# Essential C++

This handout briefly reviews the principle C++ features and their syntax. It's actually patched together from three handouts, so excuse the rought transitions. The three parts are:

1) OOP Vocabulary and Concepts — applies to any OOP language

2) Essential C++ features

3) C++ examples

## 1 — OOP Vocabulary

Object Oriented Programming is paradigm which applies in a variety of languages. This handout summarizes the basic style, elements, and vocabulary of OOP which are common to all OOP languages. In a classical compiled language like Pascal or C, data-structures and their procedures tended to group logically, but it is the programmer's duty to devise and enforce these groupings. In OOP, the language groups procedures with their data type. This produces a decomposition grouped around the types in a program. The programmer does not have to match up the right procedure with the right data-type. Instead, variables know which operations they implement. OOP is proving itself to be a better way to engineer software. OOP yields better decomposition and more opportunities for code reuse. These strengths are especially important for writing large programs, packaging up code into libraries for use by others, and programming in teams.

### Class

The most basic concept in OOP is the Class. A Class is like a Type in classical language. Instead of just storing size and structural information like a Type, a Class also stores the operations relevant to itself. A Class is like an Abstract Data Type in Pascal or C— it bundles traditional data typing information with information on the procedures which will operate on that type.

### Object

An object is a run-time value which belongs to some class. So if Classes are like Types, then Objects are like variables. Run-time values are known as Objects or Instances of their Class. The Class collects and defines the properties that all of its instances have.

### Message

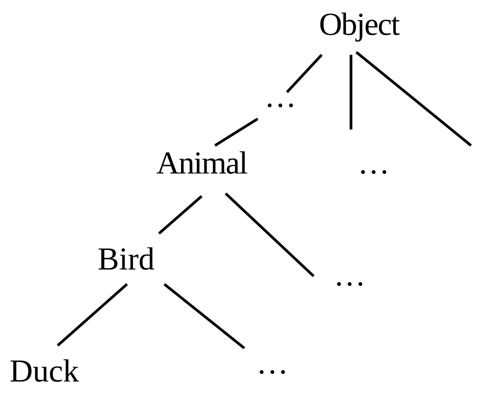
OOP uses Messages instead of procedure calls. Sending a message to an object causes that object to perform some action. It is the Object's responsibility to look at the message and think about what's appropriate. An object raises an error if it is ever messaged to do something it does not know how to do. Each Class defines code to be used to respond to messages. The code corresponding to a particular Message is known as the "Method" for that Message. A Message is just a string like "Pop". The Method for Pop is the code in the Stack class which is triggered by the "Pop" Message. There is only one "Pop" message, but many classes may have methods for Pop. The C++ specific term for method is "member function".

### Encapsulation

The internals of a class can be protected or "hidden" by the compiler. In C and Pascal, its just a convention that the client should not mess with or depend on the implementation of an ADT. With Encapsulation, the implementor can indicate that parts of the implementation are protected so that client code accessing these parts will not compile.

## Hierarchy

Classes in OOP are arranged in a tree-like hierarchy. A Class' Superclass is the class above it in the tree. The Classes below are Subclasses. The semantics of the hierarchy are that Classes have all the properties of their Superclasses. In this way the hierarchy is general towards the root and specific towards its leaves. The hierarchy helps add logic to a collection of classes. It also enables similar classes to share properties through Inheritance below. In C++, a "base class" is a synonym for superclass and "derived class" is a synonym for subclass.



## Inheritance

Inheritance is the process by which a Class inherits the properties of its Superclasses. Methods in particular are inherited. When an Object receives a Message, it checks for a corresponding Method. If one is found, it is executed. Otherwise the search for a matching Method travels up the tree to the Superclass of the Object's Class. This means that a Class automatically responds to all the Messages of its Superclasses. Most OOP languages include controls to limit which properties are inherited.

## Overriding

When an Object receives a Message, it checks its own Methods first before consulting its Superclass. This means that if the Object's Class and its Superclass both contain a Method for a Message, the Object's Method takes precedence. In other words, the first Method found in the hierarchy takes precedence. This is known as Overriding, because it gives a Class an easy way to intercept Messages before they get to its Superclass. Most OOP languages implement overriding based on the run-time class of objects. In C++, run-time overriding is an option invoked with the "virtual" keyword.

## Polymorphism

A big word for a simple concept. Often, many classes in a program will respond to some common message. In a graphics program, many of the classes are likely to implement the method "DrawSelf." In the program, such an object can safely be sent the DrawSelf message without knowing its exact class since all the classes implement DrawSelf. In other words, you can send the object a message without worrying about its class and be confident that it will just do the right thing. Polymorphism is important where the class of an object cannot be determined at compile-time.

# 2 — Core C++ Features

## Class = Storage + Behavior

In OOP in general, a "class" defines the memory size/layout its objects should have as well as the methods they should implement. In C++ the memory layout information looks like the members of a C struct, and the methods are at least reminiscent of C function headers. The instance variables and methods are collectively called the "members" of the class. The following creates an Account class with no superclass. The "virtual" keyword for each method enables run-tme method resolution which will come up later when we introduce subclasses.

```
class Account {
   private:
   float balance;          //"instance variables" or "data members"
   int transactions;

   public:
   virtual void Withdraw(float amt);   //"methods" or "member functions"
   virtual void Deposit(float amt);
   virtual float GetBalance(void);
};
```

## Access Control

In the class header, the specifiers `public:`, `private:, and protected:` control where the members can be accessed.

