name, string bColour)
    :name (name), pBalloon(new Balloon (bColour)) {}
```

```
// Revised to have consistent cxr behaviour
class Child {
  public :
    Child ();
    Child (string name);
    Child (string name, string bColour);
    virtual ~Child();
    void speak();
    Balloon* pBalloon; // Bad idea for inst var to be public!
                       // Good idea to use ptr as balloons pop
  private :
     const string name;
};

// Unclear if first cxr makes sense to define
Child::Child() : name ("Les Doe"), pBalloon(new Balloon) {}
Child::Child(string name) : name(name), pBalloon(new Balloon){}
Child::Child(string name, string bColour)
    : name (name), pBalloon(new Balloon (bColour)) {}
```

Destructors and responsibilities

- When an object dies (via `delete` or having its scope end), all of its *direct* sub-objects will die also
  - So you don't need to worry about those in your dxr!
- But if the object has a *ptr* to an object, you need to know who is (now? later?) going to kill off that object
  - If you are the only one who knows about it, then just `delete` it in the body of your dxr
  - If the object is shared (others have ptrs to the same object), you need a global agreement of some kind about who will kill it later.
  - Rule of thumb: Whichever class creates a heap-based object should be responsible for `delete`-ing it later
- Advice:
  - Don't bother to declare / define a dxr for a class if it does nothing
  - If you do declare/define a dxr, it's usually a good idea to make it `virtual`

Don't make member vars. `public`

- Letting clients have arbitrary access (i.e., `public`) to member variables can cause all kinds of headaches
  - Wouldn't it be better to allow only controlled access?
  - The use of a getter/setter pair of methods solves problem #1, but not #2 which can only be solved by a clear design of responsibilities

Shared objects (aka "resources")

- If you create an object that is going to be "shared" by other objects, consider carefully where you want to create it
  - Usually, shared objects should be created on the heap via `new`
    - They won't be automatically deleted when the current scope ends!
  - Then you can pass a ptr to that object to others who want to share it
  - But you still need a clear understanding of how this shared object is eventually going to be deleted (who "owns" it)

- So seeing an object's address ("`&obj`") passed into a function that expects a pointer should make you uneasy …
  - Is the function going to keep a reference to that object? If so, disaster looms.

```
// Make data member private; add get/set.
class Child {
  public :
    Child (string name);
    Child (string name, string bColour);
    virtual ~Child();
    void speak();
    void acquireBalloon (Balloon* pBalloon);
    Balloon* giveAwayBalloon ();
  private :
    string name;
    Balloon* pBalloon;  // now private
};

Child::Child(string name)
    : name(name), pBalloon(new Balloon) {}
Child::Child(string name, string bColour)
    : name (name), pBalloon(new Balloon (bColour)) {}
```

More on the copy constructor

- Recall, copy cxr takes another object of the same type and returns a new one that is "just like" the existing one

- If you do not define a copy cxr yourself (and most of the time, you don't need to!), then the compiler will use its default implementation of "copy each sub-part", like this:

```
// These are the default impls (which you do not have to
// define explicitly if this approach works for your needs)

Balloon::Balloon (const Balloon & otherBalloon)
    : colour (otherBalloon.colour) {}

Child::Child (const Child & otherKid)
    : name(otherKid.name), pBalloon(otherKid.pBalloon) {}
```

More on the copy constructor

- The default copy cxr for `Balloon` is fine, but for `Child` it is problematic

```
Child dolly ("Dolly", "red");
Child bonnie (dolly);
```

- Again, if we don't define a custom copy cxr, then the default behaviour is to just copy the parts
  - In this case, we end up with two `Child`s but only one `Balloon` (which they both point to)
  - This is because the instance variable `pBalloon` is a ptr (not an object), and the default copy cxr just copies the ptr value
  - If we use the following copy cxr definition, then `bonnie` gets her own new red `Balloon`

More on the copy constructor

- Moral:
  - If you have sub-parts that are shared ("resources") or are ptrs to objects on the heap or are otherwise non-trivial, you probably need to define a custom copy cxr
  - [And you probably need a non-trivial dxr, and you probably need to provide a custom definition of `operator=`]

- So far, our cxrs for `Child` have always created a `Balloon` using either a provided colour, or the default colour for `Balloon`
- But when we added `giveAwayBalloon`, we added the possibility that a `Child` might not have a `Balloon` later on
  - So what if `pBalloon` is null?
  - So we probably need to check for this in the copy cxr

Will this work?

