C++ Notes.

<< **Output or "insertion" operator** cout << "hi there" ; (also bit shift operator)
>> Input or "extraction" operator cin >> answer ;

. Member selection operator or "direct member selection" operator. (Can be overloaded)
-> indirect member access operator, or "pointer to member operator" (Can be overloaded)
:: Scope resolution operator
:: *name* means global. (The bare scope resolution operator means the variable is global.)
% Modulus operator, gives remainder.
static_cast <long>

Namespace is a way of grouping variables, example
Namespace jimbo { int x; char cat[55]};
Then can reference jimbo:: x to be explicit or use "using namespace jimbo" at top
using namespace std; means use the standard library name space.

inline means executable code inserted at every point
Functions with their body in the decleration are automatically inline.

Member function a function specific to the class objects.
method A method is a class "function." All methods are functions, but not all functions are methods.
Non member function stand alone function not part of any class, ordinary C function.

Member Functions
Public Can be called from anyplace
Private Can be called from within the class
Protected Can be called from derived classes.
Static a static member function exists independently of any object of a class
Virtual forces late binding, that is, use derived class function if it exists.

Base Class from which others can be derived these are thus "**derived**" classes.

Const functions The "**const**" declaration after a member function means that calling the function cannot change object values, only return results. "**volatile**" is the opposite of const.

Words to know **--->**
const tells the compiler any attempt to modify the value or contents is an error.
void says that a function or method will return no value.
this **this** is a pointer which refers to the current (calling) object.

New Operator creates an object in heap memory
Delete operator removes object and returns memory.

```
ptr  = new   this_class ( par1, par2,par3);
delete ptr;

char * my_string = new char[length];
delete [ ] my_string;
```

l value is something that refers to an address in memory.
A class is a data type.
Key words == reserved words.
Compound Statement == group of statements.
Signature == Parameter list.
`(*ptr).name` identical to **`ptr->name`**

INPUT AND OUTPUT

Member functions:

```
cin.ignore ( 80, '\n');                  discards up to 80 char and the end of record.
cin.ignore ( );                          "consumes" end of record from input stream.
cin.getline( buffer, count );            sucks in the whole line. up to \n
cin.getline( buffer, count, endchar );   sucks in up to count or  endchr
```

get() function takes either zero, one, two, or three arguments, and returns a reference to the object (the istream class) that invoked the get() function. This function leaves unused characters in the input stream. The get() function belongs to the istream class. You **can** include more than one get() function in a statement.

```
cin.get ( );                    "Consumes" a character
cin.get ( char );               Sucks in one character
cin.get (string, len);          Sucks up to "len" char or \n, does not consume
cin.get (string, len, endchar); similar, endchar defines end of input char
```

getline "consumes" the delimiter character, get does not !

```
cout.width ( count );       Sets width, only for next read.
cout.precision ( count);    Sets precision permanently. (six characters is the default precision.)
```

To specify the number of digits to the right of the decimal point:

```
cout.setf(ios::fixed);
cout.precision(2);          // Now precision is number decimal digits, not total digits.
```

`setf()` function takes format flags as arguments, to allow the programmer to change the output characteristics.
`unsetf()` reverses the bit settings for a particular format
Example:

```
ccc.setf(ios::showpos | ios::dec | ios::showpoint);
```

MANIPULATORS

endl equivlent to '\n' **ends** equivalent to '\0' **flush** flushes the buffer

setw (*count*) Sets field width, must be repeated for every use. requires **iomanip.h**
Example:

```
cout << setw ( 5 ) << dogcount << setw( 5 ) << catcount <<  endl   ;
```

"**setioflags**" is a manipulator, "**setf**" is a member function.

```
cout << setiosflags   ( ios::fixed | ios::showpoint  |ios::right );
cout << resetiosflags ( ios::left );
```

The setprecision() manipulator is used to specify the number of decimals that will print. It works like the precision() function. Setprecision() is considered a manipulator because you chain a call to setprecision(), along with other output and the insertion operator, rather than using an object and a dot operator, as you do with

parameters such as **fixed** are static constants of the ios class.

left	left justification	**hex**	hex ouutput
right	right justrification	**oct**	octal (base 8) output
dec	decimal	**showpos**	show + sign
showpoint	decimal point to appear	**fixed**	forces digits to right of point in precision

```
cout << setfill('*');     // sets the fill character to star instead of blank.
```

An example:

```
void Loan::showPayment ( int options, ostream & ccc )
{ ccc << setiosflags ( ios::fixed | ios::showpoint);
  ccc.precision (2);
     ccc << setw(4) << pmtCount<< setw(10)<<paidPrin;
  ccc << setw(10) << paidInt<< setw(12)<<  balance ;
  if ( options & 1) ccc << endl;
}
```

