C++

C/C++ for Scientists and Engineers

Presented by:

Jim Polzin

e-mail: james.polzin@normandale.edu
otto: http://otto.normandale.edu

TABLE OF CONTENTS

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

Ex:

```
/*     FILE: example1.cpp     */

/* Hello world!  ++ */
#include <iostream>
using namespace std;

int main( )
{
  cout << "Hello world!";

  return 0;
}


/*    OUTPUT: example1.cpp

       Hello world!
*/
```

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…

```
/*    OUTPUT: io1.cpp

       1
       2
       3
       1.100000
       2.100000
       3.100000
       6.300000
       1
       2
       3
       1.1
       2.1
       3.1
       6.3

*/
```

Ex:

```
/*     FILE: io2.cpp     */

/* Reading in data with the extraction operator. */
#include <iostream>
#include <stdio.h>
using namespace std;

int main( )
{
  int x;

  float fx;

  cout << "Please enter an integer: " << endl;   // Prompt
  cin >> x;                                      // Read

  cout << "Please enter a float: " << endl;   // Prompt
  cin >> fx;                                  // Read

  cout << endl;
  cout << "x = " << x << endl;
  cout << "fx = " << fx << endl;

  return 0;
}


/*    OUTPUT: io2.cpp

        Please enter an integer: 34

        Please enter a float: 3.75


        x = 34
        fx = 3.75

*/
```

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

*/
```

Ex:

```
/*     FILE: constant2.cpp     */

/* Avoiding #define. The compiler is watching. */
#include <iostream>

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;


  cout << "size = " << size << endl;

  size = 11;

  cout << "now size = " << size << endl;

  return 0;
}


/*    OUTPUT: constant2.cpp

       constant2.cpp: In function `int main()':
       constant2.cpp:20: assignment of read-only variable `size'

*/
```

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.

*/
```

Ex:

```
/*     FILE: sizes2.cpp     */

/* Controlling the size of constants. */
#include <iostream>
using namespace std;

int main( )
{
  cout << "Size of 123: " << sizeof(123)
       << " bytes, " << sizeof(123)*8 << " bits." << endl;
  cout << "Size of 123: " << sizeof(123L)
       << " bytes, " << sizeof(123L)*8 << " bits." << endl;
  cout << "Size of 3.21: " << sizeof(3.21)
       << " bytes, " << sizeof(3.21)*8 << " bits." << endl;
  cout << "Size of 3.21f: " << sizeof(3.21f)
       << " bytes, " << sizeof(3.21f)*8 << " bits." << endl;
  cout << "Size of 3.21L: " << sizeof(3.21L)
       << " bytes, " << sizeof(3.21L)*8 << " bits." << endl;

  return 0;
}


/*    OUTPUT: sizes2.cpp

        Size of 123: 4 bytes, 32 bits.
        Size of 123: 4 bytes, 32 bits.
        Size of 3.21: 8 bytes, 64 bits.
        Size of 3.21f: 4 bytes, 32 bits.
        Size of 3.21L: 12 bytes, 96 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…

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

void init(COMPLEX *c,double r, double im)
{
  c->Re = r;
  c->Im = im;
}


/*    OUTPUT: default1.cpp

        Result of (2 + 3i) * (3 + 2i) = (0 + 13i)

*/
```

Ex:

```
/*     FILE: default2.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 = 0);

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

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

  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;
}

void print(COMPLEX c)
{
  cout << "(" << c.Re << " + " << c.Im << "i)" ;
}

void init(COMPLEX *c,double r, double im)
{
  c->Re = r;
  c->Im = im;
}

/*    OUTPUT: default2.cpp

        Result of (2 + 3i) * (3 + 0i) = (6 + 9i)

*/
```

Ex:

```
/*     FILE: default3.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 = 0, double im = 0);

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

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

  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;
}

void print(COMPLEX c)
{
  cout << "(" << c.Re << " + " << c.Im << "i)" ;
}

void init(COMPLEX *c,double r, double im)
{
  c->Re = r;
  c->Im = im;
}

/*    OUTPUT: default3.cpp

        Result of (2 + 3i) * (0 + 0i) = (0 + 0i)

*/
```

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 *'

*/
```

Ex:

```
/*     FILE: inline2.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: inline2.cpp

        x = 5, x * x = 25
        x = 5, x * x = 25
        3 + 5 = 8, (3 + 5) * (3 + 5) = 64
        3 + 5 = 8, 3 + 5 * 3 + 5 = 23

*/
```

Ex:

```
/*     FILE: inline3.cpp     */

#include <iostream>
using namespace std;

struct COMPLEX{
   float Re;
   float Im;
};

inline 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;
}

inline void init(COMPLEX *c,float r, float im)
{
  c->Re = r;
  c->Im = im;
}

void print(COMPLEX c)
{
  cout << "(" << c.Re << " + " << c.Im << "i)" ;
}


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

  init(&c1,2,3);
  init(&c2,3,4);

  cresult = Mult(c1, c2);

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

  return 0;
}


/*    OUTPUT: inline3.cpp

       Result of (2 + 3i) * (3 + 4i) = (-6 + 17i)

*/
```

Ex:

```
/*     FILE: inline4.cpp     */

#include <iostream>
#include "inline4.h"
using namespace std;

void print(COMPLEX c)
{
  cout << "(" << c.Re << " + " << c.Im << "i)" ;
}


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

  init(&c1,2,3);
  init(&c2,3,4);

  cresult = Mult(c1, c2);

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

  return 0;
}


/*    OUTPUT: inline4.cpp

       Result of (2 + 3i) * (3 + 4i) = (-6 + 17i)

*/
```

cont…

```
/*     FILE: inline4.h     */

#include <iostream>
using namespace std;

struct COMPLEX{
   float Re;
   float Im;
};

inline 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;
}

inline void init(COMPLEX *c,float r, float im)
{
  c->Re = r;
  c->Im = im;
}
```

Ex:

```
/*     FILE: inline5.cpp     */

#include <iostream>
#include "inline5.h"
using namespace std;

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

  init(&c1,2,3);
  init(&c2,3,4);

  cresult = Mult(c1, c2);

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

  return 0;
}


/*    OUTPUT: inline5.cpp

       Result of (2 + 3i) * (3 + 4i) = (-6 + 17i)

*/
```

cont…

```
/*     FILE: inline5.h     */

#include <iostream>
using namespace std;

struct COMPLEX{
   float Re;
   float Im;
};

void print(COMPLEX c);

inline 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;
}

inline void init(COMPLEX *c,float r, float im)
{
  c->Re = r;
  c->Im = im;
}
```

cont…

```
/*     FILE: inline5_1.cpp     */

#include <iostream>
#include "inline5.h"
using namespace std;

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

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

*/
```

Ex:

```
/*     FILE: ref1_1.cpp     */

#include <iostream>
using namespace std;

int main( )
{
  int x;
  int y;
  int &r = x;           // r will be synonymous with x
                        // r cannot be made to refer to
                        // ... another variable
  x = 5;
  y = 12;
  cout << "x = " << x << endl;
  cout << "r = " << r << endl;

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

  return 0;
}



/*    OUTPUT: ref1_1.cpp

       x = 5
       r = 5
       x = 12
       r = 12

*/
```

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…

```
/*    OUTPUT: ref2.cpp

       Before:
       x = 5
       y = 7
       After:
       x = 7
       y = 5

*/
```

Ex:

```
/*     FILE: ref3.cpp     */

/* Swap function using pointers.       */

/* Side note: pointers are passed by value. */

#include <iostream>

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;

}


/*    OUTPUT: ref3.cpp

        Before:
        x = 5
        y = 7
        After:
        x = 7
        y = 5

*/
```

Ex:

```
/*     FILE: ref4.cpp     */

// More efficient COMPLEX Mult( ), less data passed to Mult( )

#include <iostream>
using namespace std;

struct COMPLEX{
   float Re;
   float Im;
};

COMPLEX Mult(COMPLEX &a, COMPLEX &b);
void print(COMPLEX c);

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

  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;

  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;
}

void print(COMPLEX c)
{
  cout << "(" << c.Re << " + " << c.Im << "i)" ;
}


/*    OUTPUT: ref4.cpp

        Result of (2 + 3i) * (2 + 3i) = (-5 + 12i)

*/
```

Ex:

```
/*     FILE: ref5.cpp     */

// Efficient COMPLEX Add( ) is passed a reference to each operand/parameter
// The "reference-ness" of the parameters can be protected with const

// Add( ) produces the correct result but produces a nasty side effect

#include <iostream>
using namespace std;

struct COMPLEX{
   float Re;
   float Im;
};

COMPLEX Add(COMPLEX &a, COMPLEX &b);
void print(COMPLEX c);

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

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

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

  cresult = Add(c1, c2);

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

  return 0;
}

COMPLEX Add(COMPLEX &a, COMPLEX &b)
{
  a.Re = a.Re + b.Re;
  a.Im = a.Im + b.Im;

  return a;
}

void print(COMPLEX c)
{
  cout << "(" << c.Re << " + " << c.Im << "i)" ;
}


/*    OUTPUT: ref5.cpp

        Result of (4 + 6i) + (2 + 3i) = (4 + 6i)

*/
```

Ex:

```
/*     FILE: ref6.cpp     */

// Efficient COMPLEX Add( ) is passed a reference to each operand/parameter
// The "reference-ness" of the parameters can be protected with const

// Add( ) would produce the correct result but tries to produce a nasty
// ... side effect but, the compiler is watching over the parameters

#include <iostream>
using namespace std;

struct COMPLEX{
   float Re;
   float Im;
};

COMPLEX Add(const COMPLEX &a, const COMPLEX &b);
void print(const COMPLEX &c);

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

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

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

  cresult = Add(c1, c2);

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

  return 0;
}

COMPLEX Add(const COMPLEX &a, const COMPLEX &b)
{
  a.Re = a.Re + b.Re;
  a.Im = a.Im + b.Im;

  return a;
}

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

cont…

```
/*    OUTPUT: ref6.cpp

        ref6.cpp: In function `struct COMPLEX Add(const COMPLEX &, const COMPLEX &)':
        ref6.cpp:43: assignment of member `COMPLEX::Re' in read-only structure
        ref6.cpp:44: assignment of member `COMPLEX::Im' in read-only structure

*/
```

Ex:

```
/*     FILE: ref7.cpp     */

// Efficient COMPLEX Add( ) is passed a reference to each operand/parameter
// The "reference-ness" of the parameters can be protected with const

// Add( ) produces the correct result and the compiler watches over the parameters

// print( ) now uses "const &" since it shouldn't alter the parameter

#include <iostream>
using namespace std;

struct COMPLEX{
   float Re;
   float Im;
};

COMPLEX Add(const COMPLEX &a, const COMPLEX &b);
void print(const COMPLEX &c);

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

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

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

  cresult = Add(c1, c2);

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

  return 0;
}

COMPLEX Add(const COMPLEX &a, const COMPLEX &b)
{
  COMPLEX result;

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

  return result;
}

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

cont…

```
/*    OUTPUT: ref7.cpp

       Result of (2 + 3i) + (2 + 3i) = (4 + 6i)

*/
```

Ex:

```
/*     FILE: ref8.cpp     */

// The caller could defend against reference parameters

#include <iostream>
using namespace std;

struct COMPLEX{
   float Re;
   float Im;
};

COMPLEX Add(COMPLEX &a, COMPLEX &b);
void print(const COMPLEX &c);

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

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

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

  const COMPLEX & p1 = c1;
  const COMPLEX & p2 = c2;

  cresult = Add(p1, p2);

// or

  cresult = Add((const COMPLEX)c1, (const COMPLEX)c2);

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

  return 0;
}

COMPLEX Add(COMPLEX &a, COMPLEX &b)
{
  a.Re = a.Re + b.Re;
  a.Im = a.Im + b.Im;

  return a;
}

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

cont…

```
/*    OUTPUT: ref8.cpp

        ref8.cpp: In function `int main()':
        ref8.cpp:27: conversion from `const COMPLEX' to `COMPLEX &' discards qualifiers
        ref8.cpp:11: in passing argument 1 of `Add(COMPLEX &, COMPLEX &)'
        ref8.cpp:27: conversion from `const COMPLEX' to `COMPLEX &' discards qualifiers
        ref8.cpp:11: in passing argument 2 of `Add(COMPLEX &, COMPLEX &)'
        ref8.cpp:31: initialization of non-const reference type `struct COMPLEX &'
        ref8.cpp:31: from rvalue of type `COMPLEX'
        ref8.cpp:11: in passing argument 1 of `Add(COMPLEX &, COMPLEX &)'
        ref8.cpp:31: initialization of non-const reference type `struct COMPLEX &'
        ref8.cpp:31: from rvalue of type `COMPLEX'
        ref8.cpp:11: in passing argument 2 of `Add(COMPLEX &, COMPLEX &)'

*/
```

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

*/
```

Ex:

```
/*     FILE: ref10.cpp     */

/* A reference return value can be assigned into. */

// A fancier function.

#include <iostream>
using namespace std;

int & ref1(int & i1, int & i2);

int main( )
{
  int x,y;

  x = 5;
  y = 1;

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

  ref1(x,y) = 7;

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

  return 0;
}

int & ref1(int & i1, int & i2)
{
  if(i1 <= i2)
    return i1;
  else
    return i2;
}


/*    OUTPUT: ref10.cpp

        Before: x = 5 y = 1
        After: x = 5 y = 7

*/
```

Ex:

```
/*     FILE: ref11.cpp     */

/* A reference return value can be assigned into. */

// This could also be done with skillful use of pointers.

#include <iostream>
using namespace std;

int &ref1(int &i1);
int *ref2(int *i1);

int main( )
{
  int x;

  x = 5;

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

  ref1(x) = 7;

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

  *ref2(&x) = 9;

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

  return 0;
}

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

}

int *ref2(int *i1)
{
  return i1;

}


/*    OUTPUT: ref11.cpp

       Before: x = 5
       After: x = 7
       After again: x = 9

*/
```

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…

```
  print("Float example:");
  cout << endl;
  cout << "Result of ";
  print(4.5);
  cout << " * ";
  print(3);
  cout << " = ";
  print(4.5 * 3);
  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;
}

void print(COMPLEX c)
{
  cout << "(" << c.Re << " + " << c.Im << "i)" ;
}

void print(char *str)
{
  cout << str ;
}

void print(float f)
{
  cout << f ;
}



/*    OUTPUT: over1.cpp

        Result of (2 + 3i) * (2 + 3i) = (-5 + 12i)
        Float example:
        Result of 4.5 * 3 = 13.5

*/
```

Ex:

```
/*     FILE: over1_1.cpp     */

/*   Overloaded Add( ) method.  */
#include <iostream>
using namespace std;

struct COMPLEX{
   double Re;
   double Im;
};

COMPLEX Add(COMPLEX a, COMPLEX b);
COMPLEX Add(COMPLEX a, double b);
COMPLEX Add(double a, COMPLEX b);
void print(COMPLEX c);

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

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

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

  cresult = Add(c1, c2);

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

  cresult = Add(c1, 5.5);

  cout << "Result of ";
  print(c1);
  cout << " + "
       << 5.5
       << " = ";
  print(cresult);
  cout << endl;

  cresult = Add(3.3, c2);

  cout << "Result of "
       << 3.3
       << " + ";
  print(c2);
  cout << " = ";
  print(cresult);
  cout << endl;

  return 0;
}
```

cont…

```
COMPLEX Add(COMPLEX a, COMPLEX b)
{
  COMPLEX result;

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

  return result;
}

COMPLEX Add(COMPLEX a, double b)
{
  COMPLEX result;

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

  return result;
}

COMPLEX Add(double a, COMPLEX b)
{
  COMPLEX result;

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

  return result;
}

void print(COMPLEX c)
{
  cout << "(" << c.Re << " + " << c.Im << "i)" ;
}


/*    OUTPUT: over1_1.cpp

        Result of (2 + 3i) + (2 + 3i) = (4 + 6i)
        Result of (2 + 3i) + 5.5 = (7.5 + 3i)
        Result of 3.3 + (2 + 3i) = (5.3 + 3i)

*/
```

Ex:

```
/*     FILE: over1_2.cpp     */
/*   Overloaded Add( ) method.  */
// Shrewd method definition to reuse code

#include <iostream>
using namespace std;

struct COMPLEX{
   double Re, Im;
};

COMPLEX Add(COMPLEX a, COMPLEX b);
COMPLEX Add(COMPLEX a, double b);
COMPLEX Add(double a, COMPLEX b);
void print(COMPLEX c);

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

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

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

  cresult = Add(c1, c2);

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

  cresult = Add(c1, 5.5);

  cout << "Result of ";
  print(c1);
  cout << " + "
       << 5.5
       << " = ";
  print(cresult);
  cout << endl;

  cresult = Add(3.3, c2);

  cout << "Result of "
       << 3.3
       << " + ";
  print(c2);
  cout << " = ";
  print(cresult);
  cout << endl;

  return 0;
}
```

cont…

```
COMPLEX Add(COMPLEX a, COMPLEX b)
{
  COMPLEX result;

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

  return result;
}

COMPLEX Add(COMPLEX a, double b)
{
  COMPLEX result;

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

  return result;
}

COMPLEX Add(double a, COMPLEX b)
{
  return Add(b,a);
}

void print(COMPLEX c)
{
  cout << "(" << c.Re << " + " << c.Im << "i)" ;
}


/*    OUTPUT: over1_2.cpp

        Result of (2 + 3i) + (2 + 3i) = (4 + 6i)
        Result of (2 + 3i) + 5.5 = (7.5 + 3i)
        Result of 3.3 + (2 + 3i) = (5.3 + 3i)

*/
```

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
*/
```

Ex:

```
/*     FILE: functmp2.cpp     */

/* Swap function template, several types. */

#include <iostream>
using namespace std;

template <class t>
void swap(t &, t &);

int main( )
{
  int x,y;
  double fx,fy;

  x = 5;
  y = 7;
  fx = 5.5;
  fy = 7.7;

  cout << "Before:" << endl;
  cout << "x = " << x << ", fx = " << fx << endl;
  cout << "y = " << y << ", fy = " << fy << endl;

  swap(x,y);
  swap(fx,fy);

  cout << "After:" << endl;
  cout << "x = " << x << ", fx = " << fx << endl;
  cout << "y = " << y << ", fy = " << fy << endl;

  return 0;
}

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

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



/*    OUTPUT: functmp2.cpp

        Before:
        x = 5, fx = 5.5
        y = 7, fy = 7.7
        After:
        x = 7, fx = 7.7
        y = 5, fy = 5.5

*/
```

Ex:

```
/*     FILE: functmp3.cpp     */

/* Swap function template, several types. */

#include <iostream>
#include "inline5.h"
using namespace std;

template <class t>
void swapfunc(t &, t &);

int main( )
{
  int x,y;
  double fx,fy;
  COMPLEX c1,c2;

  x = 5;
  y = 7;
  fx = 5.5;
  fy = 7.7;
  init(&c1,1,2);
  init(&c2,3,4);

  cout << "Before:" << endl;
  cout << "x = " << x << ", fx = " << fx << ", c1 = ";
  print(c1);
  cout << endl;
  cout << "y = " << y << ", fy = " << fy << ", c2 = ";
  print(c2);
  cout << endl;

  swapfunc(x,y);
  swapfunc(fx,fy);
  swapfunc(c1,c2);

  cout << "After:" << endl;
  cout << "x = " << x << ", fx = " << fx << ", c1 = ";
  print(c1);
  cout << endl;
  cout << "y = " << y << ", fy = " << fy << ", c2 = ";
  print(c2);
  cout << endl;

  return 0;
}

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

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

cont…

```
/*    OUTPUT: functmp3.cpp

       Before:
       x = 5, fx = 5.5, c1 = (1 + 2i)
       y = 7, fy = 7.7, c2 = (3 + 4i)
       After:
       x = 7, fx = 7.7, c1 = (3 + 4i)
       y = 5, fy = 5.5, c2 = (1 + 2i)

*/
```

Ex:

```
/*     FILE: functmp4.cpp     */

/*
   Overloaded print( ) method using a template.
*/

// Errors: What's wrong?