`public`: accessible anywhere— the "client" access level. Public members represent the abstract interface which the object presents to other objects which want to own, cooperate, or otherwise send messages to this object. Stylistically, some people feel that data members should never be public — that clients should always need to go through methods to interact with the state of an object.

`private`: accessible only within the methods of this class— the "implementor" access level. The access level for the implementation specific parts of the object. Clients will not be able to access any of the private parts of the class

`protected`: accessible in a method of this class or one of its subclasses. Like "private" but extended to include subclasses— see Subclassing below.

## Message Send

Sending a message in C++ is like accessing a member of a struct in C. The recipient object of the message is known as the "receiver" of the message. The class of the receiver will determine which *method* is invoked by the *message.* (how the message-to-method resolution happens and whether it happens at compile-time or run-time is a topic for later) The following code shows the basic syntax on the client side of an object. A client can declare an object and send it messages. Usually the data members in the object will be private so the client cannot manipulate the object internals directly.

```
{
   Account x, y;     // Declare Account objects
...<initialize x and y somehow-- see Constructors below>...

   x.Deposit(100);   // Send the "deposit" message to the receiver x
                     // which is of the Account class.
                     // Adds $100 to the x account.

   y.Deposit(200);   // Adds $200 to the y account.
```

By keeping data members `private:` and forcing clients to send messages to interact, the programmer can keep the abstraction and implementation separate. This is the same old Abstract Data Type procedural programming technique with all its attendant restrictions and advantages. The difference here is that C++ has official compiler support for keeping the internals of the object private— the programmer does not need to fiddle around with `void*` and casts to keep the implementation private.

## Methods

Method definitions look like function definitions but the accessible members of the receiver, in particular its instance variables, are available by name in the method body.

```
void Account::Withdraw(float amt) {
   balance -= amt;
   transactions++;
}
```

In this case the instance variables `balance` and `transactions` are automatically available in the method body. All of the instance variables and methods defined or inherited by the Account class are available. As a practical matter, having all the data of the receiver automatically available by name is a huge convenience. A special additional variable named "`this`" points to the receiver of the message. In the above case the `this` variable is type `(Account*)`. `This` is useful at times when an object needs to pass a pointer to itself or otherwise explicitly refer to itself. The references to `balance` and `transactions` in the method refer to the balance and transactions of the instance which received the message. So the code could be re-written equivalently in long form using `this->balance` and `this->transactions`. C++ programmers never use the long form.

An object can send a message to itself. Messages in the public interface of a class are often used by the implementor as well as the client. For example, the `deposit` method might want to send the `withdraw` message to itself to levy a $0.50 fee. As with the instance variables, message names are available without any extra syntax.

```
void Account::Deposit (float amt) {
   balance += amt;
   transactions++;
   Withdraw(0.50);   // equivalently in longhand: "this->Withdraw(0.50)"
}
```

## Constructors

Constructors are a convenient way to define the initial state of a newly allocated object. A constructor is syntactically similar to a method, but it is not invoked by the normal message-send mechanism. The constructor always has the same name as the name of the class and no return type, but it can have any type of parameter list. The following prototypes an Account constructor which takes an initial balance for the account.

```
class Account {
   public:
   Account(float initialAmount); // Initializes new accounts. No return type
...
```

## An Object is Born

Although syntactically constructors look like methods, they cannot be called or messaged explicitly as methods are. Instead constructors are automatically spliced in by the compiler whenever an object is initially allocated and needs to be given an initial state. The most common case where a constructor is invoked is when an object is declared locally like a variable in C.

```
{
   Account account(100);   // Allocates an Account object named "account" with
               // balance $100. The Account constructor is invoked by the
               // compiler automatically to initialize the new object.
```

or the following syntactic variant looks a little different but does exactly the same thing…

```
   Account account = Account(100);
```

Syntactically, the definition of a constructor looks like a method except for the absent return type. It doesn't need a return type because it is not going to be invoked like a method. Instead, it's going to "happen to" an uninitialized receiver to initialize it.

```
Account::Account(float initialAmount) {
   balance = initialAmount;
   transactions = 0;
}
```

All told, there are three cases where an object is newly allocated and so the compiler invokes its constructor.

1) (as above) An object is declared like a local variable and so is allocated on the stack.

2) An object is dynamically allocated in the heap with the new operator (below).

3) An object is a data-member inside of a containing object, and the containing object has itself just been allocated. In that case, the constructors are invoked first on any contained objects and later on the containing object. The object must be literally a data member inside the containing object. A pointer in the containing object to the contained object does not count.

To remember the order for the last case, remember the following truism "an object must be in as valid a state as possible when a method, constructor, or destructor (below) is invoked." Constructing the contained objects before their container puts the container in the best state possible (albeit incomplete) before its constructor is invoked.

## Overloading

C++, as well as many other languages, can deal with many methods sharing the same name. Through a facility known as "overloading," a language resolves which of several methods (or functions or procedures or whatever) with the same name should be called by considering the number and type of the actual arguments and the return type.