```
Child::Child (const Child & otherKid) : name(otherKid.name),
      pBalloon(new Balloon (*otherKid.pBalloon)) {}

int main (…) {
    // …
    Child dolly ("Dolly", "red");
    Balloon* myRedBalloon = dolly.giveAwayBalloon();
    delete myRedBalloon; // pop!
    Child bonnie (dolly);
    // …
```

```
// Revised to add custom copy cxr
class Child {
  public :
    Child (string name);
    Child (string name, string bColour);
    Child (const Child & otherKid);        // Copy cxr
    virtual ~Child();
    void speak();
    void acquireBalloon (Balloon* pBalloon);
    Balloon* giveAwayBalloon ();
  private :
    string name;
    Balloon* pBalloon;
};

Child::Child(const Child &otherKid) : name(otherKid.name) {
    if (otherKid.pBalloon == NULL) {
        pBalloon = new Balloon;
    } else {
        pBalloon = new Balloon(*otherKid.Balloon);
    }
}
```

More on the copy constructor

```
Balloon rb ("red");
Balloon rb1 (rb);
Balloon rb2 = rb;
```

- You might naturally think that
  - `rb1` uses the copy cxr, and
  - `rb2` is constructed using the no-arg cxr, then is overwritten using the assignment operator to have the same subparts as rb
- However, this isn't true. Both use the copy cxr!
  - "`T c = b;`" is a common enough pattern in programming that the compiler realizes that it's just creating a copy
  - So it uses the copy cxr and avoids the extra work of constructing the values of object `c` twice

"The Rule of Three"

- A strong hint that you need to define a customized copy cxr is that you have a non-trivial dxr (as `Child` does)
  - The reverse is true also!
- Once we have studied operator overloading, it will be obvious that the assignment operator (`operator=`) probably also needs a custom definition if either of the copy cxr or dxr are non-trivial
- This observation is summarized as "The rule of three" [Cline]
  - If you have a custom definition of *any of* the copy cxr, dxr, or `operator=`, then you need custom definitions for *all three* of them

Overloading

- As we saw with the `Balloon` cxrs, sometimes it is convenient to have multiple definitions of a method that differ in the parameters they take
  - This is called *method overloading*; it is particularly common in library classes, to maximize flexibility of possible use.

```
class Balloon {
  public :
     Balloon (string colour);
     ~Balloon ();
     void speak ();
     void speak (string extraMsg);
     void speak (ostream & os);
     void speak (ostream & os, string extraMsg);
  private :
     string colour;
};
```

```
// Design #1, verbose & repetitive
void Balloon::speak() {
    cout << "I'm a " << colour << " balloon!" << endl;
}

void Balloon::speak (string extraMsg) {
    cout << "I'm a " << colour << " balloon!" << endl;
    cout << extraMsg << endl;
}

void Balloon::speak(ostream & os) {
    os << "I'm a " << colour << " balloon!" << endl;
}

void speak (ostream & os, string extraMsg) {
    os << "I'm a " << colour << " balloon!" << endl;
    os << extraMsg << endl;
}
```

Overloading

- The compiler considers that there is one method here, `Balloon::speak`, with four possible definitions
  - The compiler looks at each call to `speak`, and figures out which definition to call based on the number and types of the arguments used by the caller.
  - You can't have two definitions of speak that take one string argument … the compiler can't distinguish between them.
- Can overload operators too (=,==,+,[],….) as we'll see later
  - Operator overloading Is the source of much pain in the world; Java does not support it for this reason

Access rights

- aka `public` / `protected` / `private` … (and friendship)
- `class`es (and `struct`s) can declare their member subparts to be one of the above
  - If you don't specify, `struct` subparts are `public` by default and `class` subparts are `private`
    - … this is the only difference between them. O/w C++ structs and classes are completely interchangeable tho I don't recommend you treat them this way (remember, kittens die if you do)
  - I do recommend that you always list the access rights explicitly, and don't just use the defaults.

Access rights (and *not* visibility)

`public`	Anyone may access these parts, incl. ext. clients
`protected`	Only my methods and those of my inheritance descendants may access these parts
`private`	Only my methods may access/use these parts

- `friend` classes and functions may also access public, protected, and private parts, but we won't talk about them yet
- Note that this is not the same as visibility!
  - Private parts are *visible* in children (and you can even redefine private methods in the children!), you just can't *use* them in the children