#include <iostream>
using namespace std;

template <class t>
void print(t &value);

struct COMPLEX{
   float Re;
   float Im;
};

COMPLEX Mult(COMPLEX a, COMPLEX b);

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

  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;

  print("Float example:");
  cout << endl;
  cout << "Result of ";
  print(4.5);
  cout << " * ";
  print(3);
  cout << " = ";
  print(4.5 * 3);
  cout << endl;

  return 0;
}
```

cont…

```
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;
}

template <class t>
void print(t &value)
{
  cout << value;
}


/*    OUTPUT: functmp4.cpp

       functmp4.cpp: In function `int main()':
       functmp4.cpp:43: could not convert `0x0000000000000000009001400000000000000000'
          to `double&'
       functmp4.cpp:11: in passing argument 1 of `void print(t&) [with t = double]'
       functmp4.cpp:45: could not convert `3' to `int&'
       functmp4.cpp:11: in passing argument 1 of `void print(t&) [with t = int]'
       functmp4.cpp:47: could not convert `0x0000000000000000d802400000000000000000'
          to `double&'
       functmp4.cpp:11: in passing argument 1 of `void print(t&) [with t = double]'
       functmp4.cpp: In function `void print(t&) [with t = COMPLEX]':
       functmp4.cpp:33:   instantiated from here
       functmp4.cpp:66: no match for `std::ostream& << COMPLEX&' operator
       C:/Programs/Dev-Cpp/include/c++/bits/ostream.tcc:55: candidates are:
          std::basic_ostream<_CharT, _Traits>& std::basic_ostream<_CharT,
          _Traits>::operator<<(std::basic_ostream<_CharT,
          _Traits>&(*)(std::basic_ostream<_CharT, _Traits>&)) [with _CharT = char,
          _Traits = std::char_traits<char>]
       C:/Programs/Dev-Cpp/include/c++/bits/ostream.tcc:77:
          std::basic_ostream<_CharT, _Traits>& std::basic_ostream<_CharT,
          _Traits>::operator<<(std::basic_ios<_CharT,
          _Traits>&(*)(std::basic_ios<_CharT, _Traits>&)) [with _CharT = char, _Traits
          = std::char_traits<char>]
.
.
.
       C:/Programs/Dev-Cpp/include/c++/ostream:251:
          std::basic_ostream<char, _Traits>& std::operator<<(std::basic_ostream<char,
          _Traits>&, const unsigned char*) [with _Traits = std::char_traits<char>]

*/
```

Ex:

```
/*     FILE: functmp5.cpp     */

/*
   Overloaded print( ) method.
*/

// Errors: What's wrong?
//  - value parameters that are numeric constants are not
//    ... alterable storage.

#include <iostream>
using namespace std;

template <class t>
void print(const t &value);

struct COMPLEX{
   float Re;
   float Im;
};

COMPLEX Mult(COMPLEX a, COMPLEX b);

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

  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;

  print("Float example:");
  cout << endl;
  cout << "Result of ";
  print(4.5);
  cout << " * ";
  print(3);
  cout << " = ";
  print(4.5 * 3);
  cout << endl;

  return 0;
}
```

cont…

```
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;
}

template <class t>
void print(const t &value)
{
  cout << value;
}


/*    OUTPUT: functmp5.cpp

        functmp5.cpp: In function `void print(const t&) [with t = COMPLEX]':
        functmp5.cpp:35:   instantiated from here
        functmp5.cpp:68: no match for `std::ostream& << const COMPLEX&' operator
        C:/Programs/Dev-Cpp/include/c++/bits/ostream.tcc:55: candidates are:
           std::basic_ostream<_CharT, _Traits>& std::basic_ostream<_CharT,
           _Traits>::operator<<(std::basic_ostream<_CharT,
           _Traits>&(*)(std::basic_ostream<_CharT, _Traits>&)) [with _CharT = char,
           _Traits = std::char_traits<char>]
        C:/Programs/Dev-Cpp/include/c++/bits/ostream.tcc:77:
           std::basic_ostream<_CharT, _Traits>& std::basic_ostream<_CharT,
           _Traits>::operator<<(std::basic_ios<_CharT,
           _Traits>&(*)(std::basic_ios<_CharT, _Traits>&)) [with _CharT = char, _Traits
           = std::char_traits<char>]
.
.
.
        C:/Programs/Dev-Cpp/include/c++/ostream:246:
           std::basic_ostream<char, _Traits>& std::operator<<(std::basic_ostream<char,
           _Traits>&, const signed char*) [with _Traits = std::char_traits<char>]
        C:/Programs/Dev-Cpp/include/c++/ostream:251:
           std::basic_ostream<char, _Traits>& std::operator<<(std::basic_ostream<char,
           _Traits>&, const unsigned char*) [with _Traits = std::char_traits<char>]

*/
```

Ex:

```
/*     FILE: functmp6.cpp     */

/*   Overloaded print( ) method.   */

// Errors: What's wrong?
// - value parameters that are numeric constants are not
//   ... alterable storage.
// - Compiler cannot determine how to insert our COMPLEX
//   ... into an output stream. A "specialized" template can be defined.

#include <iostream>
using namespace std;

template <class t>
void print(const t &value);

struct COMPLEX{
   float Re;
   float Im;
};

template <> void print<COMPLEX>(const COMPLEX &c);

COMPLEX Mult(COMPLEX a, COMPLEX b);

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

  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;

  print("Float example:");
  cout << endl;
  cout << "Result of ";
  print(4.5);
  cout << " * ";
  print(3);
  cout << " = ";
  print(4.5 * 3);
  cout << endl;

  return 0;
}
```

cont…

```
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;
}

template <class t>
void print(const t &value)
{
  cout << value;
}

template <> void print<COMPLEX>(const COMPLEX &c)
{
  cout << "(" << c.Re << " + " << c.Im << "i)" ;
}


/*    OUTPUT: functmp6.cpp

        Result of (2 + 3i) * (2 + 3i) = (-5 + 12i)
        Float example:
        Result of 4.5 * 3 = 13.5

*/
```

Ex:

```
/*     FILE: functmp7.cpp     */

/*
   Overloaded print( ) method.
*/

// Errors: What's wrong?
// - value parameters that are numeric constants are not
//   ... alterable storage.
// - Compiler cannot determine how to insert our COMPLEX
//   ... into an output stream. A "specialized" template can be defined.
// - A non-template function can be defined, it supercedes the template
//   ... definition

#include <iostream>
using namespace std;

template <class t>
void print(const t &value);

struct COMPLEX{
   float Re;
   float Im;
};

template <> void print<COMPLEX>(const COMPLEX &c);
void print(const COMPLEX &c);

COMPLEX Mult(COMPLEX a, COMPLEX b);

int main( )
{
  COMPLEX c1, c2, cresult;
  float x, y, fresult;

  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…

```
  x = 4.5;
  y = 3;
  print("Float example:");
  cout << endl;
  cout << "Result of ";
  print(x);
  cout << " * ";
  print(y);
  cout << " = ";
  fresult = x * y;
  print(fresult);
  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;
}

template <class t>
void print(const t &value)
{
  cout << value;
}

template <> void print<COMPLEX>(const COMPLEX &c)
{
  cout << "(" << c.Re << " + " << c.Im << "i)" ;
}

void print(const COMPLEX &c)
{
  cout << "(((" << c.Re << " + " << c.Im << "i)))" ;
}


/*    OUTPUT: functmp7.cpp

       Result of (((2 + 3i))) * (((2 + 3i))) = (((-5 + 12i)))
       Float example:
       Result of 4.5 * 3 = 13.5

*/
```

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.

Ex:

```
/*     FILE: class1.cpp     */

/*
   This is an attempt at a data type that functions
   like a complex number,  a + bi, where a is the real
   part of the complex number and b is the real
   component of the imaginary part of the complex
   number.

   We can create COMPLEX entities, initialize them,
   print them and so far, even multiply them.
   However, they still don't "look" or act exactly
   like the C++ native data types. But they are
   working.
*/

#include <iostream>
using namespace std;

struct COMPLEX{
   double Re;
   double Im;
};

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

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

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

  cresult = Mult(c1, c2);

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

  return 0;
}

COMPLEX Mult(const COMPLEX  & a, const 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…

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

void init(COMPLEX &c, double r, double im)
{
  c.Re = r;
  c.Im = im;
}


/*    OUTPUT: class1.cpp

       Result of (2 + 3i) * (2 + 3i) = (-5 + 12i)

*/
```

Ex:

```
/*     FILE: class2.cpp     */

/*
   C++ allows us to place functions and function
   prototypes in our struct and class definitions
   so that the functionality described becomes
   "encapsulated" into the data type described.

   Note: These functions now "belong" to struct
         COMPLEX.
*/

#include <iostream>
using namespace std;

struct COMPLEX{
  double Re;
  double Im;
  COMPLEX Mult(const COMPLEX  & a, const COMPLEX & b);
  void print(const COMPLEX  & c);
  void init(COMPLEX &c, double r, double im);
};


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

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

  cresult = c1.Mult(c1, c2);

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

  return 0;
}

COMPLEX Mult(const COMPLEX  & a, const 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…

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

void init(COMPLEX &c, double r, double im)
{
  c.Re = r;
  c.Im = im;
}



/*    OUTPUT: class2.cpp

       c:\windows.o(.text+0x57):class2.cpp:
                      undefined reference to `COMPLEX::init(COMPLEX &, double, double)'
       c:\windows.o(.text+0x82):class2.cpp:
                      undefined reference to `COMPLEX::init(COMPLEX &, double, double)'
       c:\windows.o(.text+0x9a):class2.cpp:
                      undefined reference to `COMPLEX::Mult(COMPLEX const &, COMPLEX const &)'
       c:\windows.o(.text+0xda):class2.cpp:
                      undefined reference to `COMPLEX::print(COMPLEX const &)'
       c:\windows.o(.text+0x102):class2.cpp:
                      undefined reference to `COMPLEX::print(COMPLEX const &)'
       c:\windows.o(.text+0x12a):class2.cpp:
                      undefined reference to `COMPLEX::print(COMPLEX const &)'

*/
```

Ex:

```
/*     FILE: class3.cpp     */

/*
   C++ allows us to place functions and function
   prototypes in our struct and class definitions
   so that the functionality described becomes
   "encapsulated" into the data type described.

   These functions now "belong" to struct COMPLEX
   and are so indicated by the scope resolution
   operator, :: .
*/

#include <iostream>
using namespace std;

struct COMPLEX{
  double Re;
  double Im;
  COMPLEX Mult(const COMPLEX  & a, const COMPLEX & b);
  void print(const COMPLEX  & c);
  void init(COMPLEX &c, double r, double im);
};


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

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

  cresult = c1.Mult(c1, c2);

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

  return 0;
}

COMPLEX COMPLEX::Mult(const COMPLEX  & a, const 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…

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

void COMPLEX::init(COMPLEX &c, double r, double im)
{
  c.Re = r;
  c.Im = im;
}


/*    OUTPUT: class3.cpp

        Result of (2 + 3i) * (2 + 3i) = (-5 + 12i)

*/
```

Ex:

```
/*     FILE: class4.cpp     */

/*
   Once encapsulated, these functions can only be called
   using the member access operator by a variable or
   object of the defined type. So C++ automatically provides
   access to the variable/object running the method/function.

   This allows us to pass one less parameter in most cases.
*/

#include <iostream>
using namespace std;

struct COMPLEX{
  double Re;
  double Im;
  COMPLEX Mult(const COMPLEX & b);
  void print( );
  void init(double r, double im);
};

COMPLEX COMPLEX::Mult(const COMPLEX & b)
{
  COMPLEX result;

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

  return result;
}

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

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

cont…

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

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

  cresult = c1.Mult(c2);

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

  return 0;
}


/*    OUTPUT: class4.cpp

       Result of (2 + 3i) * (2 + 3i) = (-5 + 12i)

*/
```

Ex:

```
/*     FILE: class5.cpp     */

/*
   These programmer defined types can now explicitly
   be labeled a "class" with the C++ class keyword.
   The only difference between a class and a struct
   is that the information inside is restricted or
   "private" in a class but is "public" in a struct,
   by default.

   To make some class information in the objects
   less restricted we use an access specifier.  In
   this case the "public" access specifier.
*/

#include <iostream>
using namespace std;

class COMPLEX{
  double Re;
  double Im;
public:
  COMPLEX Mult(const COMPLEX & b);
  void print( );
  void init(double r, double im);
};

COMPLEX COMPLEX::Mult(const COMPLEX & b)
{
  COMPLEX result;

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

  return result;
}

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

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

cont…

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

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

  cresult = c1.Mult(c2);

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

  return 0;
}


/*    OUTPUT: class5.cpp

       Result of (2 + 3i) * (2 + 3i) = (-5 + 12i)

*/
```

Ex:

```
/*     FILE: class6.cpp     */

/*
   Allowing public access to all information allows
   little control over how an object is manipulated.

   More importantly it allows "consumers" to couple
   their code to the actual implementation.  This
   means that if the internal class structure or even
   something as simple as an attribute/variable name
   is changed, code that other programmers have written
   will now fail.
*/

#include <iostream>
using namespace std;

struct COMPLEX{
  double Re;
  double Im;
  COMPLEX Mult(const COMPLEX & b);
  void print( );
  void init(double r, double im);
};

COMPLEX COMPLEX::Mult(const COMPLEX & b)
{
  COMPLEX result;

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

  return result;
}

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

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

cont…

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

  c1.Re = 2;
  c1.Im = 3;
  c2.Re = 2;
  c2.Im = 3;

  cresult = c1.Mult(c2);

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

  return 0;
}


/*    OUTPUT: class6.cpp

       Result of (2 + 3i) * (2 + 3i) = (-5 + 12i)

*/
```

Ex:

```
/*     FILE: class7.cpp     */

/*
   Restricting access to some parts of a class or
   object keeps other programmers from coupling
   their code to the actual implementation.  This
   means that if the internal class structure or even
   something as simple as an attribute/variable name
   is changed, code that other programmers have written
   will NOT fail as long as the public access/interface
   is preserved.

   Most importantly, once the restricted information is
   indicated the compiler is aware of it and can enforce
   the restriction.
*/

#include <iostream>
using namespace std;

class COMPLEX{
  double Re;
  double Im;
public:
  COMPLEX Mult(const COMPLEX & b);
  void print( );
  void init(double r, double im);
};

COMPLEX COMPLEX::Mult(const COMPLEX & b)
{
  COMPLEX result;

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

  return result;
}

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

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

cont…

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

  c1.Re = 2;
  c1.Im = 3;
  c2.Re = 2;
  c2.Im = 3;

  cresult = c1.Mult(c2);

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

  return 0;
}


/*    OUTPUT: class7.cpp

        class7.cpp: In function `int main()':
        class7.cpp:20: `double COMPLEX::Re' is private
        class7.cpp:53: within this context
        class7.cpp:21: `double COMPLEX::Im' is private
        class7.cpp:54: within this context
        class7.cpp:20: `double COMPLEX::Re' is private
        class7.cpp:55: within this context
        class7.cpp:21: `double COMPLEX::Im' is private
        class7.cpp:56: within this context

*/
```

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.

Ex:

```
/*     FILE: const1.cpp     */

/*   The init( ) method has been turned into a constructor.  */

#include <iostream>
using namespace std;

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

  COMPLEX(double r, double im);
};

COMPLEX COMPLEX::Mult(const COMPLEX & b)
{
  COMPLEX result(0,0);

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

  return result;
}

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

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

int main( )
{
//      COMPLEX c1(2,3), c2(2,3), cresult;   //This would produce an error!
  COMPLEX c1(2,3), c2(2,3), cresult(0,0);

  cresult = c1.Mult(c2);

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

  return 0;
}
```

cont…

```
/*    OUTPUT: const1.cpp

       Result of (2 + 3i) * (2 + 3i) = (-5 + 12i)

*/
```

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.

