C++ OVERVIEW AND OBJECT-ORIENTED PROGRAMMING

Abstract data types/user-defined data types

- allows the programmer to more closely model the problem space
- classes
- objects
- methods & attributes (functions & data members)
- encapsulation
- data/information hiding
- code reuse/maintenance

Object-oriented programming

- expression of relationships between classes
- inheritance
- code reuse/maintenance

C++ design goals

- strong support of object-oriented programming
- retain compatibility with C
- retain the performance of C

C++

- provides abstract data type support
- provides object-oriented programming support
- basis on C an asset and a drawback
- maintains basic C syntax and operators
- as an extension to C it is available wherever C is sold
- retains the performance of C

C++ - strengths over C

- Stronger type checking
- function prototyping is required
- in general, stronger typing rules than C
- provides support for development of abstract data types
- provides support for object-oriented programming

FUNDAMENTAL DATA TYPES

- there are four basic types of data, integer, floating-point, character and boolean
- character and boolean data types are really small integers, but usually are not treated as such
- signed and unsigned integer types available

Type	Size	Min	Max
boolean	1 byte	false / 0	true / 1
char	1 byte	0 / -128	255 / 127
short	2 bytes	-32768	32767
int	2,4 bytes	-2147483648	2147483647
long	4 bytes	-9×10^{18}	9×10^{18}
float	4 bytes ~ 7 digits	$\pm 1.0 \times 10^{-37}$	$\pm 3.4 \times 10^{+38}$
double	8 bytes ~ 14 digits	$\pm 1.0 \times 10^{-307}$	$\pm 1.8 \times 10^{+308}$
long double	12 bytes ~ 20 digits	$\pm 1.0 \times 10^{-4931}$	$\pm 1.0 \times 10^{+4932}$

IDENTIFIERS

- C++ identifiers follow a naming convention similar to C
  - may consist of letters, digits and the underscore
  - first character must be a letter or underscore, underscore is discouraged
  - no length limit
  - C++ is case-sensitive

KEYWORDS

asm	**do**	**if**	**return**	**typedef**
auto	**double**	inline	**short**	typeid
bool	dynamic_cast	**int**	**signed**	typename
break	**else**	**long**	**sizeof**	**union**
case	**enum**	mutable	**static**	**unsigned**
catch	explicit	namespace	static_cast	using
char	export	new	**struct**	virtual
class	**extern**	operator	**switch**	**void**
const	false	private	template	**volatile**
const_cast	**float**	protected	this	wchar_t
continue	**for**	public	throw	**while**
default	friend	**register**	true	
delete	**goto**	reinterpret_cast	try	

BASIC C++ PROGRAM STRUCTURE

- Basic program structure is the same as that of C.
- The function main( ) is where every C++ program begins execution.
- C++ uses braces { } to delimit the start/end of a block of code; for function definitions, class and method definitions, and statements containing sets of statements.
- Groups of instructions can be gathered together and named for ease of use and ease of programming. These “modules” are called functions. In Object-Oriented terminology they are referred to as methods.

COMMENTS

- C style comments /* … */ are allowed
- commenting to end-of-line using // is allowed

BASIC INPUT/OUTPUT

- Output is sent to some destination using the insertion/put-to operator <<. No explicit type information is required of the programmer.

- Input can also be read in from some source using the extraction/get-from operator >>. Again, no explicit type information is required.