```
class Balloon {
  public :
    Balloon (string clr);
    virtual ~Balloon();
    void speak () const;
    int age;
  private :
    string colour;
};

// etc defs
```

```
int main (…) {
   Balloon b ("red");
   // legal, but fairly evil
   b.age = 12;
   cout << b.age << endl;
   // illegal
   cout << b.colour<<endl;
   b.colour = "green";
}
```

"Information hiding"
(aka modularity aka encapsulation)

- A basic principle of software design:
  - Expose the essential API, hide the implementation details
    - The public API should smell like a clean, abstract "thing"
  - Tell clients *what* they need to know to be able to use the class, but not *how* you are going to implement it
  - Don't let clients see your dirty laundry
    - They will come to depend on them. If you change your mind about implementation details, their client code will break.

Info hiding: Object-oriented design

- So the basic principle for class design is to make the *essential* operations `public`, and most everything else `private`, esp. fields
  - Within an inheritance hierarchy, sometimes we make parts `protected`, so that descendant classes can have access to them, but it's best to be as secretive as possible
  - If you need to allow clients some access to your subparts, declare get/set methods instead
    - But do this with care

Memory leaks

```
Balloon* gb = new Balloon ("green");
Balloon* rb = new Balloon ("red");
gb = rb; // Green balloon now a memory leak
```

- A *memory leak* occurs when an object (or other piece of storage, such as a dynamic array) on the heap is no longer accessible by anything on the (active) stack
  - The basic remedy is write your code so that you `delete` any objects once you no longer need them
  - This includes writing appropriate destructors for classes AND having a clear understanding of which class has responsibility for deleting objects that are shared

Memory leaks

- A memory leak is a chunk of storage that you can ever get to (legitimately), and so is a waste of your (limited) heap space
- Memory leaks are relatively benign *except that* you eventually run out of run-time storage for more heap-based objects + your program may die
  - A *systematic* memory leak means you are continually creating objects that become inaccessible
- Memory leaks are a big problem for industrial C/C++ programs
  - Languages that have a run-time (Java, C#, Python) that supports *garbage collection* are largely (but not entirely) immune to memory leaks

```
// Let's consider sharing my Balloon.  Design #1:
class Child {
    public:
        Child (string name, string bColour);
        // Probably want to define only one of the next two
        Balloon* shareBalloon();
        Balloon getBalloonCopy();
        void speak() const;
    private:
        string name;
        Balloon b;
};

Child::Child (string name, string bColour)
        : name(name), b(bColour) {}

Balloon* Child::shareBalloon () {
    return &b;     // This is a bit dangerous; what if I die but
}                  // someone still has a ptr to my Balloon?

Balloon Child::getBalloonCopy () {
    return b;      // This is OK, but you get a copy, not orig.
}                  // This calls copy cxr for Balloon
```

```
// If we are really sharing balloons, they should probably
// be created on the heap, like this.  Design #2:
class Child {
    public:
        Child (string name, string bColour);
        virtual ~Child(); // probably have to delete Balloon
        Balloon* shareBalloon();
        Balloon getBalloonCopy();
        void speak() const;
    private:
        string name;
        Balloon *bp;
};

Child::Child (string name, string bColour)
        : name(name), bp(new Balloon (bColour)) {}

Balloon* Child::shareBalloon () { //
    return bp;    // This is OK, as long as dxr knows other
}                 // objects may have a ptr to the balloon

Balloon Child::getBalloonCopy () {
    return *bp;   // Also OK, but you get a copy, not orig.
}                 // This calls copy cxr for Balloon
```

One last note on the copy cxr

- Common uses:

```
Balloon rb("red");     // normal instantiation
Balloon rbCopy(rb);    // calls copy cxr
Balloon rbCopy2 = rb;  // calls copy cxr (!!)
```

- However, the *most* common use of the copy cxr is implicit, by returning an object of that type:

```
// You get a *copy* of the Child's Balloon.
// Can also say "return Balloon(*pb);"
Balloon Child::getBalloonCopy () {
    return *pb;          // calls copy cxr!
}
```

Accessors and mutators

- Any instance method can be seen as either an *accessor* or a *mutator*
  - An *accessor* reports on the "value" of the object instance, but does not change it; a true accessor method can be declared as `const`
  - A *mutator* may change the "value" of the object
- There is a special category of (trivial) accessor/mutator pairs called getters and setters
  - Basically, you just retrieve / set the value of a member variable, but using a method to do so
  - Setter: You can instrument the setting, set a break point in a debugger, add helpful IO, etc
  - **Don't go crazy creating getters and setters for all of your member variables. You don't need most of them, and they defeat info hiding!**