FILES

```
ifstream            Input file stream
ofstream            Output file stream
fstream
```

```
ifstream jimfile;                  // creates input file object
jimfile.open ( filename );         // opens file
ifstream jimfile ( filename );     // does both at once
```

jimfile true if success, as in

```
if ( jimfile ) cout << "file open successful";
```

`jimfile.good()` and `jimfile.fail()` also return boolian true or false.

```
jimfile.close();          // often not needed if object goes out of scope
```

Can also open with "switches" as in:

```
jimfile.open  ("file name", ios::noreplace | ios::binary );
```

That is, you can include an ios:: flag that specifies the access mode for the file, such as

ios::out	//open the file for output (default mode for ostream objects).
ios::app or append	// open the file for appending, rather than recreating the file.
ios::ate	// position the file pointer at the end of the file
ios::nocreate	// the file must exist; otherwise the open fails
ios::noreplace	// the file must not exist, otherwise the open fails
ios::binary	// force binary mode

Can read an entire file with something like:

```
while ( jimfile) {
      jimfile.getline ( iar, 100);
}
```

At the end of a file:

jimfile.**eof**()	True
jimfile.**fail**()	True
jimfile.**bad**()	True
jimfile.**good**()	False

An output file example:

```
ofstream jimout ( "puppy.txt", ios::nocreate ); / /opens the file for output only if it exists
```

There are general reads and writes:

```
jimfile.write ( pointer or address,  size );
```

To write an object:

```
jimfile.write ( (char *) object, sizeof( object )) ;  //read is the same deal
```

Includes:

conio.h	
fstream.h	file support
iostream.h	basic C++ io objects
iomanip.h	manipulators

Field – Smallest unit of information in a file.
File – Collection of records.
Format Flags – The arguments that determine the state of the cout object. Also called state flags.
Manipulator Function – Changes (manipulates) the state of the cout object.
Permanent Storage Device – device for storing information permanently, e.g., disk, tape, or CD
Record – Collection of data fields
Stream – A sequence of characters used to perform input and output operations.

STRING CLASS

treats strings as objects instead of null terminated arrays, never overruns space.
Allows stuff like string = str3 + " abc" + "\n"
You can freely substitute CString objects for const char* and LPCTSTR function arguments.
MBCS = Multibyte Character Sets

Xstr.c_str(); // returns a "c style" string.

You can use **strcpy** to convert back and forth between Cstring objects and plain old C char arrays:
(Does not work with my movstr function because strcpy is properly overloaded, I think…)

```
char xar[200];
CString mystring ("this is the string.");
strcpy ( xar, mystring);
bprint ("xar->%s",mystring);
```

strcpy(char *, const char *);	substr (start, count)	returns sub string
strcat(char *, const char *);	substr (start, count,string)	replaces string
strcmp(const char *, const char *);	Esample:	
strlen(const char *);	strx = string1.substr(6,10);	

endl equivlent to '\n'

Odd Stuff

switch (expression)
case: value
case: value
default:

```
switch  dog_age {
      case 3:   cout << 1; break;
      case 17:  cout << 17; break;
      default: cout "bad age";
}
```

Library routines to know:
pow (number, exponent) returns double #include < math.h >

Function initilization in the prototype

void xfunct_ref (char * str = "default string ");

char *strx is exactly the same as **char strx[]**

Void show_it (const char my_msg* = "Default message string");
Void show_it (const char my_msg[] = "Default message string");

Can be called show_it ( ) ; // for default string
or this way show_it {"Blessed are the poor in spirit") ; // to use specific string

Static Data Data members not associataed with any object.
Must be initialized outside of header.

```
class  doggie
{  static int count;
}
int doggie::count = 0;                    //You must have that "int" here !
```

According to the Ivor Horton Book,

-> is the "indirect member access" operator Can be overloaded
. is the "direct member selection" operator. Can be overloaded
.* is the de-reference pointer to class member operator. Can not be overloaded.

POINTERS TO CLASS MEMBERS.
The Schildt book (C++ the Complete Reference) has a section on pointers to class members. (There are pointers to methods as well as data elements.)

If ClasssX is a class, then the decleration is:

```
int ClassX::*data;   // data declared a member pointer
```

the utilization, where obx is an object of ClassX, is along these lines:

```
data = & ClasssX:: val  ;     // val a member variable of ClassX
cout << obx.*data << endl;
```

References

Reference Variable type: & symbol denotes reference, & is reference operator.

int xnum = 0;

int & xref = xnum; Now xref is a reference and can be passed as a reference to xnum

You must assign a value to a reference when you create it, a reference is once and forever!