Note: *iostream* must be included to utilize << and >>

Ex:

```
/*     FILE: io1.cpp     */

/* C++ comments and output with the insertion operator. */
#include <iostream>
#include <stdio.h>
using namespace std;

int main( )
{
  int x,y,z;

  float fx,fy,fz;

  x = 1;
  y = 2;
  z = 3;

  fx = 1.1;
  fy = 2.1;
  fz = 3.1;

  printf("%d\n",x);     // Easy one line comments
  printf("%d\n",y);     // C code works, C libraries available
  printf("%d\n",z);

  printf("%f\n",fx);
  printf("%f\n",fy);
  printf("%f\n",fz);

  float fsum = fx + fy + fz;   // Define variables anytime
  printf("%f\n",fsum);

  cout << x << endl;    // Easy "smart" output
  cout << y << endl;
  cout << z << endl;

  cout << fx << endl;
  cout << fy << endl;
  cout << fz << '\n';

  cout << fsum << '\n';

  return 0;
}
```

cont…

INTEGER VALUE REPRESENTATIONS

- Integer values can be represented in various bases, not just decimal.
- Decimal is the default interpretation.

Ex:

```
/*     FILE: data.cpp     */

/* Placing non-decimal integers in your code. */
#include <iostream>
#include <cstdio>
using namespace std;

int main( )
{
  int x = 175;

  printf("%d = %X = %o\n", x, x, x);

  x = 0xAF;
  cout << "x =  " << x << endl;
  x = 0257;
  cout << "x =  " << x << endl;
  x = 175;
  cout << "x =  " << x << endl;

  return 0;
}


/*    OUTPUT: data.cpp

       175 = AF = 257
       x =  175
       x =  175
       x =  175

*/
```

CONST KEYWORD FOR CONSTANT STORAGE

- Const can be used to enlist the compiler to try to enforce a no-write rule on storage.

Ex:

```
/*     FILE: constant.cpp     */

/* Avoiding #define. */
#include <iostream>
#include <cstdio>
using namespace std;

int main( )
{
  const int size = 5;
  int ar[size];

  for(int i=0; i<size ; i++)
    ar[i] = i+1;

  for(int i=0; i<size ; i++)
     cout << "ar[" << i << "] = " << ar[i] << endl;

  return 0;
}


/*    OUTPUT: constant.cpp

       ar[0] = 1
       ar[1] = 2
       ar[2] = 3
       ar[3] = 4
       ar[4] = 5

*/
```


TYPE CASTING

- Type casting can be done with a new "function-call" look.

Ex:

```
/*     FILE: cast.cpp     */

/* New style of casting. */
#include <iostream>
using namespace std;

int main( )
{
  int x,y;

  float fz;

  x = 1;
  y = 2;

  fz = x/y;

  cout << "x = " << x << endl;
  cout << "y = " << y << endl;
  cout << "fz = " << fz << endl;

  fz = float(x)/y;      // New cast style matches function call
  cout << "\nfz = " << fz << endl;

  return 0;
}



/*    OUTPUT: cast.cpp

       x = 1
       y = 2
       fz = 0

       fz = 0.5

*/
```

SIZES OF STORAGE

- Different data types have different storage allocations.
- The compiler can be queried about the allocation for a particular data type

Ex:

```
/*     FILE: sizes.cpp     */

/* Sizes of the various data types, on this implementation. */
#include <iostream>
using namespace std;

int main( )
{
  cout << "Size of 123: " << sizeof(123)
       << " bytes, " << sizeof(char)*8 << " bits." << endl;
  cout << "Size of char: " << sizeof(char)
       << " bytes, " << sizeof(char)*8 << " bits." << endl;
  cout << "Size of short: " << sizeof(short)
       << " bytes, " << sizeof(short)*8 << " bits." << endl;
  cout << "Size of int: " << sizeof(int)
       << " bytes, " << sizeof(int)*8 << " bits." << endl;
  cout << "Size of long: " << sizeof(long)
       << " bytes, " << sizeof(long)*8 << " bits." << endl;
  cout << "Size of float: " << sizeof(float)
       << " bytes, " << sizeof(float)*8 << " bits." << endl;
  cout << "Size of double: " << sizeof(double)
       << " bytes, " << sizeof(double)*8 << " bits." << endl;
  cout << "Size of long double: " << sizeof(long double)
       << " bytes, " << sizeof(long double)*8 << " bits." << endl;
  cout << "Size of bool: " << sizeof(bool)
       << " bytes, " << sizeof(bool)*8 << " bits." << endl;

  return 0;
}


/*    OUTPUT: sizes.cpp

        Size of 123: 4 bytes, 8 bits.
        Size of char: 1 bytes, 8 bits.
        Size of short: 2 bytes, 16 bits.
        Size of int: 4 bytes, 32 bits.
        Size of long: 4 bytes, 32 bits.
        Size of float: 4 bytes, 32 bits.
        Size of double: 8 bytes, 64 bits.
        Size of long double: 12 bytes, 96 bits.
        Size of bool: 1 bytes, 8 bits.

*/
```