Ex:

```
/*     FILE: const2.cpp     */

/*   A default constructor has been defined.   */

#include <iostream>
using namespace std;

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

  COMPLEX(double r, double im);
  COMPLEX( ){}
};

COMPLEX COMPLEX::Mult(const COMPLEX & b)
{
  COMPLEX result(0,0);

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

  return result;
}

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

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

int main( )
{
  COMPLEX c1(2,3), c2(2,3), cresult;

  cresult = c1.Mult(c2);

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

  return 0;
}
```

cont…

```
/*    OUTPUT: const2.cpp

       Result of (2 + 3i) * (2 + 3i) = (-5 + 12i)

*/
```

Ex:

```
/*     FILE: const2_1.cpp     */

/* A default constructor has been defined. Notice the definition, no
   initialization is actually done. That means, "No initialization is
   actually done!"
*/
#include <iostream>
using namespace std;

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

  COMPLEX(double r, double im);
  COMPLEX( ){}
};

COMPLEX COMPLEX::Mult(const COMPLEX & b)
{
  COMPLEX result(0,0);

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

  return result;
}

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

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

cont…

```
int main( )
{
  COMPLEX c1(2,3), c2(2,3), cresult;

  cout << "cresult = ";
  cresult.print( );
  cout << endl;
  cresult = c1.Mult(c2);

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

  return 0;
}

/*    OUTPUT: const2_1.cpp

       cresult = (1.78709e-307 + 1.90523e-307i)
       Result of (2 + 3i) * (2 + 3i) = (-5 + 12i)

*/
```

Ex:

```
/*     FILE: const3.cpp     */

/*
   The default constructor has been modified to initialize
   the object to 0,0.  For illustrative purposes the
   constructors print messages to show they are called.
*/

#include <iostream>
using namespace std;

class COMPLEX{
  double Re;
  double Im;
public:
  void print( );
  COMPLEX(double r, double im);
  COMPLEX( )
  {
    Re = Im = 0;
    cout << "Constructing-default a COMPLEX object" << endl;
  }
};

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

COMPLEX::COMPLEX(double r, double im)
{
  Re = r;
  Im = im;
  cout << "Constructing/Initializing a COMPLEX object" << endl;
}

int main( )
{
  COMPLEX c1(1,2);
  COMPLEX c2;
  COMPLEX carray[4];

  cout << "c1 = ";
  c1.print( );
  cout << endl;
  cout << "c2 = ";
  c2.print( );
  cout << endl;

  cout << "carray = " << endl;
  for(int i=0; i<4; i++){
    carray[i].print( );
    cout << endl;
  }

  return 0;
}
```

cont…

```
/*    OUTPUT: const3.cpp

       Constructing/Initializing a COMPLEX object
       Constructing-default a COMPLEX object
       Constructing-default a COMPLEX object
       Constructing-default a COMPLEX object
       Constructing-default a COMPLEX object
       Constructing-default a COMPLEX object
       c1 = (1 + 2i)
       c2 = (0 + 0i)
       carray =
       (0 + 0i)
       (0 + 0i)
       (0 + 0i)
       (0 + 0i)

*/
```

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.

Ex:

```
/*     FILE: const4.cpp     */

/*
   Access to the invoking object is provided implicitly
   thru a pointer named "this".

   "this" can be used explicitly.
*/

#include <iostream>
using namespace std;

class COMPLEX{
  double Re;
  double Im;
public:
  COMPLEX Mult(const COMPLEX & b);
  void print( );
  void print2( );

  COMPLEX(double r, double im);
  COMPLEX( ){}
};

COMPLEX COMPLEX::Mult(const COMPLEX & b)
{
  COMPLEX result(0,0);

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

  return result;
}

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

void COMPLEX::print2( )
{
  cout << "(" << this->Re << " + " << this->Im << "i)" ;
}

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

cont…

```
int main( )
{
  COMPLEX c1(2,3), c2(2,3), cresult;

  cresult = c1.Mult(c2);

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

  cout << "Result of ";
  c1.print2( );
  cout << " * ";
  c2.print2( );
  cout << " = ";
  cresult.print2( );
  cout << endl;

  return 0;
}


/*    OUTPUT: const4.cpp

       Result of (2 + 3i) * (2 + 3i) = (-5 + 12i)
       Result of (2 + 3i) * (2 + 3i) = (-5 + 12i)

*/
```

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.

Ex:

```
/*     FILE: Student.cpp     */

/*
   Static members are "class" variables, that is, they are
   independent of any particular objects.

   Static methods can be accessed using the scope resolution
   operator to identify the class and member being accessed.

*/
#include <iostream>
using namespace std;

class Student{
    char * name;
    double gpa;
    long id;
    static long count;  // Keeps count of Student objects
                        // ... and generates id numbers.
  public:
    static long getCount( );
    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;
}

Student::Student(char * n, double g)
{
  name = new char[strlen(n)+1];
  strcpy(name,n);
  gpa = g;
  id = ++count;
}

long Student::getCount( )
{
  return count;
}

char Student::getGrade( ) const
{
  char grade;

  if (gpa >= 3.5)
    grade = 'A';
  else if (gpa >= 2.5)
    grade = 'B';
  else if (gpa >= 1.5)
    grade = 'C';
  else if (gpa >= 0.5)
      grade = 'D';
  else
    grade = 'F';

  return grade;
}
```

cont…

```
void Student::display( ) const
{
  cout << "Student name = "  << name << endl;
  cout << "Student gpa = " << gpa << endl;
  cout << "Student id = " << id << endl;
}

void Student::print( ) const
{
  cout << name << ", " << gpa << ", " << id;
}

int main( )
{
    cout << "Number of students is: " << Student::getCount( ) << endl;

    Student s("Joe Cool", 3.95),
            s2("Cool Joe", 3.5),
            s3("Joe Llama", 3.2);

    s.print( );
    cout << " is a " << s.getGrade( ) << " student." << endl;
    s2.print( );
    cout << " is a " << s2.getGrade( ) << " student." << endl;
    s3.print( );
    cout << " is a " << s3.getGrade( ) << " student." << endl;
    cout << "Number of students is: " << Student::getCount( ) << endl;

    return 0;
}


/*    OUTPUT: Student.cpp

        Number of students is: 0
        Joe Cool, 3.95, 1 is a A student.
        Cool Joe, 3.5, 2 is a A student.
        Joe Llama, 3.2, 3 is a B student.
        Number of students is: 3

*/
```

Ex:

```
/*     FILE: Student2.cpp     */

/*
   Static members are "class" variables, that is, they are
   independent of any particular objects.

   Static methods can be accessed using an object of the
   class. This also identifies the class and member being
   accessed.

*/
#include <iostream>
using namespace std;

class Student{
    char * name;
    double gpa;
    long id;
    static long count;  // Keeps count of Student objects
                        // ... and generates id numbers.
  public:
    static long getCount( );
    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;
}

Student::Student(char * n, double g)
{
  name = new char[strlen(n)+1];
  strcpy(name,n);
  gpa = g;
  id = ++count;
}

long Student::getCount( )
{
  return count;
}

char Student::getGrade( ) const
{
  char grade;

  if (gpa >= 3.5)
    grade = 'A';
  else if (gpa >= 2.5)
    grade = 'B';
  else if (gpa >= 1.5)
    grade = 'C';
  else if (gpa >= 0.5)
      grade = 'D';
  else
    grade = 'F';

  return grade;
}
```

cont…

```
void Student::display( ) const
{
  cout << "Student name = "  << name << endl;
  cout << "Student gpa = " << gpa << endl;
  cout << "Student id = " << id << endl;
}

void Student::print( ) const
{
  cout << name << ", " << gpa << ", " << id;
}

int main( )
{
    cout << "Number of students is: " << Student::getCount( ) << endl;
                                        // No choice but to use class name,
                                        // ... no objects exist.
    Student s("Joe Cool", 3.95),
            s2("Cool Joe", 3.5),
            s3("Joe Llama", 3.2);

    s.print( );
    cout << " is a " << s.getGrade( ) << " student." << endl;
    s2.print( );
    cout << " is a " << s2.getGrade( ) << " student." << endl;
    s3.print( );
    cout << " is a " << s3.getGrade( ) << " student." << endl;
    cout << "Number of students is: " << s2.getCount( ) << endl;
                                        // s2 is used to call static method
    return 0;
}



/*    OUTPUT: Student2.cpp

        Number of students is: 0
        Joe Cool, 3.95, 1 is a A student.
        Cool Joe, 3.5, 2 is a A student.
        Joe Llama, 3.2, 3 is a B student.
        Number of students is: 3

*/
```

Ex:

```
/*     FILE: Student3.cpp     */

/*
   Static members are "class" variables, that is, they are
   independent of any particular objects.

   Static methods can be accessed using an object of the
   class. This also identifies the class and member being
   accessed.

*/
#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;
  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;
}

Student::Student(char * n, double g)
{
  name = new char[strlen(n)+1];
  strcpy(name,n);
  gpa = g;
  id = ++count;
}

long Student::getCount( )
{
  return count;
}

const char * Student::getName( ) const
{
  return name;
}

char * Student::setName(const char * const name)
{  // Exercising this
  strcpy(this->name,name);
  return this->name;
}
```

cont…

```
double Student::getGpa( ) const
{
  return gpa;
}

double Student::setGpa(double gpa)
{
  return this->gpa = gpa;
}

long Student::getId( ) const
{
  return id;
}

char Student::getGrade( ) const
{
  char grade;

  if (gpa >= 3.5)
    grade = 'A';
  else if (gpa >= 2.5)
    grade = 'B';
  else if (gpa >= 1.5)
    grade = 'C';
  else if (gpa >= 0.5)
     grade = 'D';
  else
    grade = 'F';

  return grade;
}

void Student::display( ) const
{
  cout << "Student name = "  << name << endl;
  cout << "Student gpa = " << gpa << endl;
  cout << "Student id = " << id << endl;
}

void Student::print( ) const
{
  cout << name << ", " << gpa << ", " << id;
}
```

cont…

```
int main( )
{
    cout << "Number of students is: " << Student::getCount( ) << endl;
                                        // No choice but to use class name,
                                        // ... no objects exist.
    Student s("Joe Cool", 3.95),
            s2("Cool Joe", 3.5),
            s3("Joe Llama", 3.2);

    s.print( );
    cout << " is a " << s.getGrade( ) << " student." << endl;
    s2.print( );
    cout << " is a " << s2.getGrade( ) << " student." << endl;
    s3.print( );
    cout << " is a " << s3.getGrade( ) << " student." << endl;
    cout << "Number of students is: " << s2.getCount( ) << endl;
                                        // s2 is used to call static method

    s.setName("Joe Coolest");
    s2.setName(s3.getName( ));
    s2.setGpa(2.25);
    s3.setName("Charlie Brown");
    s3.setGpa(3.25);

    cout << endl << endl;
    s.print( );
    cout << " is a " << s.getGrade( ) << " student." << endl;
    s2.print( );
    cout << " is a " << s2.getGrade( ) << " student." << endl;
    s3.print( );
    cout << " is a " << s3.getGrade( ) << " student." << endl;
    cout << "Number of students is: " << s2.getCount( ) << endl;
                                        // s2 is used to call static method
    return 0;
}


/*    OUTPUT: Student3.cpp

        Number of students is: 0
        Joe Cool, 3.95, 1 is a A student.
        Cool Joe, 3.5, 2 is a A student.
        Joe Llama, 3.2, 3 is a B student.
        Number of students is: 3


        Joe Coolest, 3.95, 1 is a A student.
        Joe Llama, 2.25, 2 is a C student.
        Charlie Brown, 3.25, 3 is a B student.
        Number of students is: 3

*/
```

Ex:

```
/*     FILE: Student4.cpp     */

/*
   Some errors associated with static members.

   Accessing a non-static method as if it were static.
   Not creating/initializing a static data member.
*/
#include <iostream>
using namespace std;

class Student{
    char * name;
    double gpa;
    long id;
    static long count;  // Keeps count of Student objects
                        // ... and generates id numbers.
  public:
    static long getCount( );
    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;
}

Student::Student(char * n, double g)
{
  name = new char[strlen(n)+1];
  strcpy(name,n);
  gpa = g;
  id = ++count;
}

long Student::getCount( )
{
  return count;
}

char Student::getGrade( ) const
{
  char grade;

  if (gpa >= 3.5)
    grade = 'A';
  else if (gpa >= 2.5)
    grade = 'B';
  else if (gpa >= 1.5)
    grade = 'C';
  else if (gpa >= 0.5)
      grade = 'D';
  else
    grade = 'F';

  return grade;
}
```

cont…

```
void Student::display( ) const
{
  cout << "Student name = "  << name << endl;
  cout << "Student gpa = " << gpa << endl;
  cout << "Student id = " << id << endl;
}

void Student::print( ) const
{
  cout << name << ", " << gpa << ", " << id;
}

int main( )
{
    cout << "Number of students is: " << Student::getCount( ) << endl;
                                        // No choice but to use class name,
                                        // ... no objects exist.
    Student s("Joe Cool", 3.95),
            s2("Cool Joe", 3.5),
            s3("Joe Llama", 3.2);

    Student::print( );
    cout << " is a " << s.getGrade( ) << " student." << endl;
    s2.print( );
    cout << " is a " << s2.getGrade( ) << " student." << endl;
    s3.print( );
    cout << " is a " << s3.getGrade( ) << " student." << endl;
    cout << "Number of students is: " << s2.getCount( ) << endl;
                                        // s2 is used to call static method
    return 0;
}



/*    OUTPUT: Student4.cpp

       Student4.cpp: In function `int main()':
       Student4.cpp:85: cannot call member function `Student::print() const' without object

*/
```

Ex:

```
/*     FILE: Student5.cpp     */

/*
   Some errors associated with static members.

   Not creating/initializing a static data member.
*/
#include <iostream>
using namespace std;

class Student{
    char * name;
    double gpa;
    long id;
    static long count;  // Keeps count of Student objects
                        // ... and generates id numbers.
  public:
    static long getCount( );
    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;
}

Student::Student(char * n, double g)
{
  name = new char[strlen(n)+1];
  strcpy(name,n);
  gpa = g;
  id = ++count;
}

long Student::getCount( )
{
  return count;
}

char Student::getGrade( ) const
{
  char grade;

  if (gpa >= 3.5)
    grade = 'A';
  else if (gpa >= 2.5)
    grade = 'B';
  else if (gpa >= 1.5)
    grade = 'C';
  else if (gpa >= 0.5)
      grade = 'D';
  else
    grade = 'F';

  return grade;
}
```

cont…

```
void Student::display( ) const
{
  cout << "Student name = "  << name << endl;
  cout << "Student gpa = " << gpa << endl;
  cout << "Student id = " << id << endl;
}

void Student::print( ) const
{
  cout << name << ", " << gpa << ", " << id;
}

int main( )
{
    cout << "Number of students is: " << Student::getCount( ) << endl;
                                        // No choice but to use class name,
                                        // ... no objects exist.
    Student s("Joe Cool", 3.95),
            s2("Cool Joe", 3.5),
            s3("Joe Llama", 3.2);

    s.print( );
    cout << " is a " << s.getGrade( ) << " student." << endl;
    s2.print( );
    cout << " is a " << s2.getGrade( ) << " student." << endl;
    s3.print( );
    cout << " is a " << s3.getGrade( ) << " student." << endl;
    cout << "Number of students is: " << s2.getCount( ) << endl;
                                        // s2 is used to call static method
    return 0;
}



/*    OUTPUT: Student5.cpp

       c:\windows.o(.text+0x8):Student5.cpp: undefined reference to `Student::count'
       c:\windows.o(.text+0xd):Student5.cpp: undefined reference to `Student::count'
       c:\windows.o(.text+0x69):Student5.cpp: undefined reference to `Student::count'
       c:\windows.o(.text+0x6e):Student5.cpp: undefined reference to `Student::count'
       c:\windows.o(.text+0x85):Student5.cpp: undefined reference to `Student::count'

*/
```

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…

```
Student::Student(char * n, double g)
{
  name = new char[strlen(n)+1];
  strcpy(name,n);
  gpa = g;
  setGrade( );
  id = ++count;
}

long Student::getCount( )
{
  return count;
}

const char * Student::getName( ) const
{
  return name;
}

char * Student::setName(const char * const name)
{  // Exercising this
  strcpy(this->name,name);
  return this->name;
}

double Student::getGpa( ) const
{
  return gpa;
}

double Student::setGpa(double gpa)
{
  this->gpa = gpa;
  setGrade( );
  return this->gpa;
}

long Student::getId( ) const
{
  return id;
}

char Student::getGrade( ) const
{
  return grade;
}

void Student::setGrade( )
{
  if (gpa >= 3.5)
    grade = 'A';
  else if (gpa >= 2.5)
    grade = 'B';
  else if (gpa >= 1.5)
    grade = 'C';
  else if (gpa >= 0.5)
      grade = 'D';
  else
    grade = 'F';
}

void Student::display( ) const
{
  cout << "Student name = "  << name << endl;
  cout << "Student gpa = " << gpa << endl;
  cout << "Student id = " << id << endl;
}
```

cont…

```
void Student::print( ) const
{
  cout << name << ", " << gpa << ", " << id;
}

int main( )
{
    cout << "Number of students is: " << Student::getCount( ) << endl;

    Student s("Joe Cool", 3.95),
            s2("Cool Joe", 3.5),
            s3("Joe Llama", 3.2);

    s.print( );
    cout << " is a " << s.getGrade( ) << " student." << endl;
    s2.print( );
    cout << " is a " << s2.getGrade( ) << " student." << endl;
    s3.print( );
    cout << " is a " << s3.getGrade( ) << " student." << endl;
    cout << "Number of students is: " << s2.getCount( ) << endl << endl;

    cout << s.getId( ) << " is a " << s.getGrade( ) << " student." << endl;
    cout << s2.getId( ) << " is a " << s2.getGrade( ) << " student." << endl;
    cout << s3.getId( ) << " is a " << s3.getGrade( ) << " student." << endl;

    return 0;
}



/*    OUTPUT: Student6.cpp

        Number of students is: 0
        Joe Cool, 3.95, 1 is a A student.
        Cool Joe, 3.5, 2 is a A student.
        Joe Llama, 3.2, 3 is a B student.
        Number of students is: 3

        1 is a A student.
        2 is a A student.
        3 is a B student.

*/
```

Ex:

```
/*     FILE: Student7.cpp     */

/*
   Changing the implementation of class Student more.

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

*/
#include <iostream>
#include <cstdio>
#include <cstdlib>
using namespace std;

class Student{
    static long count;  // Keeps count of Student objects
                        // ... and generates id numbers.
    char * name;
    double gpa;
    char id[10];
    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
{
  sprintf(id,"%09ld", ++count);
}

Student::Student(char * n, double g)
{
  name = new char[strlen(n)+1];
  strcpy(name,n);
  gpa = g;
  setGrade( );
  sprintf(id,"%09ld", ++count);
}

long Student::getCount( )
{
  return count;
}

const char * Student::getName( ) const
{
  return name;
}
```

cont…

```
char * Student::setName(const char * const name)
{  // Exercising this
  strcpy(this->name,name);
  return this->name;
}

double Student::getGpa( ) const
{
  return gpa;
}

double Student::setGpa(double gpa)
{
  this->gpa = gpa;
  setGrade( );
  return this->gpa;
}

long Student::getId( ) const
{
  return atol(id);
}

char Student::getGrade( ) const
{
  return grade;
}

void Student::setGrade( )
{
  if (gpa >= 3.5)
    grade = 'A';
  else if (gpa >= 2.5)
    grade = 'B';
  else if (gpa >= 1.5)
    grade = 'C';
  else if (gpa >= 0.5)
     grade = 'D';
  else
    grade = 'F';
}

void Student::display( ) const
{
  cout << "Student name = "  << name << endl;
  cout << "Student gpa = " << gpa << endl;
  cout << "Student id = " << id << endl;
}

void Student::print( ) const
{
  cout << name << ", " << gpa << ", " << id;
}
```

cont…

```
int main( )
{
    cout << "Number of students is: " << Student::getCount( ) << endl;

    Student s("Joe Cool", 3.95),
            s2("Cool Joe", 3.5),
            s3("Joe Llama", 3.2);

    s.print( );
    cout << " is a " << s.getGrade( ) << " student." << endl;
    s2.print( );
    cout << " is a " << s2.getGrade( ) << " student." << endl;
    s3.print( );
    cout << " is a " << s3.getGrade( ) << " student." << endl;
    cout << "Number of students is: " << s2.getCount( ) << endl << endl;

    cout << s.getId( ) << " is a " << s.getGrade( ) << " student." << endl;
    cout << s2.getId( ) << " is a " << s2.getGrade( ) << " student." << endl;
    cout << s3.getId( ) << " is a " << s3.getGrade( ) << " student." << endl;

    return 0;
}



/*    OUTPUT: Student7.cpp

        Number of students is: 0
        Joe Cool, 3.95, 000000001 is a A student.
        Cool Joe, 3.5, 000000002 is a A student.
        Joe Llama, 3.2, 000000003 is a B student.
        Number of students is: 3

        1 is a A student.
        2 is a A student.
        3 is a B student.

*/
```

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.