Overloading works for constructors as well as methods. For example, the following adds a second constructor which takes no arguments and gives the account an initial balance of 0 to the Account class . The overloading mechanism looks at the client code and deduces which is the appropriate constructor to use without any extra syntactic specification by the client.

```
class Account {
...
   public:
   Account(float initialAmount); // the original constructor
   Account(void);    // a second constructor which takes no arguments
...
};


Account::Account(void) {
   balance = 0;
   transactions = 0;
}



{
   Account x(100);   // Invokes the constructor which takes a float

   Account y;        // Invokes the zero argument constructor
                     // Also known as the "default constructor".

                     // not the same as "Account y();"
                     // which, of course, defines a function y which
                     // returns an Account.
                     // This is one of those times where it's pretty obvious
                     // the need to be backward compatible with C syntax has
                     // has compromised the quality of C++.

   Account y = Account();  // Alternate invocation of the default constructor
```

## Default Arguments

The header of a method constructor can specify constant default values for its formal parameters. The default values will substitute for omitted arguments when the message is sent. Syntactically, there is the restriction that actual arguments can only be omitted from right to left. In other words, if an argument is omitted, so must all the arguments to its right. Without this restriction, the compiler could not unambiguously figure out which arguments were being omitted. Default arguments are very often used with constructors. The following declaration of the Account constructor achieves the same effect as before using just one constructor and a default argument.

```
class Account {
...
   public:
   Account(float initialAmount = 0.0); // The argument defaults to 0.0
                                       // if the caller omits it.
...
};
```

If an actual parameter is given it will be used. Alternately, if there is no actual argument value for initialAmount, the default will be used.

```
{
   Account x(100);   //initialAmount = 100

   Account y;        //Rely on the default value so initialAmount = 0.0
...
}
```

## Destructors

Classes may define a destructor which is automatically invoked by the compiler whenever the memory for the object is reclaimed. The destructor can take care of any clean up which should happen whenever an instance goes away. The most common use is if an object owns a pointer to some dynamic memory, the object's destructor can free that memory. The destructor has the same name as the class but preceded by a tilde (~) and may not take arguments e.g. `~Account(void)`.

Destructors should almost always be virtual. Like Constructors, you do not explicitly invoke destructors in your code. Instead, they are "spliced in" automatically by the compiler just before an object is deleted. In particular, the compiler automatically knows to invoke the destructors all the way up the inheritance chain— starting with the deepest subclass and working up.

The three times the compiler realizes an object is going away are and so invokes its destructor are...

1) The object was a local variable in some block. At the end of the block, the variable will be deallocated.

2) The object is in the heap and `delete` is called on it (dynamic allocation is discussed below).

3) The object is a data member inside a containing object, and the containing object is about to be reclaimed. First the containing object's destructor is invoked, followed by the destructors for its contained objects. This order obeys the rule that an object should be in as complete a state as possible when code is invoked on it.

### Dynamic Objects
For many programs, all objects are allocated in the heap and referenced through pointers. The `new` operator allocates new dynamic memory. Unlike `malloc`, `new` gets the return type right. Like `malloc`, `new` will return `NULL` if it could not allocate the requested memory. The `delete` operator takes a pointer to an object and reclaims its memory (like free). As a convenience, it's ok to call delete on a NULL pointer— delete knows to just not do anything in that case.

```
Account *account;                // a pointer to an Account object.

account = new Account(100);      // allocate dynamically and invoke the
                                 // constructor
...
delete account;                  // invoke the destructor and reclaim the mem.
```

### Dynamic Arrays
The new operator can also be used to allocate arrays of objects. C++ uses the [] syntax to indicate the size of the array. The following syntax allocates and deallocates an array of characters just as you could with malloc() and free() in C.

```
char* buff;
int i = 13;

buff = new char[1000];     // Dynamically allocate array of 1000 chars

delete buff;

buff = new char[i];        // Same as above, but shows that size
                           // does not need to be a constant
```

The following code allocates and deallocates an array objects. This different from the simple C type case above because the class may have a constructor and destructor to invoke as the array is allocated and deallocated.

```
Account* accounts;
accounts = new Account[1000];    // Dynamically allocate 1000 Account objects.
                                 // The default constructor is invoked for
                                 // each of the 1000 objects.

delete [] accounts;  // The [] before the array reference reminds the compiler
                     // that this is an array of objects and so it should
                     // invoke the destructor for each object.
                     // Do not use the "delete [] xxx" syntax unless
                     // xxx really is an array of objects.
```

### Const
"Const" is a type qualifier much like "static" or "unsigned". The const qualifier applies to the type or variable which immediately follows it. Const specifies that the modified value may not be changed. In the following, the character variable ch may not be changed. Const variables must be given initial values. Const variables are preferable to #define for defining program constants— they keep their type explicitly and they are entered in the real program variable name space instead of being hacked in by the preprocessor.

```
const char ch = 'x'; // char character ch may not be changed

ch = ....;  // NO nothing of this form will compile
```