The neat thing about a reference is that the declaration of a reference in the function call means that the function can operate on the original variable, without any change in the calling procedure or declarations in the calling program:

```
#include <iostream.h>
void xrout ( int  xx)
{      xx = 777;            // no change when we get back !
}
void ref_rout  ( int & xx) // note the reference operator &
{      xx = 777;
}

/*****************************************************/
void  main( void )
{      int doggie = 234;
       xrout ( doggie);
       cout << "doggie passed as int= " << doggie << endl; // shows  234
       ref_rout (doggie);
       cout << "doggie passed as ref= " << doggie << endl; // shows 777
}
```

References can be to objects too:

CBox cigar ;

CBox & rcigar = cigar ; Creates reference rcigar to object cigar.

When a call to a function is a refernce to an object, such as **function (CBox abox)**

then references in the function are of the form **abox.volume, not abox ->volume()**

Functions returning a reference

A function declared as returning a reference can be an "l value" and thus appear on the left hand side of an assignment.

int & lowest (int xx, int len);

lowest (yy, len_yy) = 17; // A valid assignment statement!

Passing Objects to Functions.

A function call with an object as a parameter passes a complete, bit by bit, copy of the object. This is often a bad thing to do, as an object can be a huge thing!

```
{   my_class objx (10);     //  Creates object objx
    my_function (objx);     //  Call to function
}
```

Case 1: void my_function (my_class loc_obj) { } Copy of object passed, destructor called.

Case 2: void my_function (my_class & loc_obj) { } Reference to object passed.

In case 1 the original object is not effected, and destructor called when function returns.
In case 2 actions on the referenced object are actions on the original object.

The nature of the copy process can be changed by definition of a special copy constructor.
(Important case: when the creation of an object allocates memory.)

Constructors and Destructors

Always member functions, almost always public.
A class may have several (overloaded) constructors, but only one destructor.
Destructor functions never have arguments.
Constructor function name same as class name, destructor preceded with tiled "~"
They do not return a type, even void.
Never called explicitly, run automatically when object created and then destroyed .(Going out of scope destroys.)

Constructor Initialization List

```
class doggie {
        doggie  ( int parm1 = 12, int parm2 = 14) : loc_val1 ( parm1) ,  loc_val2(parm2)  {}
```

Means that the constructor doggie will set local (class) value loc_val1 to the passed value of parm1
And loc_val2 to the passed value of parm2.
If the object creation does not pass values, then 12 and 14 are used.
Thus doggie adoggie(); sets the first local parameter to 12 and the other one to 14.

Base class constructors from the derived class:
When a derived class object is created, normally the detault base class constructor is executed, then the derived class constructor.

We can force an alternate constructor to execute by using function overloading:

```
Class Bouvier  : public doggie  {
     Bouvier ( int  p1, p2   )  : doggie ( p1, p2  ) ;
…..
}
```

When a Bouvier object is created , the override constructor in doggie is called, passing on parameters p1 and p2.

Base class constructors called before derived.
Derived class destructors called first.

Constructors

Must have same name as class.
Are optional.
Can be overloaded, that is, there can be several.
Have no return type.
default constructor: Any constructor with no arguments, even one you make yourself.

```
can be called     class  newobject;
but not           class newobject ( );    // compiler would see function call to "newobject"
```

Destructors

Name must be Tilde (~) followed by class name
Never have any arguments, thus there is only one destructor function.
Have no return type.
Called whenever the object goes out of scope, which can sneak up on you, as when you exit a function where object was created.

CBox boxes[5]; Creates an array of 5 objects of class CBox.

const CBox box1 ("sample string", 5.0, 6.0); declares object box1 of class CBox which cannot be changed

If the constructor has only one parameter, you can use this form:
CBox box1 = "sample string";

Friend functions have access to private and protected class elements.

Forward Reference. A " class name decleration only."

Friend functions are allowed to access private or protected members of a class even though it is not a member of the class.

Could be a member function of another class.

A function may be a friend of a class or an entire class may be a member of the the class.

A class declares whi it's friends are, a function cannot declare itself the friend of a class..

The friend decleration may take place in either the public or private sections, it makes no difference.

Classes can be friends, which means there is access to all data members. This is not two way, if both classes are to have access then there must be two friend declarations. (Class frindship is not inherited.)
Format of decleration

```
friend class SomeClass;      //    "class" is optional.
```

Friend – A nonmember function or class that is given access to private members of a class.
Forward Reference – A class name declaration only, with details to follow. Necessary when two classes refer to each other.