Ex:

```
/*     FILE: const7.cpp     */

/*
   The destructor has been added to show that it is called
   when an object is destroyed.
*/
#include <iostream>
using namespace std;

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

  COMPLEX(double r, double im);
  COMPLEX( )
  {
    Re = Im = 0;
  }
  ~COMPLEX( )
  {
   cout << "Destroying a COMPLEX object" << endl;
  }
};

COMPLEX COMPLEX::Mult(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)" ;
}

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

cont…

```
int main( )
{
  COMPLEX carray[4];

  for(int i=0; i<4; i++){
    cout << "carray[" << i << "] = ";
    carray[i].print( );
    cout << endl;
  }

  return 0;
}


/*    OUTPUT: const7.cpp

       carray[0] = (0 + 0i)
       carray[1] = (0 + 0i)
       carray[2] = (0 + 0i)
       carray[3] = (0 + 0i)
       Destroying a COMPLEX object
       Destroying a COMPLEX object
       Destroying a COMPLEX object
       Destroying a COMPLEX object

*/
```

Ex:

```
/*     FILE: const8.cpp     */

/*
   All constructors provided by the class display a
   message to show they are called, as does the destructor.

   Notice how often they are all called.
*/
#include <iostream>
using namespace std;

class COMPLEX{
  double Re;
  double Im;
public:
  void print( ) const;
  COMPLEX Mult(const COMPLEX b) const;

  COMPLEX(double r, double im);
  COMPLEX( )
  {
    Re = Im = 0;
    cout << "Constructing-default a COMPLEX object" << endl;
  }
  ~COMPLEX( )
  {
   cout << "Destroying a COMPLEX object" << endl;
  }
};

COMPLEX COMPLEX::Mult(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)" ;
}

COMPLEX::COMPLEX(double r, double im)
{
  Re = r;
  Im = im;
  cout << "Constructing a COMPLEX object" << endl;
}
```

cont…

```
// Initializing constructor gets called 2 times
// Default contructor gets called 2 times
// Destructor gets called ? times
int main( )
{
  COMPLEX c1(2,3), c2(2,3), cresult;

  cresult = c1.Mult(c2);

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

  return 0;
}


/*    OUTPUT: const8.cpp

        Constructing a COMPLEX object
        Constructing a COMPLEX object
        Constructing-default a COMPLEX object
        Constructing-default a COMPLEX object
        Destroying a COMPLEX object
        Destroying a COMPLEX object
        Result of (2 + 3i) * (2 + 3i) = (-5 + 12i)
        Destroying a COMPLEX object
        Destroying a COMPLEX object
        Destroying a COMPLEX object

*/
```

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.

Ex:

```
/*     FILE: const9.cpp     */

/*
   A new constructor called a "copy" constructor is provided.
   It makes duplicates of objects when they are needed for
   things like pass and return by value.

   Watch and see how many times the constructors and destructors
   are called.

   Note: Mult( ) has also been changed to have a value parameter.
*/
#include <iostream>
using namespace std;

class COMPLEX{
  double Re;
  double Im;
public:
  void print( ) const;
  COMPLEX Mult(const COMPLEX b) const;

  COMPLEX(double r, double im);
  COMPLEX(COMPLEX &c)
  {
    Re = c.Re;
    Im = c.Im;
    cout << "Constructing-COPY a COMPLEX object" << endl;
  }
  COMPLEX( )
  {
    Re = Im = 0;
    cout << "Constructing-DEFAULT a COMPLEX object" << endl;
  }
  ~COMPLEX( )
  {
   cout << "Destroying a COMPLEX object" << endl;
  }
};

COMPLEX COMPLEX::Mult(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…

```
COMPLEX::COMPLEX(double r, double im)
{
  Re = r;
  Im = im;
  cout << "Constructing a COMPLEX object" << endl;
}

// Initializing constructor gets called 2 times
// Default contructor gets called 2 times
// Copy contructor gets called 2 times
// Destructor gets called 6 times
int main( )
{
  COMPLEX c1(2,3), c2(2,3), cresult;

  cresult = c1.Mult(c2);

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

  return 0;
}


/*    OUTPUT: const9.cpp

        Constructing a COMPLEX object
        Constructing a COMPLEX object
        Constructing-DEFAULT a COMPLEX object
        Constructing-COPY a COMPLEX object
        Constructing-DEFAULT a COMPLEX object
        Constructing-COPY a COMPLEX object
        Destroying a COMPLEX object
        Destroying a COMPLEX object
        Destroying a COMPLEX object
        Result of (2 + 3i) * (2 + 3i) = (-5 + 12i)
        Destroying a COMPLEX object
        Destroying a COMPLEX object
        Destroying a COMPLEX object

*/
```

Ex:

```
/*     FILE: const10.cpp     */

/*
   The copy constructor should not alter the object being copied.
   Indicating the reference as const allows the compiler to
   enforce this.

   print( ) and mult( ) also should not modify their parameters.
*/
#include <iostream>
using namespace std;

class COMPLEX{
  double Re;
  double Im;
public:
  void print( ) const;
  COMPLEX Mult(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::Mult(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)" ;
}

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

cont…

```
int main( )
{
  COMPLEX c1(2,3), c2(2,3), cresult;

  cresult = c1.Mult(c2);

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

  return 0;
}


/*    OUTPUT: const10.cpp

       Result of (2 + 3i) * (2 + 3i) = (-5 + 12i)

*/
```

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…

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

int main( )
{
  COMPLEX c1(2,3), c2(2,3), cresult;

  //cresult = c1.Mult(c2);
  cresult = c1 * c2;

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

  return 0;
}


/*    OUTPUT: op_over1.cpp

       Result of (2 + 3i) * (2 + 3i) = (-5 + 12i)

*/
```

Ex:

```
/*     FILE: op_over1_1.cpp     */

/*
   The Mult( ) function can be turned into an operator definition.

   * for multiplication operator matches the operator*( )
   ... method/function call.
*/
#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)" ;
}

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

cont…

```
int main( )
{
  COMPLEX c1(2,3), c2(2,3), cresult;

  //cresult = c1.Mult(c2);
  cresult = c1.operator*(c2);

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

  return 0;
}


/*    OUTPUT: op_over1_1.cpp

        Result of (2 + 3i) * (2 + 3i) = (-5 + 12i)

*/
```

Ex:

```
/*     FILE: op_over2.cpp     */

/*   More operators, one is unary.  */
#include <iostream>
using namespace std;

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

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;
}

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

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

  return result;
}
```

cont…

```
COMPLEX COMPLEX::operator-( ) const
{
  COMPLEX result;

  result.Re = -Re;
  result.Im = -Im;

  return result;
}

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

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

int main( )
{
  COMPLEX c1(2,3), c2(2,3), cresult;

   cresult = c1 * c2;   // multiply operator

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

   cresult = c1 - c2;   // subtraction operator

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

   cresult = -c1;       // negation operator

   cout << "- ";
   c1.print( );
   cout << " = ";
   cresult.print( );
   cout << endl;

  return 0;
}
```

cont…

```
/*    OUTPUT: op_over2.cpp

       Result of (2 + 3i) * (2 + 3i) = (-5 + 12i)
       Result of (2 + 3i) - (2 + 3i) = (0 + 0i)
       - (2 + 3i) = (-2 + -3i)

*/
```

Ex:

```
/*     FILE: op_over3.cpp     */

/*
   Multiply a COMPLEX times a double.
*/
#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;

  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( )
  {
  }
};

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;
}

COMPLEX COMPLEX::operator*(const double &x) const
{
  COMPLEX result;

  result.Re = Re*x;
  result.Im = Im*x;

  return result;
}
```

cont…

```
COMPLEX COMPLEX::operator-(const COMPLEX &b) const
{
  COMPLEX result;

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

  return result;
}

COMPLEX COMPLEX::operator-( ) const
{
  COMPLEX result;

  result.Re = -Re;
  result.Im = -Im;

  return result;
}

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

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

int main( )
{
  COMPLEX c1(2,3), c2(2,3), cresult;

   cresult = c1 * c2;   // multiply operator COMPLEX * COMPLEX

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

   cresult = c1 * 3;    // multiply operator COMPLEX * double

   cout << "Result of ";
   c1.print( );
   cout << " * 3 = ";
   cresult.print( );
   cout << endl;

  return 0;
}
```

cont…

```
/*    OUTPUT: op_over3.cpp

       Result of (2 + 3i) * (2 + 3i) = (-5 + 12i)
       Result of (2 + 3i) * 3 = (6 + 9i)

*/
```

Ex:

```
/*     FILE: op_over4.cpp     */

/*
   Multiply a COMPLEX times a double.  Won't work for double
   ...times COMPLEX.
*/
#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;

  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( )
  {
  }
};

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;
}

COMPLEX COMPLEX::operator*(const double &x) const
{
  COMPLEX result;

  result.Re = Re*x;
  result.Im = Im*x;

  return result;
}
```

cont…

```
COMPLEX COMPLEX::operator-(const COMPLEX &b) const
{
  COMPLEX result;

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

  return result;
}

COMPLEX COMPLEX::operator-( ) const
{
  COMPLEX result;

  result.Re = -Re;
  result.Im = -Im;

  return result;
}

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

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

int main( )
{
  COMPLEX c1(2,3), c2(2,3), cresult;

   cresult = c1 * c2;   // multiply operator COMPLEX * COMPLEX

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

   cresult = 3.0 * c1;    // multiply operator double * COMPLEX

   cout << "Result of ";
   cout << " 3.0 * ";
   c1.print( );
   cout << " = ";
   cresult.print( );

  return 0;
}
```

cont…

```
/*    OUTPUT: op_over4.cpp

       op_over4.cpp: In function `int main()':
       op_over4.cpp:99: no match for `double * COMPLEX &'

*/
```

Ex:

```
/*     FILE: op_over5.cpp     */

/*
   Definition of double times COMPLEX.
*/
#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;

  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( )
  {
  }
};

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;
}

COMPLEX COMPLEX::operator*(const double &x) const
{
  COMPLEX result;

  result.Re = Re*x;
  result.Im = Im*x;

  return result;
}
```

cont…

```
COMPLEX operator*(const double &x, const COMPLEX &c)
{
  COMPLEX result;

  result.Re = c.Re*x;
  result.Im = c.Im*x;

  return result;
}

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

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

  return result;
}

COMPLEX COMPLEX::operator-( ) const
{
  COMPLEX result;

  result.Re = -Re;
  result.Im = -Im;

  return result;
}

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

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

cont…

```
int main( )
{
  COMPLEX c1(2,3), c2(2,3), cresult;

   cresult = c1 * c2;   // multiply operator COMPLEX * COMPLEX

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

   cresult = 3.0 * c1;    // multiply operator double * COMPLEX

   cout << "Result of ";
   cout << " 3.0 * ";
   c1.print( );
   cout << " = ";
   cresult.print( );

  return 0;
}


/*    OUTPUT: op_over5.cpp

        op_over5.cpp:
                  In function `class COMPLEX operator *(const double &, const COMPLEX &)':
        op_over5.cpp:8: `double COMPLEX::Re' is private
        op_over5.cpp:57: within this context
        op_over5.cpp:8: `double COMPLEX::Re' is private
        op_over5.cpp:57: within this context
        op_over5.cpp:9: `double COMPLEX::Im' is private
        op_over5.cpp:58: within this context
        op_over5.cpp:9: `double COMPLEX::Im' is private
        op_over5.cpp:58: within this context

*/
```

Ex:

```
/*     FILE: op_over6.cpp     */

/*
   Test of whether this double times COMPLEX function is
   ... used by the compiler.
*/
#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;

  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( )
  {
  }
};

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;
}

COMPLEX COMPLEX::operator*(const double &x) const
{
  COMPLEX result;

  result.Re = Re*x;
  result.Im = Im*x;

  return result;
}
```

cont…

```
COMPLEX operator*(const double &x, const COMPLEX &c)
{
  COMPLEX result(1.0,1.0);

//  result.Re = c.Re*x;
//  result.Im = c.Im*x;

  return result;
}

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

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

  return result;
}

COMPLEX COMPLEX::operator-( ) const
{
  COMPLEX result;

  result.Re = -Re;
  result.Im = -Im;

  return result;
}

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

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

cont…

```
int main( )
{
  COMPLEX c1(2,3), c2(2,3), cresult;

   cresult = c1 * c2;   // multiply operator COMPLEX * COMPLEX

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

   cresult = 3.0 * c1;    // multiply operator double * COMPLEX

   cout << "Result of ";
   cout << " 3.0 * ";
   c1.print( );
   cout << " = ";
   cresult.print( );
   cout << endl;

  return 0;
}


/*    OUTPUT: op_over6.cpp

        Result of (2 + 3i) * (2 + 3i) = (-5 + 12i)
        Result of  3.0 * (2 + 3i) = (1 + 1i)

*/
```

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…

```
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;
}

COMPLEX COMPLEX::operator*(const double &x) const
{
  COMPLEX result;

  result.Re = Re*x;
  result.Im = Im*x;

  return result;
}

COMPLEX operator*(const double &x, const COMPLEX &c)
{
  COMPLEX result;

  result.Re = c.Re*x;
  result.Im = c.Im*x;

  return result;
}

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

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

  return result;
}

COMPLEX COMPLEX::operator-( ) const
{
  COMPLEX result;

  result.Re = -Re;
  result.Im = -Im;

  return result;
}

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

cont…

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

int main( )
{
  COMPLEX c1(2,3), c2(2,3), cresult;

   cresult = c1 * c2;   // multiply operator COMPLEX * COMPLEX

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

   cresult = 3.0 * c1;    // multiply operator double * COMPLEX

   cout << "Result of ";
   cout << " 3.0 * ";
   c1.print( );
   cout << " = ";
   cresult.print( );
   cout << endl;

  return 0;
}


/*    OUTPUT: op_over7.cpp

       Result of (2 + 3i) * (2 + 3i) = (-5 + 12i)
       Result of  3.0 * (2 + 3i) = (6 + 9i)

*/
```

Ex:

```
/*     FILE: op_over8.cpp     */

/*
   Definition of double times COMPLEX.

   Code reuse version.
*/
#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( )
  {
  }
};

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;
}

COMPLEX COMPLEX::operator*(const double &x) const
{
  COMPLEX result;

  result.Re = Re*x;
  result.Im = Im*x;

  return result;
}
```

cont…

```
COMPLEX operator*(const double &x, const COMPLEX &c)
{
  return (c * x);
}

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

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

  return result;
}

COMPLEX COMPLEX::operator-( ) const
{
  COMPLEX result;

  result.Re = -Re;
  result.Im = -Im;

  return result;
}

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

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

cont…

```
int main( )
{
  COMPLEX c1(2,3), c2(2,3), cresult;

   cresult = c1 * c2;   // multiply operator COMPLEX * COMPLEX

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

   cresult = 3.0 * c1;    // multiply operator COMPLEX * double

   cout << "Result of ";
   cout << " 3.0 * ";
   c1.print( );
   cout << " = ";
   cresult.print( );
   cout << endl;

  return 0;
}


/*    OUTPUT: op_over8.cpp

        Result of (2 + 3i) * (2 + 3i) = (-5 + 12i)
        Result of  3.0 * (2 + 3i) = (6 + 9i)

*/
```

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…

```
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;
}

COMPLEX COMPLEX::operator*(const double &x) const
{
  COMPLEX result;

  result.Re = Re*x;
  result.Im = Im*x;

  return result;
}

COMPLEX operator*(const double &x, const COMPLEX &c)
{
  return (c * x);
}

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

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

  return result;
}

COMPLEX COMPLEX::operator-( ) const
{
  COMPLEX result;

  result.Re = -Re;
  result.Im = -Im;

  return result;
}

ostream & operator<<(ostream& os, const COMPLEX & c)
{
  os << "(" << c.Re << " + " << c.Im << "i)" ;
  return os;
}
```

cont…

```
int main( )
{
  COMPLEX c1(2,3), c2(2,3), cresult;

   cresult = c1 * c2;   // multiply operator COMPLEX * COMPLEX

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

   cresult = 3.0 * c1;    // multiply operator COMPLEX * double

   cout << "Result of "
        << " 3.0 * "
        << c1
        << " = "
        << cresult
        << endl;

  return 0;
}


/*    OUTPUT: put_to.cpp

       Result of (2 + 3i) * (2 + 3i) = (-5 + 12i)
       Result of  3.0 * (2 + 3i) = (6 + 9i)

*/
```

Ex:

```
/*     FILE: put_to2.cpp     */

/*
   Overloading the put-to operator << for class COMPLEX.

   A near miss.
*/
#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 void operator<<(ostream&,const COMPLEX &);
};

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

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;
}
```

cont…

```
COMPLEX COMPLEX::operator*(const double &x) const
{
  COMPLEX result;

  result.Re = Re*x;
  result.Im = Im*x;

  return result;
}

COMPLEX operator*(const double &x, const COMPLEX &c)
{
  return (c * x);
}

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

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

  return result;
}

COMPLEX COMPLEX::operator-( ) const
{
  COMPLEX result;

  result.Re = -Re;
  result.Im = -Im;

  return result;
}

void operator<<(ostream& os, const COMPLEX & c)
{
  os << "(" << c.Re << " + " << c.Im << "i)" ;
}
```

cont…

```
int main( )
{
  COMPLEX c1(2,3), c2(2,3), cresult;

   cresult = c1 * c2;   // multiply operator COMPLEX * COMPLEX

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

   cresult = 3.0 * c1;    // multiply operator COMPLEX * double

   cout << "Result of "
        << " 3.0 * "
        << c1
        << " = "
        << cresult
        << endl;

  return 0;
}


/*    OUTPUT: put_to2.cpp

        put_to2.cpp: In function `int main()':
        put_to2.cpp:100: void value not ignored as it ought to be
        put_to2.cpp:109: void value not ignored as it ought to be

*/
```