In the following pointer to a character, the const applies to the pointer. So the pointer s may not be changed. However the characters pointed to do not have any special protection.

```
char * const s = "hi there";     // the pointer s cannot be changed


s = ....;      // NO nothing of this form will compile
*s = 'x';      // YES, not changing the pointer s
s[6] = 'x'; // YES, the []'s are a form of pointer derefence
```

There can be multiple consts in a single declaration, so the following declares a character pointer where both the pointer and the character are protected.

```
const char * const s = "read only string";

s = ....;      // NO
*s = ...;      // NO
```

Finally, const can follow the paramter list of a method prototype in which case it means that the method will not change the receiver.

```
Account::GetBalance(void) const; // will not change the receiver account
```

The language uses the const information for variables and methods to add a layer of logical consistency to the data flow in the program. A const variable may not be changed. Its address may not be given to a pointer or reference if that pointer does not also carry the const restriction. And finally, a const obect may only be sent const messages. Depending on the compiler and computer architecture, const may enable some significant code optimization.

## Reference

Reference types set up aliases to objects. In C, programmers implement references using pointers. References play the same role in C++ abstractly, but the compiler takes care of the bookkeeping. They are useful to avoid making copies, and to propogate data-structure changes from one part of the program to another.

A reference acts as an alias to an object. Once the reference is set up, operations on the reference behave as if they were on the original referenced object. Syntactically, a reference type is set up by following the base type with an '&'. As a practical matter, references are almost always used as formal parameters, but they can be used for normal variables as well.

```
char ch;
char& charReference = ch;
```

The reference must be initialized when it is declared, and once set up, the reference cannot be changed to refer to another object. So conceptually, the refernce is like a temporary constant alias. However, operations on the reference variable "follow the reference" to the original referenced object.

```
charReference = 'x';       // The type of charReference is just char
               // but all actions "follow the reference,"
               // in this case to the char variable ch,
               // so ch is now 'x'.
```

• When you declare a variable, you are creating two distinct things:  the memory in which that variable's state is stored and a variable name which refers to that memory.

• When you declare a reference, you are only creating one thing: a new variable name which refers to some memory which has already been allocated (by a variable declaration).

For example, the following code:

```
   int i;
   int &ir = i;
```

creates one integer sized area of memory, and two variable names ("i" and "ir"). The names "i" and "ir" may be used interchangeably. Both variables are of type `int` (not `int*`).

## Reference Parameters

Parameters are by far the most common use of reference types. The called code wants to be able to refer some data in the caller. In C you get this effect by manually inserting &'s and *'s to explicitly pass pointers. In C++ you can just make the type of argument a reference paramter and everything will just work. The following simple example shows an integer being passed as a reference parameter.

```
   void Increment(int& value) {
      value++; // 'value' is of type int and can be treated as such.
               // However, because it is an & argument, the compiler ensures
               // that operations really go back to the actual caller argument.
   }

   void Caller() {
      int i = 6;

      Increment(i);     // No need to pass "&i" the compiler takes care of it

      // i is now 7
   }
```

## The Effective Type Is Not Changed by the &

The effective type of a reference variable in the source text is not affected by the reference. Something of type `char&` behaves exactly in the source as if it were of type `char`. This means that parameters and variables can be changed to reference and back to adjust the data flow in a program without disrupting the source code at all. This is a significant improvement over using & and * in C.

You should still use C style pointers when you might want to keep or change the references over time— so you still use pointers to build data-structures. References are used more as a tun-time information transfer mechanism either as parameters or return values. For example, the .next field in a linked list should still be implemented using pointers. It would be hard to implement a linked list with references given the restrictions that references be initialized when declared and never changed.

As with const, reference arguments may enable some significant code optimization. In particular, since the compiler knows that the reference will not change once defined, its a natural to implement references in registers and so avoid churning the stack. A quick rule of thumb is: const& arguments are the best possible form for the compiler optimizer. In that case the compiler has the latitude to choose between pass by value and pass by address, and pass in the stack vs. pass in registers.

## Subclassing

The following creates a subclass of Account.

```
class MonthlyFee : public Account {    // a "public" subclass of Account
   void EndMonth(void);
};
```

The `public` keyword in this context controls the access of inherited members. Essentially: `public` retains the access from  the superclass, `protected` forces the public members to become protected in the subclass, and `private` forces the public and protected members to become private in the subclass. This control mechanism can be used to keep subclasses from getting too much access to the internals of their superclasses.

"Subclassing" and "superclass" are the broadly accepted terms in the OOP world. The C++ subculture sometimes calls a superclass a "base" class and a subclass a "derived" class. "Derive" and "subclass" can both be used as verbs, but not in polite conversation. e.g. "So then thinking quickly, I just derived off of the scroll-bar base class and I was good to go."

## Class Substitutions

An instance of a class can stand in for an instance of its superclass. In the following example, `MommaBear` is a subclass of `Bear`.

```
Bear *bear;
MommaBear *mom;

...
bear = mom;    // ok-- MommaBear has all the properties of Bear so mom can
               // stand in for a Bear

mom = bear;    // not ok

...
void Foo(Bear *aBear);
void Bar(MommaBear *aMom);

...
Foo(mom);      // ok
Bar(bear);     // not ok
```

## Compile Time Error Checking

C++ uses compile time type information to error check as much as possible. So in the compile time pass of the source code, objects are assumed to have exactly their declared type. Using compile time types is a reasonable compromise and the best a compile-time oriented process can do. With that assumption, C++ can consistency check the uses and definitions of instance variables and methods. However, as demonstrated above, objects may not have their compile time type at run-time. For a well-behaved program, the best we can assume at run time is that an object will be an instance of the compile time class, or a subclass of the compile time class.