Operator Overloading

Context dependent meaning for operators such as << , + ,=,* and >>.
operator is a keyword, always used as operator *symbol* () such as operator+() or operator/(buick &xx)

Operator overloading must preserve precedance, associativity and number of operands.

Can not be overloaded **.** (dot operator) ***** Member Selector
:: Scope resolution operator **?:** Conditional operator
Size of

```
class CBox   {                                  // An overload for the + operator for class Cbox
public:
     int height;
     int operator + ( const & CBox  abox)
     {                                          //  Argument must be a constant reference
          return   height + abox.height;        //   to an object of this class
     }
}
```

Subscript operator is [ ], one argument.
Parenthesis operator ( ), many arguments.

Overloaded Assignment (=) Operator

C++ provides a default overloaded = operator for every class, you can write your own.

Example:

```
class CMessage {
        CMessage & operator = ( const Cmessage & aMess ){
                The code….
        }
}
```

Notice that both the return and the argument are references to objects of this class.

Copy Constructor

A copy constructor is a class function whose name is the same as the class name and whose argument is a reference to a class object. (Copy constructor **must** have a reference as a parameter!)
They can be implicit or explicit
Explicit copy constructors are of the form **CBox (const CBox& xx)** where xx is an object of class CBox

```
Example         CBox   box1 ( 10.0, 11.0, 12.0);
                CBox   box2  =  box1;        Calls the copy constructor to make box2 same as box1;
          or    CBox   box2 ( box1);         Does the same thing
```

Example of a copy constructor function:

```
        CBox ( const CBox& xbox);       where xbox is reference to object of class CBox
```

Note the difference:

Copy Constructor Called when a new object is created
Assignment Operator Called when one object is set equal to another object of the same class with an assignment statement.

Prefix ++ uses no argument.class
Postfix ++ uses a single dummy integer araguement.

```
    void operator++( )      //    Prefix operator overload
    {   rate += (double).01;
    }

    void operator++( int )  //    Postfix operator overload
    {   rate += (double).02;
    }
```

Overloading your own input and Output operators.

<<	is output operator or insertion operator	cout is a member of ostream
>>	is input operator or extraction operator	cin is a member of istream.

Prototype format:

```
friend ostream & operator<< (ostream & xout, MyClass & myClassObject);

friend ostream& operator <<( ostream & xout, Sales & xx  );
```

Outside the class decledration:

```
ostream& operator <<( ostream & xout, Sales & xx  ){
     cout << " Id: " << xx.id
        << " Name: " << xx.name;
     return xout;
}
```

A simple prototype for input / output overloading to cut and paste:

```
class Inventory   // Replace "Inventory" with new class name….
{
private:
    int price;
public:
     friend ostream& operator <<( ostream & xout, Inventory & xx  );
     friend istream& operator >>( istream & xout, Inventory & xx  );
};

ostream& operator <<( ostream & xout, Inventory & xx  ){
     cout<<  " Price:  " << setw(5)  << xx.price <<endl;
     return xout;
}

istream& operator >> ( istream & xout, Inventory & xx  )   {
   cout << " Price: " ;
   cin  >>  xx.price;
   return xout;
}
```

FUNCTION TEMPLATE

Every function template begins with the keyword template. This is followed by a list of **generic types** separated by commas and enclosed in angle brackets. Each generic type consists of the word **class** followed by an identifier to represent the generic type. In the example, T has been chosen as the identifier name (place holder) to represent the type.

```
template <class T>
T reverse (T x)
{          return –x;
}
```

Explicit type specification, use angle brackets after name, before arguement list:

```
  yyy  =  reverse < double> reverse ( kk);
```

Function templates can have more than one generic type:

```
Template  < Class T, Class U>
T  somefunction (  T xx, U yy)
{   Stuff
 }
```

Can explicit code two generic types:

```
yyy  = somefunction  <int, double> (   13, abc);
```

If you want to explicit code the second, you must explicit code the first…

If the teturn value does not match an argument, it can have an explicit specification by means of a generic type.

```
Template <class T >
 T  Max ( T  x[],  int len );      T is a place holder for "int", "float" etc.
```

```
Call could be      int  xx[5],  rval;          In this case, a version of max taking an int array and returning
                   Rval = max ( xx, 5);        an int max value is created by the compiler because xx is int.
```

Function Template – A blueprint for a family of functions, each having the same code, but different data types.
Generic argument – An argument whose data type is parameterized, as in a function or class template.