C STANDARD LIBRARIES

- The C Standard Libraries are still available in C++.

Ex:

```
/*     FILE: math.cpp     */

/* Using the Math libraries, may have to link with -lm */
#include <iostream>
#include <cmath>
using namespace std;

int main( )
{
  int x;
  double root;

  cout << "Enter an integer, I'll give you its square root: " << endl;
  cin >> x;

  root = sqrt(double(x));
  cout << "The root is " << root << endl;

  return 0;
}


/*    OUTPUT: math.cpp

       Enter an integer, I'll give you its square root: 75

       The root is 8.66025

*/
```

INITIALIZATION

- The compiler can do some fancy initialization, especially on arrays.

Ex:

```
/*     FILE: array.cpp     */

/* The compiler does a lot of nice initialization.  */

#include <iostream>
using namespace std;

int main( )
{
  char name[ ] = "Jim Polzin";
  int ar[ ] = {1,2,3,4,5};

  cout << "The string is: " << name << endl;

  for(int i=0; i < 5; i++)        /* Display the array. */
    cout << "ar[" << i << "] = " << ar[i] << endl;

  return 0;
}


/*    OUTPUT: array.cpp

       The string is: Jim Polzin
       ar[0] = 1
       ar[1] = 2
       ar[2] = 3
       ar[3] = 4
       ar[4] = 5

*/
```


DEFAULT FUNCTION PARAMETERS

- Parameters at the end of a functions parameter list can have default values specified. The function can then be called without those parameters being provided and the default parameter value will be substituted for those parameters by the compiler.

Ex:

```
/*     FILE: default1.cpp     */

#include <iostream>
using namespace std;

struct COMPLEX{
   double Re;
   double Im;
};

COMPLEX Mult(COMPLEX a, COMPLEX b);
void print(COMPLEX c);
void init(COMPLEX *c,double r, double im);

int main( )
{
  COMPLEX c1, c2, cresult;

  init(&c1,2,3);
  init(&c2,3,2);

  cresult = Mult(c1, c2);

  cout << "Result of ";
  print(c1);
  cout << " * ";
  print(c2);
  cout << " = ";
  print(cresult);
  cout << endl;

  return 0;
}

COMPLEX Mult(COMPLEX a, COMPLEX b)
{
  COMPLEX result;

  result.Re = a.Re*b.Re - a.Im*b.Im;
  result.Im = a.Re*b.Im + a.Im*b.Re;

  return result;
}
```

cont…

INLINE FUNCTIONS

- Inlining of function code can be requested of the compiler by prefixing a function definition with the “inline” keyword.
- Inlining means the compiler will attempt to replace the function calls, to the inline designated function, with a copy of the functions code. This will eliminate function call overhead and improve performance. However, it will increase the size of your executable.
- Inlining gives the same functionality as macros in preprocessor directives but with the addition of stronger type checking.