## Run-Time Message Resolution — Virtual Methods

What the statement `obj->Foo()` depends on the class of `obj` if more than one class responds to the `Foo()` message. Sometimes, the class of `obj` can be determined at compile-time. In that case, the compiler can generate code for the message send much like a traditional function call. More often, the exact class of an object will be determined at run-time. To deal with this case, OOP compilers will introduce extra bits kept at run-time for each object to indicate its class. This enables run-time "polymorphism" which is invaluable for some problems.

In C++, methods where the message to method resolution should be done according to the run-time class of the receiver are called "virtual". By default, C++ source code must specify that a method requires the virtual option in the class header as below. A method which is not virtual will *not* consider the run time class of an object. Instead the message to method resolution will happen at compile time based on (often inaccurate) compile time variable and type declarations. Often the compiler has an option where it just assumes that all methods are virtual (-fall-virtual in gcc). If your code depends on this option, your comments should say so.

If you forget the virtual keyword, sadly, your code will compile perfectly, and you will just get the following the classic symptom at run time: "I send the object the correct message, but it doesn't execute the method code in its class, instead it skips immediately to the method code in its superclass."

Compiling with virtual-by-default does not mean that all message to method resolution will happen at run time. It just means that it is the compiler's problem to figure out when the resolution can be done accurately at compile time, and then to do so as an optimization invisible to the programmer.

Early on, virtual methods were regarded as a fancy option because they added to the message send overhead. However, virtual methods make some things so much easier for the programmer that they are now regarded as standard equipment. Nonetheless, the older, more conservative view that non-virtual is the default persists in C++. If a superclass declares that a method is virtual, then overriding subclasses do not strictly need to redeclare that it is virtual, but it doesn't hurt.

In the following example there are three types of bank account subclassed from Account. Each class has its own definition of the `SendNastyLetter()` method.

```
class Account {
...
   virtual void SendNastyLetter(void);
};

class Monthly : public Account {
...
   virtual void SendNastyLetter(void);
};

class NickleNDime : public Account {
...
   virtual void SendNastyLetter(void);
};

class Gambler : public Account {
...
   virtual void SendNastyLetter(void);
};

...
{
   int acctNum, numAccounts;
   Account *accounts[];  // all the accounts
...
   for (acctNum=0; acctNum<numAccounts; acctNum++)
      accounts[acctNum]->SendNastyLetter();  // Polymorphism at run time
}
```

The accounts array contains all the accounts for the bank. All of the accounts are subclasses of the Account class. At compile-time there is no way to know which of the accounts are `(NickleNDime*)`, which are`(Gambler*)`, and which are `(Monthly*)`. The only way for the line "`accounts[acctNum]->SendNastyLetter()`" to do the right thing at run-time is for each object to know what class it is at run-time and so direct the `SendNastyLetter()` message to the method for its class. This is canonical "virtual" behavior and its indispensable for some problems. (The "banking" problem is worked completely the examples section below)

# 3 — C++ Examples

This handout has some basic but compelete C++ examples. They mainly demonstrate properties of inheritance and polymorphism. They are also good examples to have at hand to remind you of the basics of C++ syntax.

```
/***** bear.h *****/
#pragma once

/*
 BEAR
 Your Basic Bear. Its abstraction is that
 it has a weight property which can be get
 and set. The bear also responds to the Meanness()
 message which tells the client how mad the
 bear is currently.
*/
class Bear {
   public:
   Bear(float aWeight);                // constructor

   void SetWeight(float aWeight);             // weight accessors
   float GetWeight(void);

   virtual float Meanness(void);       // The one interesting polymorphic method
                                       // once we introduce the Mommabear subclass

   protected:
   float weight;                // We could make this private: and so make Mommabear
                                // use the accessors like other clients.
};

/***** bear.cp *****/

#include <stdio.h>

#include "bear.h"

/*** Constructor ***/
Bear::Bear(float aWeight) {
   weight = aWeight;
}

/*** Accessors ***/
void Bear::SetWeight(float aWeight) {
   weight = aWeight;
}

float Bear::GetWeight(void) {
   return(weight);
}

/*** Meanness ***/
float Bear::Meanness(void) {
   if (weight <= 100)
      return(weight * 2);
   else
      return(weight * 3);
}
```

```
/* mommabear.h */
#pragma once

// We need to see our superclasses header
#include "bear.h"

/*
 MOMMA-BEAR
 The momma bear extends the basic bear by possibly having
 a cub.  The cub instance variable will point to the cub
 or will be NULL if no cub is present.
 Mommabears also differ from bears by being 2x
 meaner.  Also Mommabear also respond to the
 TotalMeanness() message which adds in the meanness of the cub if
 if it is present.
*/
class Mommabear : public Bear {

    public:
    Mommabear(float aWeight);
    void SetCub(Bear *aCub);
    Bear *GetCub(void);
    virtual float Meanness(void);
    float TotalMeanness(void);

    protected:
    Bear *cub;

    public:
    static void TestBears(void);
};


/***** mommabear.cp *****/

#include <stdio.h>

#include "mommabear.h"

/*** Constructor ***/
Mommabear::Mommabear(float aWeight)
              : Bear(aWeight)                 // pass along to the Bear constructor
{
    cub = NULL;
}


/*** Acessors ***/

Bear* Mommabear::GetCub(void) {
    return(cub);
}

void Mommabear::SetCub(Bear *aCub) {
    cub = aCub;
}
```

```
/*** Meannness ***/

/* Use the inherited meanness and then multiply by 2. */
float Mommabear::Meanness(void) {
   return(Bear::Meanness() * 2);
}


/* The totalMeanness is just the mom's meanness + the cub's. */
float Mommabear::TotalMeanness(void) {
   float cubMeanness;

   if (cub != NULL)
      cubMeanness =  cub->Meanness();
   else
      cubMeanness = 0.0;

   return( Meanness() + cubMeanness);  // send ourselves the Meanness() msg
}


void Mommabear::TestBears(void) {
    Bear*  cub;
    Mommabear*    mom;

    cub = new Bear(50);

    mom = new Mommabear(300);

    printf("%g\n",cub->Meanness());  /* prints 50*2 = 100 */
    printf("%g\n",mom->Meanness());  /* prints 300*3*2 = 1800 */
    mom->SetCub(cub);
    cub->SetWeight(75);
    printf("%g\n",mom->TotalMeanness()); /* prints 300*3*2 + 75*2 = 1950 */
}
```