CLASS TEMPLATES

Class Template – A blueprint for a family of classes, each having the same structure and functions, but different data types.

```
template  <class T1 >
class CList                          //  T1 is a place holder for a class name such as CBox
{
private:
      T1  value1;
      T1  value2;
public:
      CList ( T1 v1, T1 v2 ) ;
      void display (  );
};

template < class T1 >                //   The constructor
CList <T1>  CList ( T1 v1, T1 v2 )
{   value1 = v1;
    value2 = v2;
}

template < class T1 >                //   A member function
void CList <T1>  display ( )
{   cout << value1;
    cout << value2;
}
```

Class instantions always have explicit type specification in the angle brackets:

```
CList  < CBox > my_list;      //  creates a list class for CBox objects called my_list
CList  < int  > int_list;     //Creates an integer list named int_list
```

Container Class – A template class that has been written to perform common class tasks; often used to manage groups of other classes.

EXCEPTION HANDELING

```
try{
  if ( height > 10)   throw  "Height too big!";
}
catch (const char * msg )
{   cout <<"an error occurred" <<   msg  << endl ;
}
```

You can have multiple catch functions by using different types as in

catch (const char * arg);
catch (const int count);

You can also throw and catch objects.
The catch block must be right after the try block.
Exceptions throw one type/object only.
Exceptions can be thrown by functions called within the try block.
Program exits if a handler is not found for a thrown exception.

catch (…) { *code block* **}**
Handles any attempt regardless of argument list, should be last try.
Three dots called "**ellipsis**"

Exception Specification
Used to define which exceptions can be thrown by a function, basically
for documentation only.

```
int jimroutine() throw ( char, double);
```
means this routine will only throw exceptions of type char or double. Program will exit if anything else thrown.

```
int jimroutine() throw ();  // will throw no exceptions…
```

OBJECT ORIENTED CONCEPTS

Object Based Supports data abstraction & classes, no inheritance.
Object Oriented Supports inheritance and Polymorphism
An **object** is a variable of a user defined type.

ENCAPULATION (Objects)

Encapulation is the mechanism that binds together code and data, and that keeps both safe from outside interference.
It is encapulation that allows creation of an object.

POLYMORPHISM (Overloading)

Polymorphism is the concept of "one interface, multiple methods" (Greek for many faces.)
Virtual function capacity is a classic polymorphic behavior.
C++ implements polymorphism through the use of class hierarchies, virtual functions and base class pointers.

C++ achieves polymorphism through the use of function overloading.
Function Overloading need to have functionsof the same name with different argument types.
Type and/or number of parameters of each overloaded function must differ.

C++ also achieves polymorphism through "operator overloading"

INHERITANCE (Classes)

Inheritance is the process by which one object can aquire the properties of another.

In C++ inheritance is supported by allowing one class to incorporate another class into its decleration.

First you define a "base class" and then "derived classes."

Format for class decleration: Class name : type parent

If there is a : with a type and parent, then we are creating a derived class.
Example class bouvier : private doggie
Creates the derived class "bouvier" derived as private from base class "doggie"

Type can be public, private or protected.
Protected means that a class member is not accessible by other, nonmember, elements of the program

facts to know --->
C++ is a “multiple inheritance” language, which means a sub class may belong to two or more super classes. (Java is not, for example.)

Derived Classes

Must have their own constructors and destructors.
Never inherited by a derived class: **Constructors, Destructors, Overloaded "="** operator functions, overloaded "**new**" operators

If a function is not virtual in the base class, and is redefined in the derived class,
a pointer to the derived class object will use the base class function rather than the derived class function.

When a deraived class object is created, the base class constructor is called first, then the derived class constructor.
When the object is destroyed, the derived class constructor is first, then the bas class constructor.
Class access specification:

Public	Can be read/written from anyplace!
Private	The default. (No direct access from anyplace.)
Protected.	Only derived class objects may access.

If the access specified is "private" then derived class objects cannot directly access base class variables and functions.

MULTIPLE INHERITANCE
C++ is a multiple inheritance language, which means a derived class can several base classes.
Example **class CPackage : public CBox , public CContents;** means CPackage object inherits from both CBox and CContents.
You can specify a **base class** as **virtual** for multiple inheritance situations:

```
Class CBox : public virtual CRockBottom
```

Specifying the base class as virtual instructs the compiler to make sure that the data members are not duplicated in a derived class.

VIRTUAL FUCTIONS
Declaring a class function "**virtual**" forces the compiler to use the function in the derived class if it exists and the object is a derived class object.
This is called "**late binding**" as opposed to "**early binding**."
Early binding -> Compile time
Late binding -> Run time.
Base class pointer to a virtual function forces use of the function according to the class of the object.
(If the object is base class, then the base class virtual function is used, if pointer is to a derived class object, the derived class member function is used.)