Ex:

```
/*     FILE: inline.cpp     */

#include <iostream>
using namespace std;

#define SQUARE(X)  X * X

inline int square(int x){return x * x;}

int main( )
{
  int x;

  x = 5;

  cout << "x = " << x << ", x * x = " << square(x) << endl;
  cout << "x = " << x << ", x * x = " << SQUARE(x) << endl;

  cout << "3 + 5 = " << 3 + 5
       << ", (3 + 5) * (3 + 5) = " << square(3 + 5) << endl;
  cout << "3 + 5 = " << 3 + 5 << ", 3 + 5 * 3 + 5 = "
       << SQUARE(3 + 5) << endl;

  char *s = "Jim Polzin";

  cout << "s = " << s << " s*s = " << square(s) << endl;
  cout << "s = " << s << " s*s = " << SQUARE(s) << endl;

  return 0;
}

/*    OUTPUT: inline.cpp

       inline.cpp: In function `int main()':
       inline.cpp:24: passing `char *' to argument 1 of `square(int)' lacks a cast
       inline.cpp:25: invalid operands `char *' and `char *' to binary `operator *'

*/
```


REFERENCE VARIABLES

- C++ allows variables that are references to other variables.
- This is similar to, but not the same as, pointers in that the compiler handles the referencing and dereferencing with reference variables.
- Reference variables are defined by placing an & in front of the variable name when it is declared or defined.

Ex:

```
/*     FILE: ref1.cpp     */

#include <iostream>
using namespace std;

int main( )
{
  int x;
  int &r = x;           // r will be synonymous with x

  x = 5;
  cout << "x = " << x << endl;
  cout << "r = " << r << endl;

  r = 7;
  cout << "x = " << x << endl;
  cout << "r = " << r << endl;

  return 0;
}



/*    OUTPUT: ref1.cpp

        x = 5
        r = 5
        x = 7
        r = 7

*/
```

REFERENCE PARAMETERS

- Reference parameters can be defined for a function. Any value passed to a reference parameter will then be passed by reference. In C, all parameters were "value" parameters.
- This gives the called function access to the original data through the reference.
- Passing reference parameters allows a small piece of information to be passed to a function. Even when the value being reference is large, the reference is small. This allows for better performance.
- Since the compiler is "managing" the reference, there is reduced opportunity for programmer error.

Ex:

```
/*     FILE: ref2.cpp     */

/* Swap function using reference parameters */

#include <iostream>
using namespace std;

void swap(int &i1, int &i2);

int main( )
{
  int x,y;

  x = 5;
  y = 7;

  cout << "Before:" << endl;
  cout << "x = " << x << endl;
  cout << "y = " << y << endl;

  swap(x,y);

  cout << "After:" << endl;
  cout << "x = " << x << endl;
  cout << "y = " << y << endl;

  return 0;
}

void swap(int &i1, int &i2)
{
  int tmp;

  tmp = i1;
  i1 = i2;
  i2 = tmp;
}
```

cont…

REFERENCE RETURN VALUE

- A function can return a reference.
- This allows a small piece of data to be returned and gives access to the actual data, not a copy, to the caller of the function.
- An interesting result of this is that a function call result can be directly assigned to.

Ex:

```
/*     FILE: ref9.cpp     */

/* A reference return value can be assigned into. */
#include <iostream>
using namespace std;

int &ref1(int &i1);

int main( )
{
  int x;

  x = 5;

  cout << "Before: x = " << x << endl;

  ref1(x) = 7;

  cout << "After: x = " << x << endl;

  return 0;
}

int &ref1(int &i1)
{
  return i1;

}


/*    OUTPUT: ref9.cpp

       Before: x = 5
       After: x = 7

*/
```

FUNCTION OVERLOADING

- Different functions can have the same name. Producing a new function with the same name as an existing function is termed "overloading" the function.
- Functions are no longer distinguished by name alone but by the combination of the name and the types of parameters. This combination of name and types is referred to as the function's "signature."
- Function overloading gives the appearance of the same function being called for different types of data, since they have the same name.