Ex:

```
/*     FILE: put_to3.cpp     */

/*
   Overloading the put-to operator << for class COMPLEX.

   A near miss. It could work in some situations but it
   ...isn't like the standard C++ types.
*/
#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 void operator<<(ostream&,const COMPLEX &);
};

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

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;
}
```

cont…

```
COMPLEX COMPLEX::operator*(const double &x) const
{
  COMPLEX result;

  result.Re = Re*x;
  result.Im = Im*x;

  return result;
}

COMPLEX operator*(const double &x, const COMPLEX &c)
{
  return (c * x);
}

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

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

  return result;
}

COMPLEX COMPLEX::operator-( ) const
{
  COMPLEX result;

  result.Re = -Re;
  result.Im = -Im;

  return result;
}

void operator<<(ostream& os, const COMPLEX & c)
{
  os << "(" << c.Re << " + " << c.Im << "i)" ;
}
```

cont…

```
int main( )
{
  COMPLEX c1(2,3), c2(2,3), cresult;

   cresult = c1 * c2;   // multiply operator COMPLEX * COMPLEX

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

   cresult = 3.0 * c1;    // multiply operator COMPLEX * double

   cout << "Result of "
        << " 3.0 * "
        << c1;
   cout << " = "
        << cresult;
   cout << endl;

  return 0;
}


/*    OUTPUT: put_to3.cpp

        Result of (2 + 3i) * (2 + 3i) = (-5 + 12i)
        Result of  3.0 * (2 + 3i) = (6 + 9i)

*/
```

Ex:

```
/*     FILE: put_to4.cpp     */

/*
   Overloading the put-to operator << for class COMPLEX.

   Another near miss. To cascade put-to operations
   ... the put-to operator must produce a reference
   ... to the same stream, not a copy of the stream.

   Class ostream does not allow a copy of an ostream to
   ...be produced.
*/
#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;
}

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;
}
```

cont…

```
COMPLEX COMPLEX::operator*(const double &x) const
{
  COMPLEX result;

  result.Re = Re*x;
  result.Im = Im*x;

  return result;
}

COMPLEX operator*(const double &x, const COMPLEX &c)
{
  return (c * x);
}

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

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

  return result;
}

COMPLEX COMPLEX::operator-( ) const
{
  COMPLEX result;

  result.Re = -Re;
  result.Im = -Im;

  return result;
}

ostream operator<<(ostream& os, const COMPLEX & c)
{
  os << "(" << c.Re << " + " << c.Im << "i)" ;
  return os;
}
```

cont…

```
int main( )
{
  COMPLEX c1(2,3), c2(2,3), cresult;

  cresult = c1 * c2;   // multiply operator COMPLEX * COMPLEX

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

  cresult = 3.0 * c1;    // multiply operator COMPLEX * double

  cout << "Result of "
       << " 3.0 * "
       << c1
       << " = "
       << cresult
       << endl;

  return 0;
}


/*    OUTPUT: put_to4.cpp

        put_to4.cpp: In method `ostream::ostream(const ostream &)':
        D:\LANGC++++.h:128: `ios::ios(const ios &)' is private
        put_to4.cpp:96: within this context

*/
```

Ex:

```
/*     FILE: put_to5.cpp     */

/*
   Overloading the put-to operator << for class COMPLEX.

   Another near miss. cout isn't the only ostream destination.

   This version only writes output to cout.
*/
#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;
}

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;
}
```

cont…

```
COMPLEX COMPLEX::operator*(const double &x) const
{
  COMPLEX result;

  result.Re = Re*x;
  result.Im = Im*x;

  return result;
}

COMPLEX operator*(const double &x, const COMPLEX &c)
{
  return (c * x);
}

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

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

  return result;
}

COMPLEX COMPLEX::operator-( ) const
{
  COMPLEX result;

  result.Re = -Re;
  result.Im = -Im;

  return result;
}

ostream& operator<<(ostream& os, const COMPLEX & c)
{
  cout << "(" << c.Re << " + " << c.Im << "i)" ;
  return cout;
}
```

cont…

```
int main( )
{
  COMPLEX c1(2,3), c2(2,3), cresult;

  cresult = c1 * c2;   // multiply operator COMPLEX * COMPLEX

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

  cresult = 3.0 * c1;    // multiply operator COMPLEX * double

  cout << "Result of "
       << " 3.0 * "
       << c1
       << " = "
       << cresult
       << endl;

  return 0;
}


/*    OUTPUT: put_to5.cpp

        Result of (2 + 3i) * (2 + 3i) = (-5 + 12i)
        Result of  3.0 * (2 + 3i) = (6 + 9i)

*/
```

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!

*/
```

Ex:

```
/*     FILE: File2.cpp     */

// Write some objects to a file using operator<<

#include <fstream>

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

class MyDouble{
    double d;
  public:
    MyDouble(double param)
    { d = param;}
    double getDouble( )
    { return d;}
    bool operator<(const MyDouble & param) const
    { return d < param.d; }

    friend ostream& operator<<(ostream &os, const MyDouble & d);
};

ostream& operator<<(ostream &os, const MyDouble & d)
{
  os << d.d ;

  return os;
}

int main( )
{
  ofstream outs;
  MyDouble d1(5.2), d2(3.2);

  outs.open("File2.out");

  outs << "MyDouble d1 = " << d1 << endl;
  outs << "MyDouble d2 = " << d2 << endl;
  outs << d1 << " < " << d2 << " = " << (d1 < d2) << endl;
  outs << d2 << " < " << d1 << " = " << (d2 < d1) << endl;

  outs.close( );
}


/*   OUTPUT: File2.OUT

       MyDouble d1 = 5.2
       MyDouble d2 = 3.2
       5.2 < 3.2 = 0
       3.2 < 5.2 = 1

*/
```

Ex:

```
/*     FILE: File3.cpp     */

// Write some objects to a file using operator<<
// ... control the sign on the COMPLEX objects
#include <fstream>

using std::ofstream;
using std::ostream;
using std::endl;
using std::ios;


class COMPLEX{
  float Re;
  float Im;
public:
  COMPLEX(float r, float im);
  COMPLEX( ){Re = 0; Im = 0;}
  COMPLEX operator*(const COMPLEX &b) const;
  COMPLEX operator+(const COMPLEX &b) const;
  int operator<(const COMPLEX &b) const;
  friend ostream& operator<<(ostream&,const COMPLEX &);
};

int COMPLEX::operator<(const COMPLEX &b) const
{
  float mag1, mag2;

  mag1 = Re*Re + Im*Im;
  mag2 = b.Re*b.Re + b.Im*b.Im;

  return mag1<mag2;
}

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

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

  return result;
}

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;
}

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

ostream & operator<<(ostream & os, const COMPLEX & c)
{
   os << c.Re ;
   os.setf(ios::showpos);
   os << c.Im << "i";
   os.unsetf(ios::showpos);
   return os;
}
```

cont…

```
int main( )
{
  ofstream outs;
  COMPLEX c(3.4,5.2);
  COMPLEX c2(-3.4,-5.2);

  outs.open("File3.out");

  outs << "COMPLEX  c =  " << c << endl;
  outs << "COMPLEX c2 = " << c2 << endl;
  outs << "COMPLEX  c =  " << c << endl;

  outs.close( );
}


/*   OUTPUT: File3.OUT

       COMPLEX  c =  3.4+5.2i
       COMPLEX c2 = -3.4-5.2i
       COMPLEX  c =  3.4+5.2i

*/
```

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.

Ex:

```
/*     FILE: ShapesTest.cpp     */

/*  Classes triangle and circle are using existing class point.

    triangle "has-a" set of points that are vertices.
    circle "is-a" point with a radius.

*/
#include <iostream>
using namespace std;

class point{
    int x,y;

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

    void setpoint(int new_x, int new_y);
    void setX(int new_x);
    void setY(int new_y);
    int getX( ) const;
    int getY( ) const;

    friend ostream & operator<<(ostream & os, const point& p);
};

point::point(int xvalue, int yvalue)
{
  setpoint(xvalue,yvalue);
}

void point::setX(int new_x)
{
        x = new_x;
}

void point::setY(int new_y)
{
  y = new_y;
}

int point::getX( ) const
{
  return x;
}

int point::getY( ) const
{
  return y;
}

void point::setpoint(int new_x, int new_y)
{
  x = new_x;
  y = new_y;
}

ostream & operator<<(ostream & os, const point& p)
{
  os << "(" << p.x << "," << p.y << ")";
  return os;
}
```

cont…

```
class circle:public point{
    double radius;

  public:
    circle( ) {}
    circle(int x, int y, double radius);
    circle(point center, double radius);

    void setCenter(point center);
    point getCenter( ) const;

    void scale(double factor);

    void setRadius(double new_r);
    double getRadius( ) const;

    friend ostream & operator<<(ostream & os, const circle& c);
};

circle::circle(int x, int y, double radius):point(x,y)
{
  setRadius(radius);
}

circle::circle(point center, double radius)
{
  setX(center.getX( ));
  setY(center.getY( ));
  setRadius(radius);
}

void circle::setCenter(point center)
{
  setX(center.getX( ));
  setY(center.getY( ));
}

point circle::getCenter( ) const
{
  return point(getX( ), getY( ));
}

void circle::scale(double factor)
{
  radius = fabs(factor)*radius;
}

void circle::setRadius(double new_r)
{
  radius = new_r;
}

double circle::getRadius( ) const
{
  double radius;
}

ostream & operator<<(ostream & os, const circle& c)
{
  os << "C" << (point)c << "  r = " << c.radius;
  return os;
}
```

cont…

```
class triangle{
    point v1;
    point v2;
    point v3;
  public:
    triangle( )
    {}
    triangle( point p1, point p2, point p3);
    triangle(int px1, int py1, int px2, int py2, int px3, int py3);

    void setTriangle( int px1, int py1, int px2, int py2, int px3, int py3);
    point getVertex1( ) const;
    point getVertex2( ) const;
    point getVertex3( ) const;
    void setVertex1(point p1);
    void setVertex2(point p2);
    void setVertex3(point p3);

    friend ostream & operator<<(ostream & os, const triangle& t);
};

triangle::triangle( point p1, point p2, point p3)
{
  setTriangle(p1.getX( ), p1.getY( ), p2.getX( ), p2.getY( ), p3.getX( ), p3.getY( ));
}

triangle::triangle(int px1, int py1, int px2, int py2, int px3, int py3)
{
  setTriangle(px1, py1, px2, py2, px3, py3);
}

void triangle::setTriangle( int px1, int py1, int px2, int py2, int px3, int py3)
{
  v1 = point(px1, py1);
  v2 = point(px2, py2);
  v3 = point(px3, py3);
}

point triangle::getVertex1( ) const
{
  return v1;
}


point triangle::getVertex2( ) const
{
  return v2;
}


point triangle::getVertex3( ) const
{
  return v3;
}

void triangle::setVertex1(point p1)
{
  v1 = p1;
}

void triangle::setVertex2(point p2)
{
  v2 = p2;
}

void triangle::setVertex3(point p3)
{
  v3 = p3;
}
```

cont…

```
ostream & operator<<(ostream & os, const triangle& t)
{
  os << t.v1 << " " << t.v2 << " " << t.v3;
  return os;
}

int main( )
{
    point p(7,5);          // Build some shapes
    triangle t(1,2,3,4,1,6);
    circle c(1,2,3.5);

    cout << c << endl; // Display the shapes
    cout << t << endl;
    cout << p << endl;

    c = circle(p,1.5);

    cout << c << endl;
}


/*    OUTPUT: ShapesTest.cpp

       C(1,2)  r = 3.5
       (1,2) (3,4) (1,6)
       (7,5)
       C(7,5)  r = 1.5

*/
```

Ex:

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

#ifndef _point_h
#define _point_h

#include <iostream>

using namespace std;

class point{
    int x,y;

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

    void setpoint(int new_x, int new_y);
    void setX(int new_x);
    void setY(int new_y);
    int getX( ) const;
    int getY( ) const;

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

```
/*     FILE: point.cpp     */

#include "point.h"

point::point(int xvalue, int yvalue)
{
  setpoint(xvalue,yvalue);
}

void point::setX(int new_x)
{
  x = new_x;
}

void point::setY(int new_y)
{
  y = new_y;
}

int point::getX( ) const
{
  return x;
}

int point::getY( ) const
{
  return y;
}

void point::setpoint(int new_x, int new_y)
{
  x = new_x;
  y = new_y;
}

ostream & operator<<(ostream & os, const point& p)
{
  os << "(" << p.x << "," << p.y << ")";
  return os;
}
```

cont…

```
/*     FILE: circle.h     */

#ifndef _circle_h
#define _circle_h

#include <iostream>
#include "point.h"

using namespace std;

class circle:public point{
    double radius;

  public:
    circle( ) {}
    circle(int x, int y, double radius);
    circle(point center, double radius);

    void setCenter(point center);
    point getCenter( ) const;

    void scale(double factor);

    void setRadius(double new_r);
    double getRadius( ) const;

    friend ostream & operator<<(ostream & os, const circle& c);
};

#endif
```

cont…

```
/*     FILE: circle.cpp     */

#include "circle.h"

circle::circle(int x, int y, double radius):point(x,y)
{
  setRadius(radius);
}

circle::circle(point center, double radius)
{
  setCenter(center);
  setRadius(radius);
}

void circle::setCenter(point center)
{
  setX(center.getX( ));
  setY(center.getY( ));
}

point circle::getCenter( ) const
{
  return point(getX( ), getY( ));
}

void circle::scale(double factor)
{
  radius = fabs(factor)*radius;
}

void circle::setRadius(double new_r)
{
  radius = new_r;
}

double circle::getRadius( ) const
{
  double radius;
}

ostream & operator<<(ostream & os, const circle& c)
{
  os << "C" << (point)c << "  r = " << c.radius;
  return os;
}
```

cont…

```
/*     FILE: triangle.h     */

#ifndef _triangle_h
#define _triangle_h

#include <iostream>
#include "point.h"

using namespace std;

class triangle{
    point v1;
    point v2;
    point v3;
  public:
    triangle( )
    {}
    triangle( point p1, point p2, point p3);
    triangle(int px1, int py1, int px2, int py2, int px3, int py3);

    void setTriangle( int px1, int py1, int px2, int py2, int px3, int py3);
    point getVertex1( ) const;
    point getVertex2( ) const;
    point getVertex3( ) const;
    void setVertex1(point p1);
    void setVertex2(point p2);
    void setVertex3(point p3);

    friend ostream & operator<<(ostream & os, const triangle& t);
};

#endif
```

cont…

```
/*     FILE: triangle.cpp     */

#include "triangle.h"

triangle::triangle( point p1, point p2, point p3)
{
  setTriangle(p1.getX( ), p1.getY( ), p2.getX( ), p2.getY( ), p3.getX( ), p3.getY( ));
}

triangle::triangle(int px1, int py1, int px2, int py2, int px3, int py3)
{
  setTriangle(px1, py1, px2, py2, px3, py3);
}

void triangle::setTriangle( int px1, int py1, int px2, int py2, int px3, int py3)
{
  v1 = point(px1, py1);
  v2 = point(px2, py2);
  v3 = point(px3, py3);
}

point triangle::getVertex1( ) const
{
  return v1;
}

point triangle::getVertex2( ) const
{
  return v2;
}

point triangle::getVertex3( ) const
{
  return v3;
}

void triangle::setVertex1(point p1)
{
  v1 = p1;
}

void triangle::setVertex2(point p2)
{
  v2 = p2;
}

void triangle::setVertex3(point p3)
{
  v3 = p3;
}

ostream & operator<<(ostream & os, const triangle& t)
{
  os << t.v1 << " " << t.v2 << " " << t.v3;
  return os;
}
```

cont…

```
/*     FILE:  ShapesTest2.cpp     */

/*  Classes triangle and circle are using existing class point.

    triangle "has-a" set of points that are vertices.
    circle "is-a" point with a radius.