```
class Child {
    public :
      Child (string name);
      virtual ~Child();
      void speak() const;
      string getName () const;
      void setName (string name);
    private :
      string name;
      Balloon* pBalloon;
};

string Child::getName() const {
    return name;
}

// Not clear we should permit the name to be reset …
void Child::setName(string name) {
    this->name = name;
}
```

The `const` modifier for methods

- If you have an true accessor method, you can "take the `const` pledge" by adding `const` to its signature
  - The compiler checks that you don't change any part of the object
    - It's OK to change params and local vars inside the method, tho
  - You're also not allowed to call non-`const` instance methods that might change the object
    - It's OK to call `static` non-`const` methods, tho. Why?
- This is an excellent habit to get into, as it makes you think hard about your design (who should be allowed to change this and when?)
  - It also serves as good documentation to others who read your code
  - I wish I remembered to do it more often in my own code

Taking the `const` pledge

- … means that you are promising not to change any of the subparts of the object
  - For subparts that are objects (i.e., not ptrs), the meaning is obvious: *You can't change the sub-parts!*
  - … but if the subpart is a pointer, you are promising only not to change the pointer to point to a different object
    - You *are* allowed to change the subparts of any object you point to, including calling non-`const` methods
    - This is maybe surprising and a little evil; however, it's best that you realize what the `const` pledge really means (and doesn't mean)

```
// Balloon-less Child class
class Child {
  public :
    Child (string name);
    virtual ~Child();
    void speak() const;
    string getName () const;
    void setName (string name);
    Child* getBff () const;
    void setBff (Child * bff);
  private :
    string name;
    Child* bff;
};

Child::Child(string name) : name(name), bff(NULL) {}
Child::~Child(){} // Don't delete bff!
```

```
void Child::speak() const {
    cout << "Child named " << name;
    if (NULL != bff) {
        cout << " with BFF " << bff->getName();
    }
    cout << endl;
}

string Child::getName() const {
    return name;
}

void Child::setName(string name) {
    this->name = name;
}
```

```
// "const" modifier means that we can't make bff point to
// another Child, but the below is totally legal (and evil).
Child* Child::getBff() const {
    bff->setName(bff->getName() + " stinks");  // Evil!
    return bff;
}

void Child::setBff (Child * bff) {
    this->bff = bff;
}

int main (int argc, char* argv[]) {
    Child trev ("Trevor");
    trev.speak();        // Child named Trevor
    Child ian ("Ian");
    ian.speak();         // Child named Ian
    trev.setBff (&ian);  // Child named Trevor with BFF Ian
    trev.speak();
    cout << trev.getBff()->getName() << endl; // Ian stinks
    ian.speak();        // Child named Ian stinks
}
```

Reference parameters

- C /Java support "call-by-value" (only) for parameters
  - Copy value of parameter onto call *stack frame* (aka *activation record*)
    - This has real overhead cost if the param is a "big" entity, like an object that contains many sub-objects
  - Only the return value (if any) is copied back
  - Can change param values inside proc, but changes do not propagate back to the calling environment
  - Can use ptr params to "cheat" (common C practice, we will avoid this)
    - Changes to ptr don't propagate back, but changes to values pointed to do!
    - Can use ptrs to ptrs to change ptrs!
- Wouldn't it be nice to allow changes to parameters to propagate back to the calling environment, if/when we want?

Reference parameters

- C++ also supports the idea of *reference* parameters
  - Put an ampersand after the parameter type in a proc decl

```
void swap (int& x, int& y) {…}
```

  - The param is not copied onto the call stack
  - Any changes you make in procedure will propagate back to the caller
    - This is called *call-by-reference*
  - The param acts kinda like a ptr, but you use it "normally" as a variable of that type, not by dereferencing with *
  - Ref params are the norm in C++; get used to them
  - Can have a ref or `const` ref *return type* …

Warning

- Using a (non-`const`) reference parameter is like giving away the keys to your car at a frat party
  - Just how much do you trust that method, anyway?
  - Sometimes, you do need to do it, but evaluate the risk

Advice:
  - If you might need to change the param inside the procedure (a mutator/setter operation), then use a ref param
  - If not (accessor/getter), do one of:
    1. Use a non-ref (value) param, as we have been doing
    2. Use a "`const` ref" param (the idiomatic C++ way)