Ex:

```
/*     FILE: over1.cpp     */

/*  Overloads a method named print( ).
    Print can be called for several different
    data types.
    Also creates a data type to model a complex number.
*/
#include <iostream>
using namespace std;

struct COMPLEX{
   float Re;
   float Im;
};

COMPLEX Mult(COMPLEX a, COMPLEX b);
void print(COMPLEX c);
void print(char *str);
void print(float f);

int main( )
{
  COMPLEX c1, c2, cresult;    // Note: struct keyword not required

  c1.Re = 2;
  c1.Im = 3;

  c2.Re = 2;
  c2.Im = 3;

  cresult = Mult(c1, c2);

  cout << "Result of ";
  print(c1);
  cout << " * ";
  print(c2);
  cout << " = ";
  print(cresult);
  cout << endl;
```

cont…

FUNCTION TEMPLATES

- A generic or "parameterized" function definition can be made in C++. This is called a function template.
- The compiler will produce a type-specific implementation of the function for each different call in the code.

Ex:

```
/*     FILE: functemp.cpp     */

/* Swap function template. */

#include <iostream>
using namespace std;

template <class t>
void swap(t &i1, t &i2)
{
  t tmp;

  tmp = i1;
  i1 = i2;
  i2 = tmp;
}

int main( )
{
  int x,y;

  x = 5;
  y = 7;

  cout << "Before:" << endl;
  cout << "x = " << x << endl;
  cout << "y = " << y << endl;

  swap(x,y);

  cout << "After:" << endl;
  cout << "x = " << x << endl;
  cout << "y = " << y << endl;

  return 0;
}

/*    OUTPUT: functemp.cpp

       Before:
       x = 5
       y = 7
       After:
       x = 7
       y = 5
*/
```

C++ PROVIDES CLASSES

- A class definition is a description of the commonality of a group of objects. This description includes the physical structure of the objects and the actions that can be performed by and on these objects.
- Basically a class is a user-defined type. Anywhere a C++ native type can be used, a class can be used.
- The goal of the class definition mechanism in C++ is to make programmer-defined types relatively easy to create, easy to use, and indistinguishable from inherent C++ types.

CONSTRUCTORS

- When an object of a class is created, a special “initializing” function is called. These initializers are called “Constructors.”
- Constructor calls are set up by the compiler so that newly created objects are initialized according to the initialization information provided at creation time.
- Constructors are defined with the same name as the class and with no return type.
- Constructors can be overloaded.

DEFAULT CONSTRUCTOR

- When no constructor has been defined for a class a default constructor is provided.
- The default constructor is called when no initialization information is provided when an object is created.
- When any other constructor has been defined the "default" default constructor is no longer provided. If a default constructor is still desired then it must be explicitly defined.
- A default constructor is produced by defining a constructor with no parameters.

THIS

- The implicit access to the invoking object is produced by an implicit pointer that is passed to all non-static member methods. The pointer is named *this*.
- The *this* pointer can be referenced explicitly within a non-static or instance method and at times it must be.
- The compiler handles all the implicit work of creating, passing and accessing members thru the *this* pointer.

STATIC MEMBERS

- The *static* keyword, within a class definition, makes a member have "class" scope.
- Static members are shared by all objects of the class or, are independent of any particular object.
- Static data members are independent of any particular object and store data common to all objects of the class. This data is available to objects of the class but is not part of each object, like an instance variable.
- Static methods are independent of any particular object and provide some functionality that is meaningful for the class itself. These methods can be called without or with an existing object.

INFORMATION HIDING – AN EXAMPLE

- Access specifiers allow a programmer to restrict the availability of information about, and access to, the actual physical structure of a class. This insulates existing code that utilizes the class from structural changes made to the class.
- get( )/set( ) methods allow an interface that buffers the consumer of the class from the actual implementation.