*/
#include <iostream>
using namespace std;

#include "point.h"
#include "circle.h"
#include "triangle.h"

int main( )
{
  point p(7,5);          // Build some shapes
  triangle t(1,2,3,4,1,6);
  circle c(1,2,3.5);

  cout << c << endl; // Display the shapes
  cout << t << endl;
  cout << p << endl;

  c = circle(p,1.5);

  cout << c << endl;
}


/*    OUTPUT: ShapesTest2.cpp

       C(1,2)  r = 3.5
       (1,2) (3,4) (1,6)
       (7,5)
       C(7,5)  r = 1.5

*/
```

Ex:

```
/*     FILE: ./shapes2/point.h     */

#ifndef _point_h
#define _point_h

#include <iostream>

using namespace std;

class point{
    int x,y;

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

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

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

cont…

```
/*     FILE: ./shapes2/point.cpp     */

#include "point.h"

point::point(int xvalue, int yvalue)
{
  setPoint(xvalue,yvalue);
}

void point::setX(int new_x)
{
  x = new_x;
}

void point::setY(int new_y)
{
  y = new_y;
}

int point::getX( ) const
{
  return x;
}

int point::getY( ) const
{
  return y;
}

void point::setPoint(int new_x, int new_y)
{
  x = new_x;
  y = new_y;
}

ostream & operator<<(ostream & os, const point& p)
{
  os << "(" << p.x << "," << p.y << ")";
  return os;
}
```

cont…

```
/*     FILE: ./shapes2/circle.h     */

#ifndef _circle_h
#define _circle_h

#include <iostream>
#include "point.h"

using namespace std;

class circle:public point{
    double radius;

  public:
    circle( ) {}
    circle(int x, int y, double radius);
    circle(point center, double radius);

    void setCenter(int x, int y);
    void setCenter(point center);
    point getCenter( ) const;

    void scale(double factor);

    void setRadius(double new_r);
    double getRadius( ) const;

    friend ostream & operator<<(ostream & os, const circle& c);
};

#endif
```

cont…

```
/*     FILE: ./shapes2/circle.cpp     */

#include "circle.h"

circle::circle(int x, int y, double radius):point(x,y)
{
  setRadius(radius);
}

circle::circle(point center, double radius):point(center)
{
  setRadius(radius);
}

void circle::setCenter(int x, int y)
{
  setX(x);
  setY(y);
}

void circle::setCenter(point center)
{
  *(point *)this = center;
}

point circle::getCenter( ) const
{
  return *this;
}

void circle::scale(double factor)
{
  radius = fabs(factor)*radius;
}

void circle::setRadius(double new_r)
{
  radius = new_r;
}

double circle::getRadius( ) const
{
  return radius;
}

ostream & operator<<(ostream & os, const circle& c)
{
  os << "C" << (point)c << "  r = " << c.radius;
  return os;
}
```

cont…

```
/*     FILE: ./shapes2/triangle.h     */

#ifndef _triangle_h
#define _triangle_h

#include <iostream>
#include "point.h"

using namespace std;

class triangle{
    point v1;
    point v2;
    point v3;
  public:
    triangle( )
    {}
    triangle( point p1, point p2, point p3);
    triangle(int px1, int py1, int px2, int py2, int px3, int py3);

    void setTriangle( int px1, int py1, int px2, int py2, int px3, int py3);
    point getVertex1( ) const;
    point getVertex2( ) const;
    point getVertex3( ) const;
    void setVertex1(point p);
    void setVertex2(point p);
    void setVertex3(point p);

    friend ostream & operator<<(ostream & os, const triangle& t);
};

#endif
```

cont…

```
/*     FILE: ./shapes2/triangle.cpp     */

#include "triangle.h"

triangle::triangle(point p1, point p2, point p3)
{
  setTriangle(p1.getX( ), p1.getY( ), p2.getX( ), p2.getY( ), p3.getX( ), p3.getY( ));
}

triangle::triangle(int px1, int py1, int px2, int py2, int px3, int py3)
{
  setTriangle(px1, py1, px2, py2, px3, py3);
}

void triangle::setTriangle( int px1, int py1, int px2, int py2, int px3, int py3)
{
  v1 = point(px1, py1);
  v2 = point(px2, py2);
  v3 = point(px3, py3);
}

point triangle::getVertex1( ) const
{
  return v1;
}

point triangle::getVertex2( ) const
{
  return v2;
}

point triangle::getVertex3( ) const
{
  return v3;
}

void triangle::setVertex1(point p)
{
  v1 = p;
}

void triangle::setVertex2(point p)
{
  v2 = p;
}

void triangle::setVertex3(point p)
{
  v3 = p;
}

ostream & operator<<(ostream & os, const triangle& t)
{
  os << t.v1 << " " << t.v2 << " " << t.v3;
  return os;
}
```

cont…

```
/*     FILE: Shapes2Test.cpp     */

/*  Classes triangle and circle are using existing class point.

    triangle "has-a" set of points that are vertices.
    circle "is-a" point with a radius.

*/
#include <iostream>

#include "shapes2/point.h"
#include "shapes2/circle.h"
#include "shapes2/triangle.h"

int main( )
{
    point p(7,5);          // Build some shapes
    triangle t(1,2,3,4,1,6);
    circle c(1,2,3.5);

    cout << "\nShapes." << endl;
    cout << "c = " << c << endl; // Display the shapes
    cout << "t = " << t << endl;
    cout << "p = " << p << endl;

    c = circle(p,1.5);

    cout << "c = " << c << endl;

    circle c2(c);      // Exercise some Circle methods

    p = point(3,2);
    c.setCenter(p);
    c.setRadius(3.2);

    cout << "\nCircles." << endl;
    cout << "c = " << c << endl;         // Display exercise results
    cout << "c2 = " << c2 << endl;

    cout << "\nBuild c3 from c and c2." << endl;
    circle c3(c);
    c3.setCenter(c2);
    cout << "c3 = " << c3 << endl;

    cout << "\nAdjust c and c2." << endl;
    c.setCenter(1,1);
    c2.setCenter(0,0);
    cout << "c = " << c << endl;
    cout << "c2 = " << c2 << endl;
    cout << "c3 = " << c3 << endl;

    c3.setCenter(3,3);

    cout << "\nAdjust c3." << endl;
    cout << "c = " << c << endl;
    cout << "c2 = " << c2 << endl;
    cout << "c3 = " << c3 << endl;
}
```

cont…

```
/*    OUTPUT: Shapes2Test.cpp


        Shapes.
        c = C(1,2)  r = 3.5
        t = (1,2) (3,4) (1,6)
        p = (7,5)
        c = C(7,5)  r = 1.5

        Circles.
        c = C(3,2)  r = 3.2
        c2 = C(7,5)  r = 1.5

        Build c3 from c and c2.
        c3 = C(7,5)  r = 3.2

        Adjust c and c2.
        c = C(1,1)  r = 3.2
        c2 = C(0,0)  r = 1.5
        c3 = C(7,5)  r = 3.2

        Adjust c3.
        c = C(1,1)  r = 3.2
        c2 = C(0,0)  r = 1.5
        c3 = C(3,3)  r = 3.2

*/
```

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…

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

#include "point.h"

point::point(double xvalue, double yvalue)
{
  setPoint(xvalue,yvalue);
}

void point::setX(double new_x)
{
  x = new_x;
}

void point::setY(double new_y)
{
  y = new_y;
}

double point::getX( ) const
{
  return x;
}

double point::getY( ) const
{
  return y;
}

void point::setPoint(double new_x, double new_y)
{
  x = new_x;
  y = new_y;
}

void point::move(double x, double y)
{
  setX(x);
  setY(y);
}

void point::shift(double dx, double dy)
{
  setX(getX( )+dx);
  setY(getY( )+dy);
}

void point::print(ostream & os) const
{
  os << "(" << x << "," << y << ")";
}

ostream & operator<<(ostream & os, const point& p)
{
  os << "(" << p.x << "," << p.y << ")";
  return os;
}
```

cont…

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

#ifndef _circle_h
#define _circle_h

#include <iostream>
#include "point.h"

using std::ostream;

class circle:public point{
    double radius;

  public:
    circle( ) {}
    circle(double x, double y, double radius);
    circle(point center, double radius);

    void setCenter(double x, double y);
    void setCenter(point center);
    point getCenter( ) const;

    void scale(double factor);

    void setRadius(double new_r);
    double getRadius( ) const;

    void print(ostream&)const;
    friend ostream & operator<<(ostream & os, const circle& c);
};

#endif
```

cont…

```
/*     FILE: ./shapes3/circle.cpp     */

#include "circle.h"

circle::circle(double x, double y, double radius):point(x,y)
{
  setRadius(radius);
}

circle::circle(point center, double radius):point(center)
{
  setRadius(radius);
}

void circle::setCenter(double x, double y)
{
  setX(x);
  setY(y);
}

void circle::setCenter(point center)
{
  *(point *)this = center;
}

point circle::getCenter( ) const
{
  return *this;
}

void circle::scale(double factor)
{
  radius = fabs(factor)*radius;
}

void circle::setRadius(double new_r)
{
  radius = new_r;
}

double circle::getRadius( ) const
{
  return radius;
}

void circle::print(ostream & os) const
{
  os << "C" << (point)*this << "  r = " << radius;
}

ostream & operator<<(ostream & os, const circle& c)
{
  os << "C" << (point)c << "  r = " << c.radius;
  return os;
}
```

cont…

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

#ifndef _triangle_h
#define _triangle_h

#include <iostream>
#include "point.h"

using std::ostream;

class triangle{
    point v1;
    point v2;
    point v3;
  public:
    triangle( )
    {}
    triangle(point p1, point p2, point p3);
    triangle(double px1, double py1, double px2, double py2, double px3, double py3);

    void setTriangle( double px1, double py1, double px2, double py2, double px3, double py3);
    point getVertex1( ) const;
    point getVertex2( ) const;
    point getVertex3( ) const;
    void setVertex1(point p);
    void setVertex2(point p);
    void setVertex3(point p);

    void print(ostream&)const;
    friend ostream & operator<<(ostream & os, const triangle& t);
};

#endif
```

cont…

```
/*     FILE: ./shapes3/triangle.cpp     */

#include "triangle.h"

triangle::triangle(point p1, point p2, point p3)
{
  setTriangle(p1.getX( ), p1.getY( ), p2.getX( ), p2.getY( ), p3.getX( ), p3.getY( ));
}

triangle::triangle(double px1, double py1, double px2, double py2, double px3, double py3)
{
  setTriangle(px1, py1, px2, py2, px3, py3);
}

void triangle::setTriangle( double px1, double py1, double px2, double py2,
                                                                double px3, double py3)
{
  v1 = point(px1, py1);
  v2 = point(px2, py2);
  v3 = point(px3, py3);
}

point triangle::getVertex1( ) const
{
  return v1;
}

point triangle::getVertex2( ) const
{
  return v2;
}

point triangle::getVertex3( ) const
{
  return v3;
}

void triangle::setVertex1(point p)
{
  v1 = p;
}

void triangle::setVertex2(point p)
{
  v2 = p;
}

void triangle::setVertex3(point p)
{
  v3 = p;
}

void triangle::print(ostream & os) const
{
  os << v1 << " " << v2 << " " << v3;
}

ostream & operator<<(ostream & os, const triangle& t)
{
  os << t.v1 << " " << t.v2 << " " << t.v3;
  return os;
}
```

cont…

```
/*     FILE: Shapes3Test.cpp     */

/*  Classes point and circle are related in such a way that a
    circle "is-a" point with a radius.  Since a circle is-a
    point, it can be treated as a point.
*/
#include <iostream>
using std::cout;
using std::endl;
using std::cin;

#include "shapes3/point.h"
#include "shapes3/circle.h"
#include "shapes3/triangle.h"

int main( )
{
    point *ptr;          // pointer to a point

    point p(7,5);          // create a couple of shapes
    circle c(1,2,3.5);

    cout << "\nShapes:" << endl;
    cout << "c = " << c << endl; // Display the shapes
    cout << "p = " << p << endl;

    ptr = &p;   // ptr tracks a point
    ptr->move(3,3);

    cout << "ptr points to  = " << *ptr << endl;

    ptr = &c;           // ptr tracks a circle, since a circle
    ptr->move(4,4);   // ... is-a point
    cout << "ptr points to  = " << *ptr << endl;

    cout << "\nShapes:" << endl;
    cout << "c = " << c << endl; // Display the shapes
    cout << "p = " << p << endl;

}


/*    OUTPUT: Shapes3Test.cpp


        Shapes:
        c = C(1,2)  r = 3.5
        p = (7,5)
        ptr points to  = (3,3)
        ptr points to  = (4,4)

        Shapes:
        c = C(4,4)  r = 3.5
        p = (3,3)

*/
```

Ex:

```
/*     FILE: Shapes3Test2.cpp     */

/*  Classes point and circle are related in such a way that a
    circle "is-a" point with a radius.  Since a circle is-a
    point, it can be treated as a point.

    Note: As far the compiler is concerned, ptr is only tracking
          a point and anything referred to by the point pointer
          will be treated as a point.
*/
#include <iostream>
using std::cout;
using std::endl;
using std::cin;

#include "shapes3/point.h"
#include "shapes3/circle.h"
#include "shapes3/triangle.h"

int main( )
{
    point *ptr;          // pointer to a point

    point p(7,5);          // create a couple of shapes
    circle c(1,2,3.5);

    cout << "\nShapes:" << endl;
    cout << "c = " << c << endl; // Display the shapes
    cout << "p = " << p << endl;

    ptr = &p;   // ptr tracks a point
    ptr->move(3,3);

    cout << "ptr points to  = ";
    ptr->print(cout);
    cout << endl;

    ptr = &c;           // ptr tracks a circle, since a circle
    ptr->move(4,4);   // ... is-a point
    cout << "ptr points to  = ";
    ptr->print(cout);
    cout << endl;

    cout << "\nShapes:" << endl;
    cout << "c = " << c << endl; // Display the shapes
    cout << "p = " << p << endl;

}


/*    OUTPUT: Shapes3Test2.cpp


       Shapes:
       c = C(1,2)  r = 3.5
       p = (7,5)
       ptr points to  = (3,3)
       ptr points to  = (4,4)

       Shapes:
       c = C(4,4)  r = 3.5
       p = (3,3)

*/
```

Ex:

```
/*     FILE: ./shapes4/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;

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

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

cont…

```
/*     FILE: ./shapes4/point.cpp     */

#include "point.h"

point::point(double xvalue, double yvalue)
{
  setPoint(xvalue,yvalue);
}

void point::setX(double new_x)
{
  x = new_x;
}

void point::setY(double new_y)
{
  y = new_y;
}

double point::getX( ) const
{
  return x;
}

double point::getY( ) const
{
  return y;
}

void point::setPoint(double new_x, double new_y)
{
  x = new_x;
  y = new_y;
}

void point::move(double x, double y)
{
  setX(x);
  setY(y);
}

void point::shift(double dx, double dy)
{
  setX(getX( )+dx);
  setY(getY( )+dy);
}

void point::print(ostream & os) const
{
  os << "(" << x << "," << y << ")";
}

ostream & operator<<(ostream & os, const point& p)
{
  os << "(" << p.x << "," << p.y << ")";
  return os;
}
```

cont…

```
/*     FILE: ./shapes4/circle.h     */

#ifndef _circle_h
#define _circle_h

#include <iostream>
using std::ostream;

#include "point.h"

class circle:public point{
    double radius;

  public:
    circle( ) {}
    circle(double x, double y, double radius);
    circle(point center, double radius);

    void setCenter(double x, double y);
    void setCenter(point center);
    point getCenter( ) const;

    void scale(double factor);

    void setRadius(double new_r);
    double getRadius( ) const;

    void print(ostream&)const;
    friend ostream & operator<<(ostream & os, const circle& c);
};

#endif
```

cont…

```
/*     FILE: ./shapes4/circle.cpp     */

#include "circle.h"

circle::circle(double x, double y, double radius):point(x,y)
{
  setRadius(radius);
}

circle::circle(point center, double radius):point(center)
{
  setRadius(radius);
}

void circle::setCenter(double x, double y)
{
  setX(x);
  setY(y);
}

void circle::setCenter(point center)
{
  *(point *)this = center;
}

point circle::getCenter( ) const
{
  return *this;
}

void circle::scale(double factor)
{
  radius = fabs(factor)*radius;
}

void circle::setRadius(double new_r)
{
  radius = new_r;
}

double circle::getRadius( ) const
{
  return radius;
}

void circle::print(ostream & os) const
{
  os << "C" << (point)*this << "  r = " << radius;
}

ostream & operator<<(ostream & os, const circle& c)
{
  os << "C" << (point)c << "  r = " << c.radius;
  return os;
}
```

```
/*     FILE: ./shapes4/triangle.h     */

#ifndef _triangle_h
#define _triangle_h

#include <iostream>
using std::ostream;
#include "point.h"

class triangle{
    point v1;
    point v2;
    point v3;
  public:
    triangle( )
    {}
    triangle(point p1, point p2, point p3);
    triangle(double px1, double py1, double px2, double py2, double px3, double py3);

    void setTriangle( double px1, double py1, double px2, double py2, double px3, double py3);
    point getVertex1( ) const;
    point getVertex2( ) const;
    point getVertex3( ) const;
    void setVertex1(point p);
    void setVertex2(point p);
    void setVertex3(point p);

    void print(ostream&)const;
    friend ostream & operator<<(ostream & os, const triangle& t);
};

#endif
```

cont…

```
/*     FILE: ./shapes4/triangle.cpp     */

#include "triangle.h"

triangle::triangle(point p1, point p2, point p3)
{
  setTriangle(p1.getX( ), p1.getY( ), p2.getX( ), p2.getY( ), p3.getX( ), p3.getY( ));
}

triangle::triangle(double px1, double py1, double px2, double py2, double px3, double py3)
{
  setTriangle(px1, py1, px2, py2, px3, py3);
}

void triangle::setTriangle( double px1, double py1, double px2, double py2,
                                                                 double px3, double py3)
{
  v1 = point(px1, py1);
  v2 = point(px2, py2);
  v3 = point(px3, py3);
}

point triangle::getVertex1( ) const
{
  return v1;
}

point triangle::getVertex2( ) const
{
  return v2;
}

point triangle::getVertex3( ) const
{
  return v3;
}

void triangle::setVertex1(point p)
{
  v1 = p;
}

void triangle::setVertex2(point p)
{
  v2 = p;
}

void triangle::setVertex3(point p)
{
  v3 = p;
}

void triangle::print(ostream & os) const
{
  os << v1 << " " << v2 << " " << v3;
}

ostream & operator<<(ostream & os, const triangle& t)
{
  os << t.v1 << " " << t.v2 << " " << t.v3;
  return os;
}
```

cont…

```
/*     FILE: Shapes4Test.cpp     */

/*  Classes point and circle are related in such a way that a
    circle "is-a" point with a radius.  Since a circle is-a
    point, it can be treated as a point.

    To allow for run-time determination of the appropriate method
    for the object referenced by a base class pointer a function
    must be declared "virtual".  This causes C++ to handle the
    pointer in such a way that the appropriate method is called.

    The code that uses the classes is the same, but in the classes
    themselves methods are declared virtual.
*/
#include <iostream>
using std::cout;
using std::endl;
using std::cin;

#include "shapes4/point.h"
#include "shapes4/circle.h"
#include "shapes4/triangle.h"

int main( )
{
    point *ptr;          // pointer to a point

    point p(7,5);          // create a couple of shapes
    circle c(1,2,3.5);

    cout << "\nShapes:" << endl;
    cout << "c = " << c << endl; // Display the shapes
    cout << "p = " << p << endl;

    ptr = &p;   // ptr tracks a point
    ptr->move(3,3);

    cout << "ptr points to  = ";
    ptr->print(cout);
    cout << endl;

    ptr = &c;           // ptr tracks a circle, since a circle
    ptr->move(4,4);   // ... is-a point
    cout << "ptr points to  = ";
    ptr->print(cout);
    cout << endl;

    cout << "\nShapes:" << endl;
    cout << "c = " << c << endl; // Display the shapes
    cout << "p = " << p << endl;

}


/*    OUTPUT: Shapes4Test.cpp


        Shapes:
        c = C(1,2)  r = 3.5
        p = (7,5)
        ptr points to  = (3,3)
        ptr points to  = C(4,4)  r = 3.5

        Shapes:
        c = C(4,4)  r = 3.5
        p = (3,3)

*/
```