Example:
```
class mammal {
public:
        virtual int canine ( int weight)
}
```

Pure Virtual
A virtual function can be declared **pure** by placing **= 0** at the end of the decleration.
Example: virtual double volume () const **= 0** ;
A pure virtual function is declared but not defined in the base class.

If a function is not virtual and exists in the base class and the derived class, any pointer to object will result in the use of the base class function instead of the function in the derived class. (Not what you usually want!)
And, in fact, an object of the derived class will always use the base class function !

ABSTRACT CLASS Contains a pure virtual function.
Cannot be used to create an object.

Constructor functions cannot be virtual.
Destructor functions can and often should be virtual

Pointers to Class Objects

Base Class Pointer -- A pointer to a base class object can be assigned the address of a derived class object as well as a base class object.

If a virtual class function is called with a base class pointer, the function version in the derived class will be used if the object is of the derived class and the function of the base class will be used if the object is of the base class. This is very important!

(If the functions are not declared virtual, then the base class function will always be used. Usually bad!)

When a function is declared virtual in the base class, it is virtual in all layers of derived classes.

A virtual function does not have to be overridden in a derived class, in which case the parent class version of the member function is used.

When a base pointer points to a derived class object, it can only access elements inherited from the base, not those defined in the derived class.

"**Overridden**" is the term used when a version of a virtual function is created in a derived class.

***Example* --->**

```
class  doggie     {
public:                                  Anybody can see public data and use public methods.
     doggie();                           "Constructor" because name is the same as the class;
                                         the Constructor is optional.
     ~doggie();                          Tilde means "destructor", optional.

     void set_weight ( int wt  );        Means assign a weight value in private object
                                         memory for this particular doggie.
     int weight_is() const;              Means return the private value of weight, without
                                         changing any object internal values.
     virtual int fighting_drive();       The "fighting drive" is defined in the subsidiary class,
                                         that is, on a breed by breed basis.

protected:                               Only subsidiary classes can see this data or use
                                         these methods.
private:
     int    weight;
}

class Bouvier : public doggie {          Bouvier is a subsidiary class to doggie, or is a
public:                                  derived class of doggie.  "doggie" is the "base class."

private:

}

class Centauri : protected Bouvier
{

}
```

Keywords ->

inline	Expand code in place
operator	Used to override operators as in **operator+()**

MICROSOFT FOUNDATION CLASS

Windows API Application Program Interface
General areas: Windows API (Application Program Interface)
Internet Services
DAO Data Access Objects
ODBC
Active X ADO Active X data objects.
Collections

CObject is the base class for most of the MFC stuff.
It supports **serialization, run time class information, diagnostics.**

When we set up a SDI (Single Document Interface) we get 4 class definitions:

Application Class	**CWinApp**	The application itself, inherits through CwinThread.
Frame Window	**CFrameWnd**	(from cWnd) The "main frame" window.
Document cl	**CDocument**	(from cCmd Target) The document, that is, main data collection
View cl	**CView**	The portion of the document shown in the main window.

A document template class object is used to tie together a document, a view and a window.
SDI document CSingleDocTemplate
MDI CMultiDocTemplate.

A single "**application object**" of class derived from **CwinApp** is always created at global scope.
When the program starts, the application object member function **InitInstance()** is always called.

Message handlers used to respond to **WM** (**Windows Messages**) Example: WM_ COMMAND
WM_PAINT is the paint message, results in **OnDraw()** call in the view object.
GetDocument() returns a pointer to the document object..
OnDraw() is a view class function called in response to WM_PAINT to redraw screen.
GetDocument() is the view class virtual member function which gets a pointer to the associated document object.

AfxGetMainWnd() fetches the main window pointer (?)
AfxMessageBox(String,Options) display a message.
SDI Single Document Interface
MDI Multi Document Interface

Queued Message Sent to WinMain() loop for processing (such as char message.)
Non-Queued Message Sent directly to WinProc (such as resize window message.)

afxwin.h	definition header for MFC class
m_pszAppName	Pointer to application name.
m_nCmdShow	Defines window at start
p_pMainWnd	Pointer to main window, inherited from CwinApp

"Document" is a general term for the application data

& in front of a character in a menu item identifies the char as a short cut key.

COLLECTIONS A set of class templates for storing information, used in the document.

Array	**CArray**	An ordered arrangement of elements. Elements returned by int index value.
List	**CList**	Doubly linked list, 2 pointers, backward and forward. Most common.
Map	**CMap**	Unordered collection if items, associated with a key. Key usually a string. Works with **hashing**

To create a point array collection from a template:
CArray <Cpoint, CPoint & > MyPointArray; i.e. you define an object type and indicate a reference.