Ex:

```
/*     FILE: Student6.cpp     */

/*
   Changing the implementation of class Student.

   Notice that Student changes structurally but
   maintains its interface.  This protects the
   "client" code, main( ).

*/
#include <iostream>
using namespace std;

class Student{
    static long count;  // Keeps count of Student objects
                        // ... and generates id numbers.
    char * name;
    double gpa;
    long id;
    char grade;
    void setGrade( );

  public:
    static long getCount( );

    const char * getName( ) const;
    char * setName(char const * const);
    double getGpa( ) const;
    double setGpa(double);
    long getId( ) const;
    char getGrade( ) const;
    void display( ) const;
    void print( ) const;

    Student( );
    Student(char * n, double g);
};

long Student::count = 0;

Student::Student( )  // default constructor
{
  id = ++count;
}
```

cont…

DESTRUCTOR

- When objects of a class are destroyed the class destructor is called for each object immediately before it is destroyed.
- The name of the destructor is the class name preceded by a ~.
- In contrast to constructors there can be only one destructor for a class.
- A destructor primarily functions as an opportunity to “clean up” after an object, immediately before it is destroyed.

COPY CONSTRUCTOR

- When exact duplicates of an object need to be created a special constructor called the “Copy” constructor is called.
- A copy constructor is a constructor whose only parameter is a reference to an object of the same class.
- Like the default constructor, a copy constructor is provided by default if one has not been explicitly defined.

OPERATOR OVERLOADING

- In C++ most of the operators can be defined to operate on objects of newly defined classes. This is known as “operator overloading”.

Ex:

```
/*     FILE: op_over1.cpp     */

/*   The Mult( ) function can be turned into an operator definition.  */
#include <iostream>
using namespace std;

class COMPLEX{
  double Re;
  double Im;
public:
  void print( ) const;
  COMPLEX operator*(const COMPLEX & b) const;

  COMPLEX(double r, double im);
  COMPLEX(const COMPLEX &c)
  {
    Re = c.Re;
    Im = c.Im;
  }
  COMPLEX( )
  {
    Re = Im = 0;
  }
  ~COMPLEX( )
  {
  }
};

COMPLEX COMPLEX::operator*(const COMPLEX & b) const
{
  COMPLEX result;

  result.Re = Re*b.Re - Im*b.Im;
  result.Im = Re*b.Im + Im*b.Re;

  return result;
}

void COMPLEX::print( ) const
{
  cout << "(" << Re << " + " << Im << "i)" ;
}
```

cont…

OPERATOR OVERLOADING / FRIENDS

- In some cases operators and functions will not be invoked by members of the class but will still need access to private/protected data. These functions can be designated as “friend” functions and they will be given access to private/protected data.

Ex:

```
/*     FILE: op_over7.cpp     */

/*
   Definition of double times COMPLEX.

   Granting an "outside" the class function access to
   ... restricted data.  A friend.
*/
#include <iostream>
using namespace std;

class COMPLEX{
  double Re;
  double Im;
public:
  void print( ) const;
  COMPLEX operator*(const COMPLEX & b) const;
  COMPLEX operator*(const double &x) const;
  friend COMPLEX operator*(const double &x, const COMPLEX &c);

  COMPLEX operator-(const COMPLEX &b) const;
  COMPLEX operator-( ) const;

  COMPLEX(double r, double im);
  COMPLEX(const COMPLEX &c)
  {
    Re = c.Re;
    Im = c.Im;
  }
  COMPLEX( )
  {
    Re = Im = 0;
  }
  ~COMPLEX( )
  {
  }
};
```

cont…

PUT-TO OPERATOR OVERLOAD

- The put-to operator can be overloaded to make a user-defined type or class act more like an inherent C++ data-type.
- Also note: The need for a friend function, the parameter types and return type.