Ex:

```
/*     FILE: Shapes4Test2.cpp     */

/*  Classes point and circle are related in such a way that a
    circle "is-a" point with a radius.  Since a circle is-a
    point, it can be treated as a point.

    To allow for run-time determination of the appropriate method
    for the object referenced by a base class pointer a function
    must be declared "virtual".  This causes C++ to handle the
    pointer in such a way that the appropriate method is called.

    The code that uses the classes is the same, but in the classes
    themselves methods are declared virtual.

    Note:  The object referred to by the pointer cannot be
           determined until run-time. This is polymorphism.
*/
#include <iostream>
using std::cout;
using std::endl;
using std::cin;

#include "shapes4/point.h"
#include "shapes4/circle.h"
#include "shapes4/triangle.h"

int main( )
{
   point *ptr;          // pointer to a point
   char choice;

   cout << "Enter a 'p' for a point or a 'c' for a circle: ";
   cin >> choice;

   if (choice == 'c')           // Object created is not known until
     ptr = new circle(1,2,3.5); // ... run-time.
   else
     ptr = new point(7,5);


   ptr->move(3,3);

   cout << "ptr points to  = ";  // print( ) prints what it is actually
   ptr->print(cout);             // ... pointing to.
   cout << endl;
}


/*    OUTPUT: Shapes4Test2.cpp

        Enter a 'p' for a point or a 'c' for a circle: c
        ptr points to  = C(3,3)  r = 3.5

*/
```

Ex:

```
/*     FILE: ./shapes5/shape.h     */

#ifndef _shape_h
#define _shape_h

#include <iostream>
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…

```
/*     FILE: ./shapes5/shape.cpp     */

#include "shape.h"

shape::shape(double xvalue, double yvalue)
{
  setShape(xvalue,yvalue);
}

void shape::setX(double new_x)
{
  x = new_x;
}

void shape::setY(double new_y)
{
  y = new_y;
}

double shape::getX( ) const
{
  return x;
}

double shape::getY( ) const
{
  return y;
}

void shape::setShape(double new_x, double new_y)
{
  x = new_x;
  y = new_y;
}

void shape::print(ostream & os) const
{
  os << "(" << x << "," << y << ")";
}

ostream & operator<<(ostream & os, const shape& s)
{
  os << "(" << s.x << "," << s.y << ")";
  return os;
}
```

cont…

```
/*     FILE: ./shapes5/point.h     */

#ifndef _point_h
#define _point_h

#include <iostream>
#include "shape.h"
using std::ostream;
using std::cout;
using std::endl;

class point: public shape{

  public:
    point( ):shape( )
    { }
    point(double xvalue, double yvalue);

    void setPoint(double new_x, double new_y);

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

};
#endif
```

cont…

```
/*     FILE: ./shapes5/point.cpp     */

#include "point.h"

point::point(double xvalue, double yvalue):shape(xvalue,yvalue)
{

}

void point::setPoint(double new_x, double new_y)
{
  setShape(new_x, new_y);
}

void point::move(double x, double y)
{
  setX(x);
  setY(y);
}

void point::shift(double dx, double dy)
{
  setX(getX( )+dx);
  setY(getY( )+dy);
}

void point::rotate(double r)
{

}

void point::draw( )
{
  cout << "Drawing: " << *this << endl;
}
```

cont…

```
/*     FILE: ./shapes5/circle.h     */

#ifndef _circle_h
#define _circle_h

#include <iostream>
#include "shape.h"
#include "point.h"
using std::ostream;

class circle:public point{
    double radius;

  public:
    circle( ) {}
    circle(int x, int y, double radius);
    circle(point center, double radius);

    void setCenter(int x, int y);
    void setCenter(point center);
    point getCenter( ) const;

    void scale(double factor);

    void setRadius(double new_r);
    double getRadius( ) const;

    void print(ostream & os) const;
    friend ostream & operator<<(ostream & os, const circle& c);
};

#endif
```

cont…

```
/*     FILE: ./shapes5/circle.cpp     */

#include "circle.h"

circle::circle(int x, int y, double radius):point(x,y)
{
  setRadius(radius);
}

circle::circle(point center, double radius):point(center)
{
  setRadius(radius);
}

void circle::setCenter(int x, int y)
{
  setX(x);
  setY(y);
}

void circle::setCenter(point center)
{
  *(point *)this = center;
}

point circle::getCenter( ) const
{
  return point((point)*this);
}

void circle::scale(double factor)
{
  radius = fabs(factor)*radius;
}

void circle::setRadius(double new_r)
{
  radius = new_r;
}

double circle::getRadius( ) const
{
  return radius;
}

void circle::print(ostream & os) const
{
  os << "C" << (point)*this << "  r = " << radius;
}

ostream & operator<<(ostream & os, const circle& c)
{
  os << "C" << (point)c << "  r = " << c.radius;
  return os;
}
```

cont…

```
/*     FILE: ./shapes5/triangle.h     */

#ifndef _triangle_h
#define _triangle_h

#include <iostream>
#include "point.h"
using std::ostream;

class triangle{
    point v1;
    point v2;
    point v3;
  public:
    triangle( )
    {}
    triangle(point p1, point p2, point p3);
    triangle(double px1, double py1, double px2, double py2, double px3, double py3);

    void setTriangle( double px1, double py1, double px2, double py2, double px3, double py3);
    point getVertex1( ) const;
    point getVertex2( ) const;
    point getVertex3( ) const;
    void setVertex1(point p);
    void setVertex2(point p);
    void setVertex3(point p);

    void print(ostream & os) const;
    friend ostream & operator<<(ostream & os, const triangle& t);
};

#endif
```

cont…

```
/*     FILE: ./shapes5/triangle.cpp     */

#include "triangle.h"

triangle::triangle(point p1, point p2, point p3)
{
  setTriangle(p1.getX( ), p1.getY( ), p2.getX( ), p2.getY( ), p3.getX( ), p3.getY( ));
}

triangle::triangle(double px1, double py1, double px2, double py2, double px3, double py3)
{
  setTriangle(px1, py1, px2, py2, px3, py3);
}

void triangle::setTriangle( double px1, double py1, double px2, double py2,
                                                                 double px3, double py3)
{
  v1 = point(px1, py1);
  v2 = point(px2, py2);
  v3 = point(px3, py3);
}

point triangle::getVertex1( ) const
{
  return v1;
}

point triangle::getVertex2( ) const
{
  return v2;
}

point triangle::getVertex3( ) const
{
  return v3;
}

void triangle::setVertex1(point p)
{
  v1 = p;
}

void triangle::setVertex2(point p)
{
  v2 = p;
}

void triangle::setVertex3(point p)
{
  v3 = p;
}

void triangle::print(ostream & os) const
{
  os << v1 << " " << v2 << " " << v3;
}

ostream & operator<<(ostream & os, const triangle& t)
{
  os << t.v1 << " " << t.v2 << " " << t.v3;
  return os;
}
```

cont…

```
/*     FILE: Shapes5Test.cpp     */

/*  Class shape has been added to the point and circle hierarchy.
    They are related in such a way that a shape is the base class
    of both point and circle. Clas point and class circle both
    exhibit the "is-a" relationship with shape.

    In addition they exhibit the "is-a" relationship with point,
    since a circle "is-a" point with a radius.

    Note: The shape class contains method declarations with no
          definition. These methods are referred to as "pure virtual"
          methods.  Pure virtual methods imply that a definition
          will be required eventually but as yet, are not defined.
          This impacts the class containing the pure virtual methods
          by making the class "abstract" i.e., you cannot create an
          object of that class.

          You can however still have references of that type.
*/
#include <iostream>
using std::cout;
using std::endl;

#include "shapes5/point.h"
#include "shapes5/circle.h"
#include "shapes5/triangle.h"

int main( )
{
    shape *ptr;            // pointer to a shape

    point p(7,5);          // create a couple of shapes
    circle c(1,2,3.5);

    cout << "\nShapes:" << endl;
    cout << "c = " << c << endl; // Display the shapes
    cout << "p = " << p << endl;

    ptr = &p;   // ptr tracks a point
    ptr->move(3,3);

    cout << "ptr points to  = ";
    ptr->print(cout);
    cout << endl;

    ptr = &c;           // ptr tracks a circle, since a circle
    ptr->move(4,4);   // ... is-a point
    cout << "ptr points to  = ";
    ptr->print(cout);
    cout << endl;

    cout << "\nShapes:" << endl;
    cout << "c = " << c << endl; // Display the shapes
    cout << "p = " << p << endl;

}


/*    OUTPUT: Shapes5Test.cpp


        Shapes:
        c = C(1,2)  r = 3.5
        p = (7,5)
        ptr points to  = (3,3)
        ptr points to  = C(4,4)  r = 3.5

        Shapes:
        c = C(4,4)  r = 3.5
        p = (3,3)

*/
```

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…

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

#include "shape.h"

shape::shape(double xvalue, double yvalue)
{
  setShape(xvalue,yvalue);
}

void shape::setX(double new_x)
{
  x = new_x;
}

void shape::setY(double new_y)
{
  y = new_y;
}

double shape::getX( ) const
{
  return x;
}

double shape::getY( ) const
{
  return y;
}

void shape::setShape(double new_x, double new_y)
{
  x = new_x;
  y = new_y;
}

void shape::print(ostream & os) const
{
  os << "(" << x << "," << y << ")";
}

ostream & operator<<(ostream & os, const shape& s)
{
  s.print(os);
  return os;
}
```

cont…

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

#ifndef _point_h
#define _point_h

#include <iostream>
#include "shape.h"
using std::cout;
using std::endl;
using std::ostream;

class point: public shape{

  public:
    point( ):shape( )
    { }
    point(double xvalue, double yvalue);

    void setPoint(double new_x, double new_y);

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

};
#endif
```

cont…

```
/*     FILE: ./shapes6/point.cpp     */

#include "point.h"

point::point(double xvalue, double yvalue):shape(xvalue,yvalue)
{

}

void point::setPoint(double new_x, double new_y)
{
  setShape(new_x, new_y);
}

void point::move(double x, double y)
{
  setX(x);
  setY(y);
}

void point::shift(double dx, double dy)
{
  setX(getX( )+dx);
  setY(getY( )+dy);
}

void point::rotate(double r)
{

}

void point::draw( )
{
  cout << "Drawing: " << *this << endl;
}
```

cont…

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

#ifndef _circle_h
#define _circle_h

#include <iostream>
#include "shape.h"
#include "point.h"
using std::cout;
using std::endl;
using std::ostream;

class circle:public point{
    double radius;

  public:
    circle( ) {}
    circle(int x, int y, double radius);
    circle(point center, double radius);

    void setCenter(int x, int y);
    void setCenter(point center);
    point getCenter( ) const;

    void scale(double factor);

    void setRadius(double new_r);
    double getRadius( ) const;

    void print(ostream & os) const;
    friend ostream & operator<<(ostream & os, const circle& c);
};

#endif
```

cont…

```
/*     FILE: ./shapes6/circle.cpp     */

#include "circle.h"

circle::circle(int x, int y, double radius):point(x,y)
{
  setRadius(radius);
}

circle::circle(point center, double radius):point(center)
{
  setRadius(radius);
}

void circle::setCenter(int x, int y)
{
  setX(x);
  setY(y);
}

void circle::setCenter(point center)
{
  *(point *)this = center;
}

point circle::getCenter( ) const
{
  return point((point)*this);
}

void circle::scale(double factor)
{
  radius = fabs(factor)*radius;
}

void circle::setRadius(double new_r)
{
  radius = new_r;
}

double circle::getRadius( ) const
{
  return radius;
}

void circle::print(ostream & os) const
{
  os << "C" << (point)*this << "  r = " << radius;
}

ostream & operator<<(ostream & os, const circle& c)
{
  c.print(os);
  return os;
}
```

cont…

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

#ifndef _triangle_h
#define _triangle_h

#include <iostream>
#include "point.h"
using std::cout;
using std::endl;
using std::ostream;

class triangle{
    point v1;
    point v2;
    point v3;
  public:
    triangle( )
    {}
    triangle(point p1, point p2, point p3);
    triangle(double px1, double py1, double px2, double py2, double px3, double py3);

    void setTriangle( double px1, double py1, double px2, double py2, double px3, double py3);
    point getVertex1( ) const;
    point getVertex2( ) const;
    point getVertex3( ) const;
    void setVertex1(point p);
    void setVertex2(point p);
    void setVertex3(point p);

    void print(ostream & os) const;
    friend ostream & operator<<(ostream & os, const triangle& t);
};

#endif
```

cont…

```
/*     FILE: ./shapes6/triangle.cpp     */

#include "triangle.h"

triangle::triangle(point p1, point p2, point p3)
{
  setTriangle(p1.getX( ), p1.getY( ), p2.getX( ), p2.getY( ), p3.getX( ), p3.getY( ));
}

triangle::triangle(double px1, double py1, double px2, double py2, double px3, double py3)
{
  setTriangle(px1, py1, px2, py2, px3, py3);
}

void triangle::setTriangle( double px1, double py1, double px2, double py2,
                                                                double px3, double py3)
{
  v1 = point(px1, py1);
  v2 = point(px2, py2);
  v3 = point(px3, py3);
}

point triangle::getVertex1( ) const
{
  return v1;
}

point triangle::getVertex2( ) const
{
  return v2;
}

point triangle::getVertex3( ) const
{
  return v3;
}

void triangle::setVertex1(point p)
{
  v1 = p;
}

void triangle::setVertex2(point p)
{
  v2 = p;
}

void triangle::setVertex3(point p)
{
  v3 = p;
}

void triangle::print(ostream & os) const
{
  os << v1 << " " << v2 << " " << v3;
}

ostream & operator<<(ostream & os, const triangle& t)
{
  t.print(os);
  return os;
}
```

cont…

```
/*     FILE: Shapes6Test.cpp     */

/*  How to make a "virtual" friend function.

    The put-to operator << that allows run-time
    type determination.
*/
#include <iostream>
using std::cout;
using std::endl;

#include "shapes6/point.h"
#include "shapes6/circle.h"
#include "shapes6/triangle.h"

int main( )
{
    shape *ptr;            // pointer to a shape

    point p(7,5);          // create a couple of shapes
    circle c(1,2,3.5);

    cout << "\nShapes:" << endl;
    cout << "c = " << c << endl; // Display the shapes
    cout << "p = " << p << endl;

    ptr = &p;   // ptr tracks a point
    ptr->move(3,3);
                               // run-time type determination of <<
    cout << "ptr points to  = " << *ptr << endl;

    ptr = &c;           // ptr tracks a circle, since a circle
    ptr->move(4,4);   // ... is-a point
                               // run-time type determination of <<
    cout << "ptr points to  = " << *ptr << endl;

}


/*    OUTPUT: Shapes6Test.cpp


        Shapes:
        c = C(1,2)  r = 3.5
        p = (7,5)
        ptr points to  = (3,3)
        ptr points to  = C(4,4)  r = 3.5

*/
```

OPENGL/GLUT

- OpenGL is an open source Graphics Library. There are versions available for most major operating system. There's versions for the WinTel platform.
- Using the Graphics Library Utility Toolkit (GLUT) you can write C/C++ code with very little effort that produces windowing and graphics capability.
- See the Appendix for this document for additional notes and resources regarding OpenGL and GLUT.

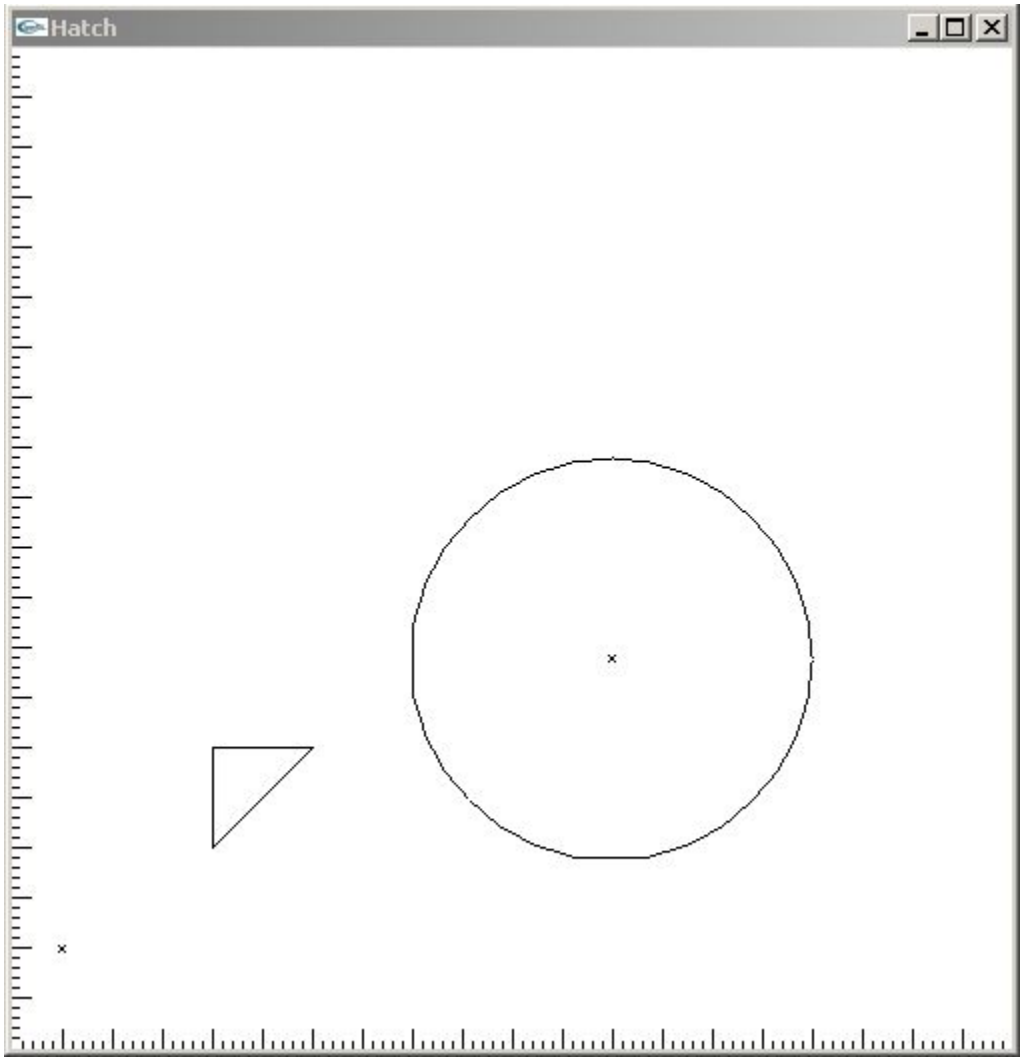
Ex:

```
/*     FILE: ./OpenGLExamples/hatch.cpp     */

#include <stdlib.h>
#include "GL/glut.h"
#include "../shapes7/point.h"
#include "../shapes7/triangle.h"
#include "../shapes7/circle.h"

int scrWidth = 500, scrHeight = 500;
point p(25,50);
triangle t(100,100,100,150,150,150);
circle c(300,195,100);
void myinit( );
void display( );
void reshape(int, int);
void hatch( );

void myinit( )
{
  glClearColor(1.,1.,1.,1.);          /* white background */
  glColor3f(0.,0.,0.);               /* black foreground */
  glShadeModel(GL_FLAT);
  /* set up viewing scrWidth x scrHeight window with origin lower left */
  glViewport(0,0,scrWidth,scrHeight);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity( );
  gluOrtho2D(-(GLdouble)scrWidth/2,(GLdouble)scrWidth/2,
        -(GLdouble)scrHeight/2,(GLdouble)scrHeight/2);     /* ensure aspect ratio */
  glMatrixMode(GL_MODELVIEW);
}

void display( )
{
  glClear(GL_COLOR_BUFFER_BIT);

  glBegin(GL_LINES);

  hatch( );
  p.draw( );
  t.draw( );
  c.draw( );

  glEnd( );

  glFlush( );

}
```

cont…

```
void hatch( )
{
  int halfHeight = scrHeight/2;
  int halfWidth = scrWidth/2;
  for(int i=5; i < scrHeight; i+=5){
    glVertex2i(0,i);
    glVertex2i(4,i);

    if(i%25 ==0){
         glVertex2i(0,i);
      glVertex2i(10,i);
    }
  }
  for(int i=5; i < scrWidth; i+=5){
    glVertex2i(i,0);
    glVertex2i(i,4);

    if(i%25 ==0){
         glVertex2i(i,0);
      glVertex2i(i,10);
    }
  }
}

void reshape(int nescrWidth, int nescrHeight)
{
  scrHeight = nescrHeight;
  scrWidth = nescrWidth;
  glViewport(0,0,scrWidth,scrHeight);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity( );
  gluOrtho2D(0.,(GLdouble)scrWidth,0.,(GLdouble)scrHeight);
  glMatrixMode(GL_MODELVIEW);
  display( );
}

int main(int argc, char** argv)
{
  glutInit(&argc,argv);
  glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
  glutInitWindowSize(scrWidth,scrHeight);
  glutCreateWindow("Hatch");
  glutDisplayFunc(display);
  myinit( );
  glutReshapeFunc(reshape);
  glutMainLoop( );

  return 0;
}
```

cont…



Ex:

```
/*     FILE: shapes7/shape.h     */

#ifndef _shape_h
#define _shape_h

#include <iostream>
using std::ostream;
#include "..\glut.h"

class shape{
    int x,y;

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

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

    virtual void move(int x, int y) = 0;
    virtual void shift(int dx, int 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…

```
/*     FILE: shapes7/shape.cpp     */

#include "shape.h"

shape::shape(int xvalue, int yvalue)
{
  setShape(xvalue,yvalue);
}

void shape::setX(int new_x)
{
  x = new_x;
}

void shape::setY(int new_y)
{
  y = new_y;
}

int shape::getX( ) const
{
  return x;
}

int shape::getY( ) const
{
  return y;
}

void shape::setShape(int new_x, int new_y)
{
  x = new_x;
  y = new_y;
}

void shape::print(ostream & os) const
{
  os << "(" << x << "," << y << ")";
}

ostream & operator<<(ostream & os, const shape& s)
{
  s.print(os);
  return os;
}
```

cont…

```
/*     FILE: shapes7/point.h     */

#ifndef _point_h
#define _point_h

#include <iostream>
#include "shape.h"

class point: public shape{

  public:
    point( ):shape( )
    { }
    point(int xvalue, int yvalue);

    void setPoint(int new_x, int new_y);

    void move(int x, int y);
    void shift(int dx, int dy);
    void rotate(double r);
    void draw( );

};
#endif



/*     FILE: shapes7/point.cpp     */

#include "point.h"

point::point(int xvalue, int yvalue):shape(xvalue,yvalue)
{

}

void point::setPoint(int new_x, int new_y)
{
  setShape(new_x, new_y);
}

void point::move(int x, int y)
{
  setX(x);
  setY(y);
}

void point::shift(int dx, int dy)
{
  setX(getX( )+dx);
  setY(getY( )+dy);
}

void point::rotate(double r)
{

}

void point::draw( )
{
  glVertex2i(getX( )-2,getY( )-2);
  glVertex2i(getX( )+2,getY( )+2);

  glVertex2i(getX( )-2,getY( )+2);
  glVertex2i(getX( )+2,getY( )-2);

}
```

cont…

```
/*     FILE: shapes7/circle.h     */

#ifndef _circle_h
#define _circle_h

#include <iostream>
#include "point.h"

class circle:public point{
    double radius;

  public:
    circle( ) {}
    circle(int x, int y, double radius);
    circle(point center, double radius);

    void setCenter(int x, int y);
    void setCenter(point center);
    point getCenter( ) const;

    void scale(double factor);

    void setRadius(double new_r);
    double getRadius( ) const;

    void draw( );

    void print(ostream & os) const;
    friend ostream & operator<<(ostream & os, const circle& c);
};

#endif
```

```
/*     FILE: shapes7/circle.cpp     */

#include "circle.h"
#include <cmath>

circle::circle(int x, int y, double radius):point(x,y)
{
  setRadius(radius);
}

circle::circle(point center, double radius):point(center)
{
  setRadius(radius);
}

void circle::setCenter(int x, int y)
{
  setX(x);
  setY(y);
}

void circle::setCenter(point center)
{
  *(point *)this = center;
}

point circle::getCenter( ) const
{
  return point((point)*this);
}

void circle::scale(double factor)
{
  radius = fabs(factor)*radius;
}
```

cont…

```
void circle::setRadius(double new_r)
{
  radius = new_r;
}

double circle::getRadius( ) const
{
  return radius;
}

void circle::draw( )
{
       int n = 32;
       double angle = 2*M_PI/n;

  glVertex2i(getX( )-2,getY( )-2);
  glVertex2i(getX( )+2,getY( )+2);

  glVertex2i(getX( )-2,getY( )+2);
  glVertex2i(getX( )+2,getY( )-2);

  for(int i=1; i<=n; i++){
    glVertex2i((int)(getX( )+radius*cos((i-1)*angle)),(int)(getY( )+radius*sin((i-1)*angle)));
    glVertex2i((int)(getX( )+radius*cos(i*angle)),(int)(getY( )+radius*sin(i*angle)));
  }

}

void circle::print(ostream & os) const
{
  os << "C" << (point)*this << "  r = " << radius;
}

ostream & operator<<(ostream & os, const circle& c)
{
  c.print(os);
  return os;
}
```

cont…

```
/*     FILE: shapes7/triangle.h     */

#ifndef _triangle_h
#define _triangle_h

#include <iostream>
#include "point.h"

class triangle{
    point v1;
    point v2;
    point v3;
  public:
    triangle( )
    {}
    triangle(point p1, point p2, point p3);
    triangle(int px1, int py1, int px2, int py2, int px3, int py3);

    void setTriangle( int px1, int py1, int px2, int py2, int px3, int py3);
    point getVertex1( ) const;
    point getVertex2( ) const;
    point getVertex3( ) const;
    void setVertex1(point p);
    void setVertex2(point p);
    void setVertex3(point p);

    void draw( );
    void print(ostream & os) const;
    friend ostream & operator<<(ostream & os, const triangle& t);
};
#endif
```

cont…

```
/*     FILE: shapes7/triangle.cpp     */

#include "triangle.h"

triangle::triangle(point p1, point p2, point p3)
{
  setTriangle(p1.getX( ), p1.getY( ), p2.getX( ), p2.getY( ), p3.getX( ), p3.getY( ));
}

triangle::triangle(int px1, int py1, int px2, int py2, int px3, int py3)
{
  setTriangle(px1, py1, px2, py2, px3, py3);
}

void triangle::setTriangle( int px1, int py1, int px2, int py2, int px3, int py3)
{
  v1 = point(px1, py1);
  v2 = point(px2, py2);
  v3 = point(px3, py3);
}

point triangle::getVertex1( ) const
{
  return v1;
}

point triangle::getVertex2( ) const
{
  return v2;
}

point triangle::getVertex3( ) const
{
  return v3;
}

void triangle::setVertex1(point p)
{
  v1 = p;
}

void triangle::setVertex2(point p)
{
  v2 = p;
}

void triangle::setVertex3(point p)
{
  v3 = p;
}
```

cont…

```
void triangle::draw( )
{
  point p1 = getVertex1( );
  point p2 = getVertex2( );

  glVertex2i(p1.getX( ), p1.getY( ));
  glVertex2i(p2.getX( ), p2.getY( ));

  p1 = getVertex2( );
  p2 = getVertex3( );

  glVertex2i(p1.getX( ), p1.getY( ));
  glVertex2i(p2.getX( ), p2.getY( ));

  p1 = getVertex3( );
  p2 = getVertex1( );

  glVertex2i(p1.getX( ), p1.getY( ));
  glVertex2i(p2.getX( ), p2.getY( ));

}

void triangle::print(ostream & os) const
{
  os << v1 << " " << v2 << " " << v3;
}

ostream & operator<<(ostream & os, const triangle& t)
{
  t.print(os);
  return os;
}
```

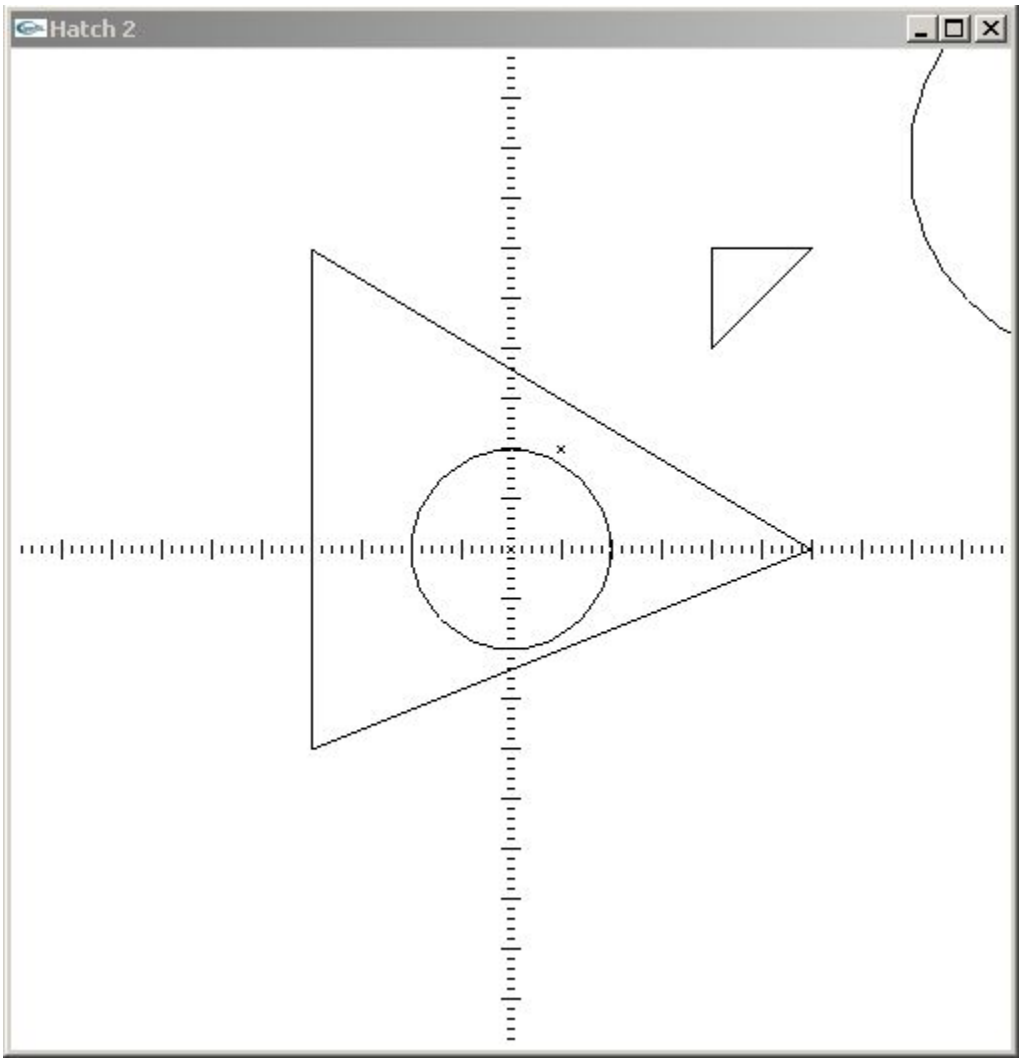
cont…

```
/*     FILE: ./OpenGLExamples/hatch2.cpp     */

#include <stdlib.h>
#include "GL/glut.h"
#include "..\shapes7\point.h"
#include "..\shapes7\triangle.h"
#include "..\shapes7\circle.h"

int scrWidth = 500, scrHeight = 500;
point p(25,50);
triangle t(100,100,100,150,150,150);
triangle t2(-100,-100,-100,150,150,0);
circle c(300,195,100);
circle c2(0,0,50);
void myinit( );
void display( );
void reshape(int, int);
void hatch( );

void myinit( )
{
  glClearColor(1.,1.,1.,1.);          /* white background */
  glColor3f(0.,0.,0.);                /* black foreground */
  glShadeModel(GL_FLAT);
  /* set up viewing scrWidth x scrHeight window with origin lower left */
  glViewport(0,0,scrWidth,scrHeight);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity( );
  gluOrtho2D(-(GLdouble)scrWidth/2,(GLdouble)scrWidth/2,-
                       (GLdouble)scrHeight/2,(GLdouble)scrHeight/2);
                                                    /* ensure aspect ratio */
  glMatrixMode(GL_MODELVIEW);
}

void display( )
{
  glClear(GL_COLOR_BUFFER_BIT);

  glBegin(GL_LINES);

  hatch( );
  p.draw( );
  t.draw( );
  c.draw( );
  t2.draw( );
  c2.draw( );

  glEnd( );

  glFlush( );

}
```

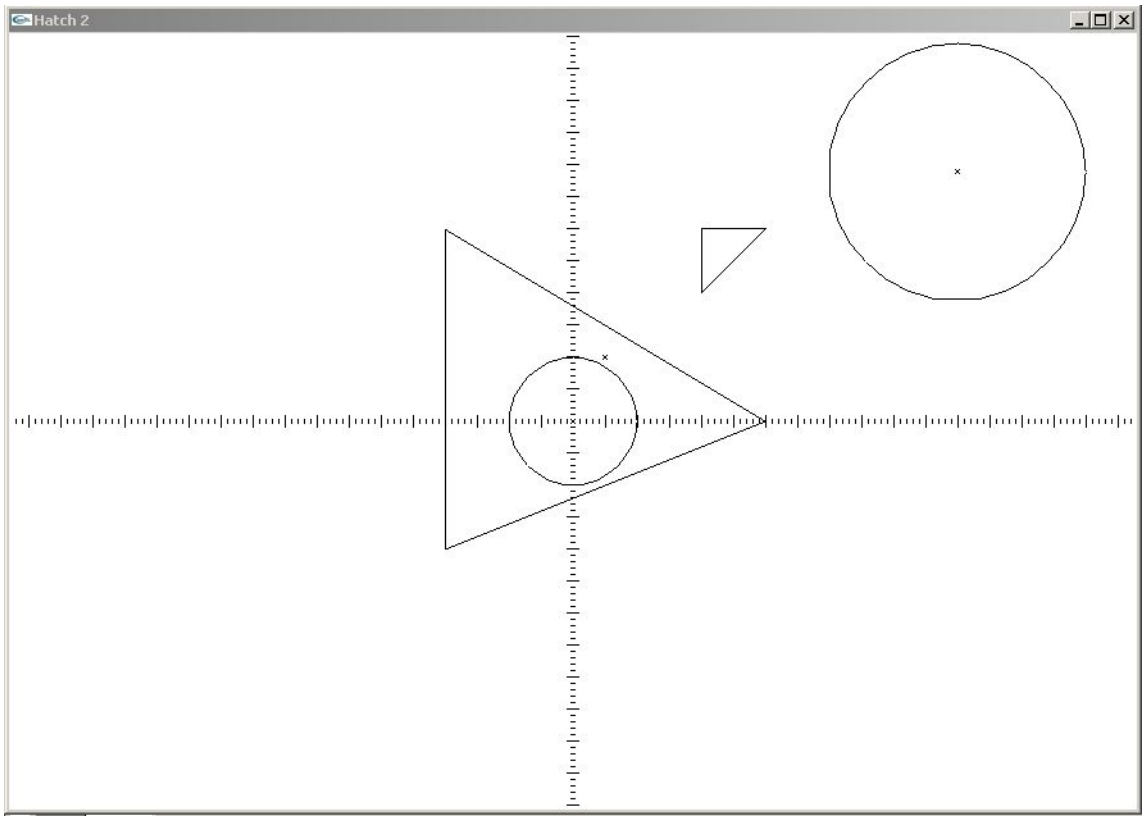
cont…

```
void hatch( )
{
  int halfHeight = scrHeight/2;
  int halfWidth = scrWidth/2;
  for(int i=5; i < halfHeight; i+=5){
    glVertex2i(-2,i);
    glVertex2i(2,i);

    glVertex2i(-2,-i);
    glVertex2i(2,-i);

    if(i%25 ==0){
         glVertex2i(-5,i);
      glVertex2i(5,i);
         glVertex2i(-5,-i);
      glVertex2i(5,-i);
    }
  }
  for(int i=5; i < halfWidth; i+=5){
    glVertex2i(i,-2);
    glVertex2i(i,2);

    glVertex2i(-i,-2);
    glVertex2i(-i,2);

    if(i%25 ==0){
      glVertex2i(i,-5);
      glVertex2i(i,5);
      glVertex2i(-i,-5);
      glVertex2i(-i,5);
    }
  }
}

void reshape(int nescrWidth, int nescrHeight)
{
  scrHeight = nescrHeight;
  scrWidth = nescrWidth;
  glViewport(0,0,scrWidth,scrHeight);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity( );
  gluOrtho2D(-(GLdouble)scrWidth/2,(GLdouble)scrWidth/2,-
(GLdouble)scrHeight/2,(GLdouble)scrHeight/2);
  glMatrixMode(GL_MODELVIEW);
  display( );
}

int main(int argc, char** argv)
{
  glutInit(&argc,argv);
  glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
  glutInitWindowSize(scrWidth,scrHeight);
  glutCreateWindow("Hatch 2");
  glutDisplayFunc(display);
  myinit( );
  glutReshapeFunc(reshape);
  glutMainLoop( );

  return 0;
}
```

cont…



cont…



INDEX