Type- safe collections are the preferred type. They are **"template based."**

Up to now, we are talking about **"collections of objects"**
There is a very similar set of **"collections of pointers to objects."**

GDI Graphical Device Interface makes display independent of hardware.
Device Context is a data structure that enables device display.
CDC Class Device Context
MM Mapping Mode MM_TEXT is the default, uses pixels.
CClientDC is a device context class derived from CDC that represents only the client area of a window.

InvalidateRect() is a member function inherited from the View class to redraw the window.

Brush Selection Example
CBrush * pOldBrush = static_const<CBrush *> (pDC->SelectStockObject(NULL_BRUSH));

pDC->MoveTo (x,y) move pen "pointer device context -> moveto member function"
pDC->Line To(x,y) draw line
A device context has a "brush"

MAPI Messaging API for communication with other computers
Windows Sockets Supports TCP/IP

Dialog Boxes

Modal	Locks up application until exit
Modeless	can shift focus back and forth.

Physical appearance defined in **Resource File** and Set up with **Resource Editor**

A dialog is based on a resource file and a class object derived from **CDialog**.

Derivation is: **CObject -> CCmdTarget -> CWnd -> CDialog -> CMyDialog**

The code to create a Modal dialog for an object aDlg is **aDlg.DoModal();**

Enum (IDD = IDD_PENWIDTH_DLG) is a "trick" to assign the value to IDD.
Used in the class definition header.

When the dialog is created, a **WM_INITDIALOG** message causes a call to the member function **OnInitDialog()** where you do the set up. (This is in additon to the call to the class constructor.)

Dialog Data Exchange and Validation.

DDX Do Data Exchange
DDV Do Dialog Validation
Provides transfer of information between a dialog and its common controls.
Example:

```
Void CScaleDialog::DoDataExchanage (CdataExchange * pDX) { }
```

This is "bi- directional", the parameter **pDX** controls the direction in which data is transferred.

Data Base Notes

ODBC Open Database Connectivity

RDO Remote Data Objects (Out of date…)

DAO DataBase Access objects - Primarily for Microsoft Access

OLD DB Object Linking and Embedding - Data baseOLE DB is a COM-based API;

ADO Active X Data Objects A higher level way to get at OLD DB mostly for Visual Basic & C#

UDA Universal Data Access

COM Common Object Module
Distributed Component Object Model (DCOM)

MDAC Microsoft Data Access Components

Microsoft Access
is "file based", not true client server.
Originally used the "JET" data base engine.
Can use Microsoft Data Engine (MSDE) True client server engine, fully SQL Server compatible.

Windows 2000 Server

IIS Internet Information Server The Microsoft Web Server
Sequel Server 2000

Microsoft Terminology…

BSTR - Basic **STR**ing a structured data type that contains a char string and length. Can have embedded zeros.

Variant - a structured data type that contains a value member and a data type member. A **Variant** may contain a wide range of other data types including another Variant, BSTR, Boolean, IDispatch or IUnknown pointer, currency, date, and so on.

globally unique identifier (GUID),

Style Notes

Scope of control should be 3 to 5 modules.

Cohesion how a function "holds together", should be high.

Coupling

Data coupling		the best
Data Structured	passes structs or objects	
Control	passes parameter that effects control	
External	uses global variables	very bad!
Common	global objects	
Pathological	two functions change each others data.	

Cohesion

Functional	all operations contribute to task	highest cohesion, good
Sequential	operations in certain order	
Communicational	performing an operation, such as input,sort,output	
Temporal	done at same time	
Procedural	done in sequence, butr not sharing data	
Logical	depends on a decision	
Coincidental		bad boy! bad boy! Worst one!

four kinds of functions, page 196 text

Inspector or Access function	looking at data, not changing data,,	get
Implementer or Mutator	modifies data	set
Auxiliary or Facilitator	performs action	sorting, searching
Manager function	creates or destroys objects	constructor, destructor