## Banking Problem

Consider an object-oriented design for the following problem. You need to store information for bank accounts. For purposes of the problem, assume that you only need to store the current balance, and the total number of transactions for each account. The goal for the problem is to avoid duplicating code between the three types of account. An account needs to respond to the following messages:

```
Constructor or Init message
Initialize a new account

void Deposit(float amt)
add the amt to the balance and increment the number of transactions

void Withdraw(float amt)
subtract the amt to the balance and increment the number of transactions

float GetBalance();
return the current balance

void EndMonth()
the account will be sent this message once a month, so it should levy any
monthly fees at that time and print out the account monthly summary
```

There are three types of account:

*Normal*: deposit and withdraw just affect the balance. There is a $5.00 monthly fee which counts as a normal withdrawal transaction.

*Nickle 'n Dime*: Each withdrawal generates a $0.50 fee. Both the withdrawal and the fee count as a transaction.

*The Gambler*: A withdrawal returns the requested amount of money- however the amount deducted from the balance is as follows: there is a 0.49 probability that no money will actually be subtracted from the balance. There is a 0.51 probability that twice the amount actually withdrawn will be subtracted.

Propose some classes to store the idea of an account. You may use an abstract super class or you may subclass some of the account from others.

Where are the instance variables declared?

Give the definitions of the `withdraw:` methods

Where do you use overriding?