```
string top1 (List& first) {
    string ans = first-> val;
    first = NULL;  // legal and nasty to caller
    return ans;
}

string top2 (List first) {
    string ans = first-> val;
    first = NULL; // legal but has no effect on caller
    return ans;
}

string top3 (const List& first) { // Use this style!
    string ans = first-> val;
    first = NULL; // illegal, compiler will complain
    return ans;
}
```

`const&` parameters

- A `const` ref param is used like a ref param, but the compiler will prevent you from changing the object inside the procedure
- Basically, it's almost the same as a value param
  - Tho it is legal (but mostly useless) to change a value param
  - `const` refs are also slightly more efficient, as you don't copy the whole object onto the run time stack, just a ref
  - This is the preferred way in C++ to declare params you don't intend to change

```
void GiveRaise1 (Employee e, int raise) {…}
void GiveRaise2 (const Employee e, const int raise) {…}
void GiveRaise3 (Employee &e, int &raise) {…}
void GiveRaise4 (const Employee &e, …) {…}
void GiveRaise5 (Employee *e, …) {…}
void GiveRaise6 (const Employee *e, …) {…}
void GiveRaise7 (Employee *const e, …) {…}
void GiveRaise8 (const Employee *const e, …) {…}

// …

int main (…) {
    Employee pFrank  = new Employee ("Frank", 5000);
    // salary is per month, int
    // …
    franksRaise = 700;
    GiveRaise1 (*pFrank, franksRaise);
    // …
}
```

`void GiveRaise1 (Employee e, int raise) {…}`

- When this fcn is called, a copy of the caller's object `e` and integer `raise` are created on stack.
- These copies may be changed within the fcn/method, but the values do not percolate back to caller and are lost at end of method call.
- The method for making a copy is defined by the `Employee` copy constructor

`void GiveRaise2 (const Employee e, const int raise){…}`

- A copy of object `e` and integer `raise` are created on stack; they may not be changed within the fcn/method.
- Effectively, this is pretty similar to GiveRaise1.

`void GiveRaise3 (Employee &e, int &raise) {…}`

- Any mention of object `e` or integer `raise` inside the method actually refers to the variables in the caller's environment.
- Thus, changes can be made to `raise` or `e` percolate back to the caller immediately.

`void GiveRaise4 (const Employee &e, …) {…}`

- Read this as "`e` is a ref to a thing that's a `const Employee`"
- Any mention of `e` refers to caller's environment, but you can't change the object `e` refers to in any way.
- Effectively, this is similar to `GiveRaise2` except that it may be more efficient if `Employee` is a "large" object.

`void GiveRaise5 (Employee *e, …) {…}`

- Call w/o dereferencing: `GiveRaise5(pFrank,…)` [same for 6, 7, 8]
- Read this as "`e` is a pointer to an `Employee`"
- A copy of a pointer is made on the stack; if you manage to change the member variables of the object `e` points to, these changes will persist in the calling environment
- You can also make `e` to point to a new or different object, but that change will be lost at the end of the method call, and not percolate back

`void GiveRaise6 (const Employee *e,…) {…}`

- Read this as "`e` is a pointer to a thing that's a `const Employee`"
- That is, `e` can be changed to point to a different `const Employee`, but any instance it points to cannot be changed internally.
- If you do change `e` to point to another object, this is not percolated back to the caller.

`void GiveRaise7 (Employee *const e, …) {…}`

- Read this as "`e` is a `const` pointer to an `Employee`"
- You can change the internal values of the object `e` points to, but you can't make `e` point to a different object.

`void GiveRaise8 (const Employee *const e, …) {…}`

- Read this as "`e` is a `const` pointer to a `const Employee`"
- Can't change the value of the object `e` points to; can't change `e` to point to another object

- Usually, use the style of
  - `GiveRaise3` if you *want* changes to percolate back to the calling environment, or
  - `GiveRaise4` if you *don't want* changes to percolate back
    - … but using value params as in `GiveRaise1` is also common

Object-*based* programming, instantiation, cxrs, static members, access rights, ref params, etc.

Supplementary notes

CS246 Fall 2012
Mike Godfrey