Introduction to the

C++ Programming Language

Dr. David J. Ennis

Ohio Supercomputer Center

The two aspects of C++...

1. A Better C

   -Providing features that make common C errors unlikely

   -C++ is a <u>superset</u> of C

   -Extensions to C

   -Improvements to C performance/ease-of-use

2. Enabling Object-Oriented Programming (OOP)

   -Entirely new approach to programming: next major programming style beyond functional decomposition

      *Model real world by making data types that replicate real objects

      *Facilitates code reuse: reuse <u>and</u> enhance existing libraries

   -Main program consists of creation and manipulation of objects

   -*Object= data + functions* which allow you to interact with the data

   -In general, data controlled only through these interface functions: **Data Encapsulation**

   -C operators can work with your objects: **Operator Overloading**

   -"Advanced" Objects can incorporate features of "Simpler" objects: **Inheritance**

   -With the same syntax, objects can act differently in different situations: **Polymorphism**

A Better C: Table of Contents

End-of-Line Comment

Single line form of a comment begun with the // symbol

- Everything from the delimiter // to the end-of-the line is ignored
- Reminiscent of the Fortran 90 use of ! for one-line commenting

Familiar, multi-line style of commenting still works in C++

- Warning: /*... */ commenting may not nest with some compilers

Sample Program:

```
#include <stdio.h>
main () {
  float r,theta; // polar coordinates
  float x,y,z; // Cartesian coordinates
             // only works for one line
  /*  Code might need later
  x=10; y=100;
  z=x*3+2-y/4;
  printf("z is %d \n",z);
  */
}
```

Type-casting Syntax

In C, override type casting was possible with the following (awkward) syntax

```
int i;
(double) i
```

In C++, a second (less awkward) syntax can also be used:

```
double(i)
char(3+'A')
```

Additional usefulness of this new syntax will be seen later in the OOP section.

Input/Output Streams

Probably the strangest thing you would notice when looking at a C++ for the first time...

Replacement of the stdio library of C with the iostream library of C++. For example, the C++ version of the classic "Hello World" program look as follows:

```
#include <iostream.h>
main () {
  cout << "Hello World\n";
}
```

Notice the three changes:

1. Include iostream.h instead of stdio.h
2. **cout** indicates that the output should go the standard out. There are corresponding keywords **cin** and **cerr** which represent standard in and standard error, respectively
3. Use of the *insertion operator* **<<** as opposed to the printf function. The corresponding *extraction operator* for input is **>>**

Advantages of iostream approach:

-Can <u>extend</u> the insertion/extraction operators to work as you wish for your own data types (structures variables/objects)

-Can just use default format for quick output display [Analogous to Fortran print *]

-Stream approach more type-safe. Avoid run-time mismatch of format specifiers and their "matching" arguments in printf and scanf.

-Better performance: avoids run-time interpretation of format specifiers in stdio library functions

I/O Streams: Default formats

```
#include <iostream.h>
main () {
  int i=44;
  float f=10.40;
  double d=2.718281828459045;
  char c='E';
  char *s="Logan";

  cout << c << endl;
  cout << s << endl;
  cout << f << endl;
  cout << d << endl;
  cout << i << endl;
  cout << s << c << f << i << endl;
  cout << &i << endl;
}
------------------------------------
E
Logan
10.4
2.71828
44
LoganE10.444
0xae41
```

Formatting I/O streams

Formatted output is performed by including **format manipulators** (such as the just seen endl) into the stream

Include the iomanip.h file in your code header

Some popular manipulators and their functions:

" "	-string of desired spaces
"/t"	-horizontal tab
setw(w)	-set field width to w
setfill(c)	-set fill character to c [blank is default]
setprecision(p)	-set float precision to p
dec,oct,hex,	-use the indicated base

Sample Program:

```
#include <iostream.h>
#include <iomanip.h>
main () {
  cout << "[" << setw(6) << setfill('*') << 192;
  cout << "]" << endl;
  cout << hex << "[" << setw(6);
  cout << setfill('^') << 192 << "]" << endl;
  cout << setprecision(4) << 3.14159 << endl;
}
---------------------------------------------
[***192]
[^^^^c0]
3.142
```

The Input Stream

Analogous to output stream use in syntax and use format manipulators.

```
int c,d;
cin >> c >> d;
```

Advantage: Avoid the common error of forgetting the arguments to scanf are addresses.