How difficult would it be to extend your scheme so each account also stored its minimum monthly balance? What ramifications does this have for using (inherited withdraw) versus just changing the balance instance variable directly?

```
/***** account.h**** */
/*
The Account class is an abstract super class with the default
characteristics of a bank account. It maintains a balance
and a current number of transactions. It does not have a
full implementation of EndMonth() behavior which should
be filled in by its subclasses.

*/

class Account  {
   public:
   Account(void);
    virtual void Withdraw (float amt);
   virtual void Deposit (float amt);
    virtual float GetBalance(void);
    virtual void EndMonth(void) = NULL;        // force subclasses to implement


   protected:
    float balance;
    int transactions;
   void EndMonthUtil(void);     // save some code repetition in the subclasses

   // Class Function
    public:
    static void TestOneMonth(void);
};

/***** account.cp *****/

#include <stdio.h>
#include <stdlib.h>

#include "account.h"



Account::Account(void) {
   balance = 0.0;
   transactions = 0;
}


void Account::Withdraw (float amt) {
   balance = balance - amt;
   transactions++;
}


void Account::Deposit (float amt) {
   balance = balance + amt;
   transactions++;
}


float Account::GetBalance(void) {
   return(balance);
}
```

```
/* Factors some common behavior up which will occur in
the subclasses EndMonth(). */
void Account::EndMonthUtil(void) {

  printf("transactions:%d balance:%f\n", transactions, balance);
  transactions = 0;
}


/*-------------Subclasses--------------------*/

/*The Monthly Fee type of account*/
class MonthlyFee :  public Account {
    void EndMonth(void);
};


/*The Nickle 'n' Dime type of account.*/
class NickleNDime :  public Account {
    void EndMonth(void);
};


/*The Gambler type of account*/
class Gambler :  public Account {
    void Withdraw (float amt);
    void EndMonth(void)  { EndMonthUtil(); }   // this is a syntax
};


void NickleNDime::EndMonth(void) {
   Withdraw(transactions * 0.50);
   Account::EndMonthUtil();
}

void MonthlyFee::EndMonth(void) {
   Withdraw(5.00);
   Account::EndMonthUtil();
}

static int RandomNum(int num) {
   return(rand() % num);
}

void Gambler::Withdraw (float amt) {
   if (RandomNum(100) <= 50) Account::Withdraw(2 * amt);
   else Account::Withdraw(0.00);
}


/*------------Support Functions----------------------*/


/* If C++ kept class name information around at run-time,
this would be easier. */
static Account *RandomAccount(void) {
   switch (RandomNum(3)) {
       case 0: return(new Gambler); break;
       case 1: return(new NickleNDime); break;
       case 2: return(new MonthlyFee); break;
   }
   return(0);
}
```

```
#define NUMACCOUNTS 20

/*
 This is a static class function, so you call it like regular function,
 except with Account:: before the function name.
*/
void Account::TestOneMonth(void) {
    Account* accounts[NUMACCOUNTS];
    int accountNum;
    int day;

    /* Create a bunch of accounts of various types */
    for (accountNum=0; accountNum<NUMACCOUNTS; accountNum++) {
        accounts[accountNum] = RandomAccount();
        accounts[accountNum]->Deposit(100);
    }

    /* do a months worth of random transactions */
    for (day=1; day<=31; day++) {
        accountNum = RandomNum(NUMACCOUNTS);           // select an account at random
            if (RandomNum(2))                   // deposit or withdraw
            accounts[accountNum]->Deposit(RandomNum(100));    // yeah! polymorphism!!
            else
            accounts[accountNum]->Withdraw(RandomNum(100));   // yeah! polymorphism!!
    }

    for (accountNum=0; accountNum<NUMACCOUNTS; accountNum++) {
        printf("account:%d ", accountNum);
        accounts[accountNum]->EndMonth();
    }

}
```

```
/* main.cp */

#include "account.h"
#include "bear.h"
#include "mommabear.h"

void main(void) {
   Account::TestOneMonth();     // Invoke the static class function

   Mommabear::TestBears();
}
```

```
output:

Account Number:   1 transactions:   4 balance:-35.50
Account Number:   2 transactions:   4 balance:13.00
Account Number:   3 transactions:   4 balance:281.00
Account Number:   4 transactions:   3 balance:-31.00
Account Number:   5 transactions:   1 balance:100.00
Account Number:   6 transactions:   2 balance:130.00
Account Number:   7 transactions:   1 balance:100.00
Account Number:   8 transactions:   3 balance:147.00
Account Number:   9 transactions:   2 balance:88.00
Account Number:  10 transactions:   5 balance:70.00
Account Number:  11 transactions:   5 balance:252.00
Account Number:  12 transactions:   5 balance:-63.00
Account Number:  13 transactions:   3 balance:153.00
Account Number:  14 transactions:   3 balance:249.00
Account Number:  15 transactions:   2 balance:146.00
Account Number:  16 transactions:   3 balance:183.00
Account Number:  17 transactions:   3 balance:108.00
Account Number:  18 transactions:   2 balance:99.50
Account Number:  19 transactions:   3 balance:83.00
Account Number:  20 transactions:   3 balance:187.00
```

## Instructor Inheritance Example

This is my favorite inheritance example of all time. The first parts demonstrate basic overriding. In the last part, if you can grasp how GetMail() is written to surgically factor out the shared behavior between the three classes, then you truly understand inheritance. This is actually an CS107 final exam problem— more evidence that you never want to take a class from me!

For this problem, you will design C++ classes suitable for storing information about university instructors. The goal of the example is to demonstrate arranging classes in a hierarchy for maximum code-sharing. There are three types of instructor: Faculty, Lecturer, and Grad student. At any time, an instructor can best be described by three quantities: number of unread e-mail messages, age, and number of eccentricities. An instructor should initially have no unread mail and no eccentricities.

There are two measures of an instructor's current mood: Stress and Respect.

Stress— an instructor's stress level is the number of unread messages. However, Stress is never more than 1000. Grad students are the exception. Their stress is double the number of unread messages and their maximum stress is 2000.

Respect- generally an instructor's level of respect in the community is their age minus the number of eccentricities. Respect can never be negative. Faculty are the exception— Faculty eccentricities are regarded as "signs of a troubled genius" thereby increasing respect. So for faculty respect is age *plus* number of eccentricities.