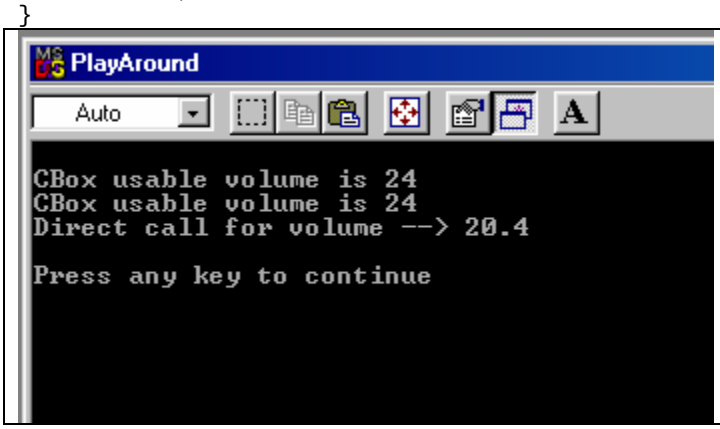
KEY TERMS

Accessor – A method that reveals information about an object, without changing its state.
Cohesion – Refers to how well the statements of a function relate to one another.
Constructor – A method designed to put an object into its initial state
Coupling – The complexity of the interface between two functions.
Facilitator– A method that performs an accessory service, such as searching or sorting.
File Guard – A technique that uses preprocessor directives to prevent a file from being included in another file more than once.
Manager– A method that creates or destroys an object
Mutator – A method that changes the internal state of an object.

Associativity – The rules governing how multiple operators of the same precedence are evaluated (left-to-right or right-to-left)
New operator – allocates memory dynamically.
Operator Overloading – Providing multiple function definitions for the same symbol.
Polymorphism – Many forms. Operator overloading is a weak example of polymorphism.
Precedence – The order in which multiple operators are evaluated in a single expression.

```
#include <iostream.h>

class mammal  {
public:

   virtual int jump( char *jj){
//   int jump( char *jj){
      cout << jj << "  in mammal Jump\n";
      return 444;
   }
};

class dog : public mammal {
public:
   int jump( char *jj){
      cout << jj << "  in dog Jump\n";
      return 33;
   }
};

/*****************************************************/
void  main( void )
{
   cout <<"starting\n";
   dog bootsie;
   mammal  pinkie;

   bootsie.jump( "Bootsie");
   pinkie.jump ( "Pinkie");

   mammal * bptr = &bootsie;

   bootsie .jump("Bootsie");
   bptr->jump("Bootsie");
   cout << "finished\n";
}
```

```
// EX10_06.CPP      FROM IVOR HORTON BOOK
// Behavior of inherited functions in a derived class
#include <iostream>
using namespace std;

// Listing 10_06-01
class CBox               // Base class
{
   public:
      // Function to show the volume of an object
      void ShowVolume() const
      {
         cout << endl
              << "CBox usable volume is " << Volume();
      }

      // Function to calculate the volume of a CBox object
      virtual double Volume() const            // PRODUCES UNEXPECTED RESULTS
      { return m_Length*m_Breadth*m_Height; }  // WHEN NOT VIRTUAL!

      // Constructor
      CBox(double lv = 1.0, double bv = 1.0, double hv = 1.0)
                            :m_Length(lv), m_Breadth(bv), m_Height(hv) {}

   protected:
      double m_Length;
      double m_Breadth;
      double m_Height;
};

// Listing 10_06-02
class CGlassBox: public CBox       // Derived class
{
   public:
      // Function to calculate volume of a CGlassBox
      // allowing 15% for packing
      double Volume() const
      { return 0.85*m_Length*m_Breadth*m_Height; }

      // Constructor
      CGlassBox(double lv, double bv, double hv): CBox(lv, bv, hv){}
};

int main()
{
   CBox myBox(2.0, 3.0, 4.0);            // Declare a base box
   CGlassBox myGlassBox(2.0, 3.0, 4.0);  // Declare derived box - same size

   myBox.ShowVolume();                   // Display volume of base box
   myGlassBox.ShowVolume();              // Display volume of derived box

//Jim additions -->
   cout << endl;
   cout <<  "Direct call for volume --> "<<   myGlassBox.Volume() << endl ;
   cout << endl;
   return 0;
}
```



I did a little research on the stuff we talked about at the end of class last night.

According to the Ivor Horton Book,

`->`	is the "indirect member access" operator	Can be overloaded
`.`	is the "direct member selection" operator.	Can be overloaded
`.*`	is the de-reference pointer to class member operator.	Can not be overloaded.

The Schildt book (C++ the Complete Reference) has a section on pointers to class members. (There are pointers to methods as well as data elements.)

If ClasssX is a class, then the decleration is

int ClassX::*data; // data declared a member pointer

the utilization, where obx is an object of ClassX, is along these lines:

data = & cl:: val ; // val a member variable of ClassX
cout << obx.*data << endl;

Schildt goes on to say that this is to be used only in rare instances, not as a normal practice. (This book has pretty good coverage of operator overloading.)

This is pretty obscure stuff, but the class text should get it right and use the correct terms to the extent it is mentioned.