Ex:

```
/*     FILE: put_to.cpp     */

/*
   Overloading the put-to operator << for class COMPLEX.
*/
#include <iostream>
using namespace std;

class COMPLEX{
  double Re;
  double Im;
public:
  COMPLEX(double r, double im);
  COMPLEX(const COMPLEX &c)
  {
    Re = c.Re;
    Im = c.Im;
  }
  COMPLEX( )
  {
    Re = Im = 0;
  }
  ~COMPLEX( )
  {
  }

  COMPLEX operator*(const COMPLEX & b) const;
  COMPLEX operator*(const double &x) const;
  friend COMPLEX operator*(const double &x, const COMPLEX &c);

  COMPLEX operator-(const COMPLEX &b) const;
  COMPLEX operator-( ) const;

  friend ostream& operator<<(ostream&,const COMPLEX &);
};

COMPLEX::COMPLEX(double r, double im)
{
  Re = r;
  Im = im;
}
```

cont…

FILE I/O

- I/O in C++ is based on the concept of streams. The two C++ connections to standard input and output, *cin/cout*, are streams. A stream is considered a connection through which bytes or data "stream" into or out of your program.
- To read and write data to a file the stream concept is also used. File stream information is contained in a header file called *fstream*.
- The basic techniques used with *cin/cout* will still be used with files.

Ex:

```
/*     FILE: File1.cpp     */

// Write some output to a file.

#include <fstream>

using std::ofstream;
using std::endl;

int main( )
{
  ofstream outs;

  outs.open("File1.out");

  outs << "Hello world!" << endl;

  outs.close( );
}


/*   OUTPUT: File1.OUT

       Hello world!

*/
```

INHERITANCE

- A new class can be created or derived from an existing class.
- The class used as the “basis” of the new class is called the “base class.” The new class is referred to as a “derived” class.
- A derived class begins as the structure and functionality of the base class, with only additions to the base class needing to be defined.
- The new derived class is related to the base class in that it “is-a” example of the base class, since it has inherited everything from the base class. This relationship will be capitalized on when we are using references and pointers to refer to these related objects.

POLYMORPHIC/VIRTUAL FUNCTIONS

- Polymorphism is the ability of a function to “change” according to what object is currently invoking it.
- When a base class pointer is used to invoke a function, the actual class of object invoking the function may not be known until run-time.
- In order for the correct function to be called, “dynamic-binding” must be performed or a “run-time” determination must be made.
- This dynamic-binding or run-time determination is produced in C++ with virtual functions.

Ex:

```
/*     FILE: ./shapes3/point.h     */

#ifndef _point_h
#define _point_h

#include <iostream>

using std::ostream;

class point{
    double x,y;

  public:
    point( )
    { x=y=0;}
    point(double xvalue, double yvalue);

    void setPoint(double new_x, double new_y);
    void setX(double new_x);
    void setY(double new_y);
    double getX( ) const;
    double getY( ) const;

    void move(double x, double y);
    void shift(double dx, double dy);

    void print(ostream&)const;
    friend ostream & operator<<(ostream & os, const point& p);
};
#endif
```

cont…

VIRTUAL OPERATOR<<, A VIRTUAL FRIEND FUNCTION

- Friend functions pose a “special” challenge in regards to polymorphism since they are not member functions.
- A slight “trick” can be applied to the design of the friend functions so that polymorphic behavior can be produced when they are invoked through a base-class reference

Ex:

```
/*     FILE: ./shapes6/shape.h     */

#ifndef _shape_h
#define _shape_h

#include <iostream>
using std::cout;
using std::endl;
using std::ostream;

class shape{
    double x,y;

  public:
    shape( )
    { x=y=0;}
    shape(double xvalue, double yvalue);

    void setShape(double new_x, double new_y);
    void setX(double new_x);
    void setY(double new_y);
    double getX( ) const;
    double getY( ) const;

    virtual void move(double x, double y) = 0;
    virtual void shift(double dx, double dy) = 0;
    virtual void draw( ) = 0;
    virtual void rotate(double r) = 0;

    virtual void print(ostream&)const;
    friend ostream & operator<<(ostream & os, const shape& s);
};
#endif
```

cont…