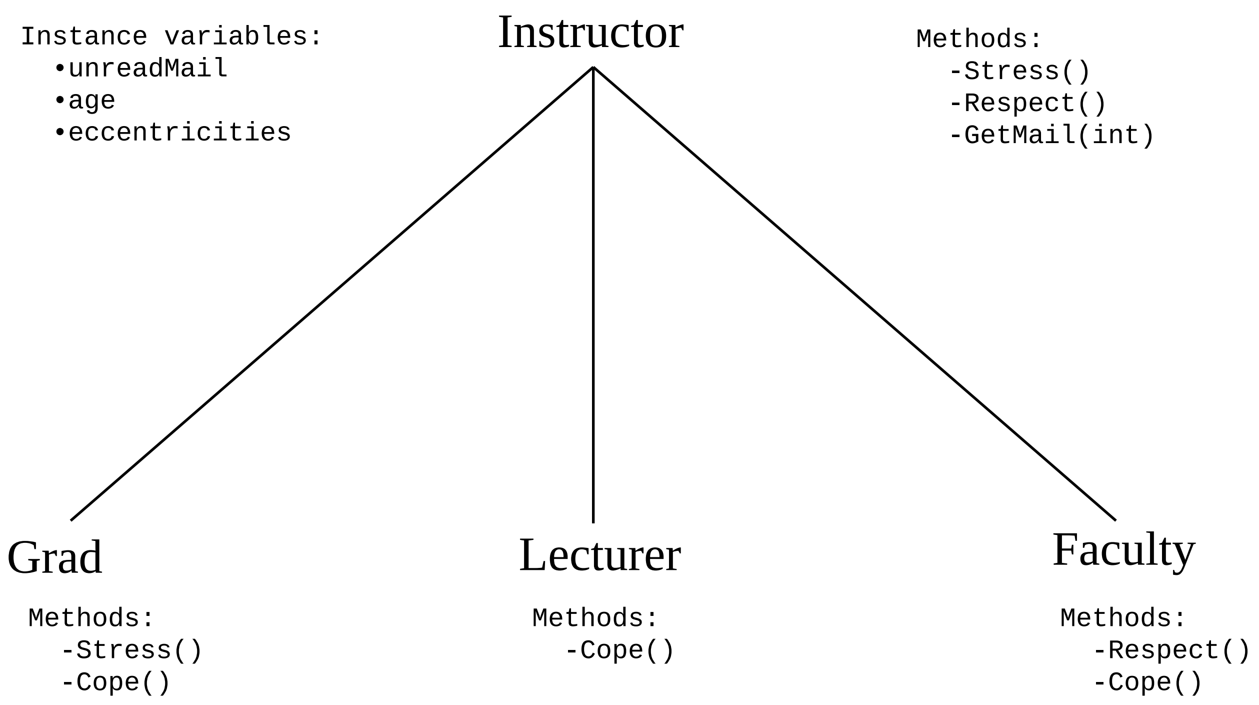
Everything in an instructor's life is driven by receiving e-mail. Several things happen when an instructor gets new, un-read e-mail:

First, the amount of unread-mail is increased.

Second, 90% of the time, there will be no change in the number of eccentricities. 10% of the time the number of eccentricities will randomly go up or down by one.

Third, the new unread mail may cause the instructor's Stress factor to become larger than their Respect factor which makes the instructor unhappy. In this case, instructors have different coping mechanisms. Faculty react by gaining 10 eccentricities. Lecturers react by accidentally deleting half their unread mail. Grad students react by reading all of their unread mail. The resulting mental anguish causes the Grad student's eccentricities to go up or down by one randomly. The coping mechanisms may or may not bring the Stress and Respect factors back in line. The coping mechanism is activated at most one time for each batch of incoming mail.

This is the sort of design drawing you might make to help think about your solution:



```
#include <stdlib.h>
const int kMaxStress = 1000;

class Instructor {
public:
   Instructor(int anAge);
   virtual int Stress() const;
   virtual int Respect() const;
   virtual void GetMail(int);

protected:
   int unreadMail;
   int age;
   int eccentricities;

   void RandomEccentric();// private helper to change eccentricities by +-1

   // "pure virtual" helper method
   // This indicates to the compiler that Instructor
   // is an abstract super class.
   virtual void Cope() = NULL;
};

// Construct with an age and init other to 0.
Instructor::Instructor(int anAge) {
   age = anAge;
   unreadMail = 0;
   eccentricities = 0;
}


int Instructor::Stress() const {
   if (unreadMail <= kMaxStress) return(unreadMail);
   else return(kMaxStress);
}


int Instructor::Respect() const {
   int stress = age - eccentricities;

   if (stress>=0) return(stress);
   else return(0);
}


void Instructor::GetMail(int numMessages) {
   unreadMail += numMessages;

   if ((rand() % 10) == 0) RandomEccentric(); // 10% chance of ecc. change

   if (Stress() > Respect()) Cope();// the key polymorphic line
}
```

```
// Helper which randomly change the eccentricities
// up or down by 1. Does not allow negative number
// of eccentricities.
void Instructor::RandomEccentric() {
   int delta;

   //if ((rand() % 2) == 0) delta = 1;
   //else delta = -1;

   eccentricities += delta;

   if (eccentricities<0) eccentricities = 0;
}

/**********GRAD STUDENT**********/
class GradStudent : public Instructor {
   GradStudent(int anAge):Instructor(anAge){};
   int Stress() const;
   void Cope();
};


int GradStudent::Stress() const {
   return(2 * Instructor::Stress());
}

void GradStudent::Cope() {
   unreadMail = 0;
   RandomEccentric();
}


/********** LECTURER **********/
class Lecturer : public Instructor {
   Lecturer(int anAge): Instructor(anAge){};
   void Cope();
};

void Lecturer::Cope() {
   unreadMail = unreadMail / 2;
}

/********** PROFESSOR **********/
class Professor : public Instructor {
   Professor(int anAge):Instructor(anAge){};
   int Respect() const;
   void Cope();
};


int Professor::Respect() const {
   return(age + eccentricities);
}


void Professor::Cope() {
   eccentricities += 10;
}
```