Compare

```
cout << "Number of time steps: ";
cin >> T
```

to

```
printf("Number of time steps: ");
scanf("%d",&T);
```

The I/O stream approach also exist for files <fstream.h> and strings <strstream.h>

Using the cerr Stream

```
#include <iostream.h>
main() {
  int number=34;
  float divisor;
  cout << number << " divided by 2.3 equals "
         << number/2.3 << endl;
  cout << "Please enter you own divisor: ";
  cin >> divisor;
  if (divisor==0.0)
    cerr << "Cannot divide by zero" << endl;
  else
    cout << "Results is " << number/divisor << endl;
}
---------------------------------------------------
(981) conan% a.out
34 divided by 2.3 equals 14.7826
Please enter you own divisor: 0.0
Cannot divide by zero
(982) conan% a.out
34 divided by 2.3 equals 14.7826
Please enter you own divisor: 78.2
Results is 0.434783
```

Movable Variable Declarations

Variables can be declared anywhere in C++ code. Local variables need not all be declared at the beginning of a function.

-Similar to implicit-typing and variable creation/use in Fortran

Variables come into existence when declared and cease existence when the present code block is exit

-Don't tie-up memory until you *really* need it and free it up when you don't need it anymore.

-Improve code readability ("intuitively right" places for declaration)

Popular use of this feature is with loop counter:

```
float sum(float *a,int N) {
      float res=0.0;
      for(int i=0;i<N;++i)
        res += a[i];
      return(res);
}
```

Warning(*compiler-dependent*): scope of int extends to end of entire sum function. So if included another "int i" statement anywhere else is the sum function would get compiler error for redeclaration of the same variable.

Possible performance improvement: make loop counter register class as well

```
for(register int i=0;......
```

Symbolic Constants in C++

In C, there exists the standard macro method for declaring symbolic constants:

```
#define LIMIT 2300
```

Two problems with this approach are that macros lack type and the macro name is typically not available to a debugger.

In C++ (and ANSI C), the **const** keyword can be used at declaration to indicated that the identifier <u>cannot be subsequently changed</u>:

```
const int LIMIT=2300;
```

LIMIT now has scope and type and is tracked in the name space used by the compiler and most debuggers. Making symbolic constants as long been a C/C++ convention.

The const keyword can also be used in function argument list to insure that the argument cannot be changed in the body of the function:

```
void bobba(const int i)  {
        i=5   // Illegal!
        ...
```

C++ Memory allocation functions

In C++, the malloc library of dynamic memory allocations functions are replaced with the **new** and **delete** operators built-in to the language.The justification for this change is that the "new" operators are easier to use: avoiding the ancillary use of the C sizeof function and extraneous type-casting. This is especially true with user-defined types.

Warning: <u>do not</u> mix & match C's malloc() and free() with C++'s new and delete

Sample code for C++ dynamic allocation of a single variable type:

```
float *fp;
fp=new float;
*fp=9.87;
cout << *fp << endl;
delete fp;
```

As this program demonstrates, the **new** operator returns the address of a block of memory large enough to hold a variable of the argument type. The **delete** operator then frees up the memory when it is no longer needed.

A similar procedure with slightly different syntax is used for dynamic memory allocation for entire arrays:

```
float *a;
a=new float[50];
a[44]=34.5;
delete [] a;
```

<u>Rule of the thumb</u>: Whenever new is followed by [], delete should be followed by it to.

But perhaps, the most convincing proof of how easily new & delete allow dynamic memory control is when working with user defined data types. Consider this classic linked list example:

```
#include <iostream.h>

struct node {
  int id;
  double value;
  node *next;
};

main() {
  node *current;
  current=new node;
  current->id=78;
  current->value=45.67;
  cout << current->id << endl;
  cout << current->value << endl;
  delete current;
}
```

Type Checking

In general, C++ uses very **strict type checking**.

With respect to functions, C++ (as well as ANSI C) requires that functions be prototyped before their use.This is to insure that the compiler can check that

- the *proper number* of arguments are passed

-where possible and if needed, cast the actual arguments to the *proper type*

- the *return type* is *properly* used

To illustrate the power of function prototyping and argument checking, the following code

```
#include <iostream.h>
int firstchar(char *s);
main () {
  float name=3.0;
  int ascii;
  ascii=firstchar(name);
  cout << "Ascii code of first char is " << ascii << endl;
}
int firstchar(char *s)  {
    return(int(s[0]));
}
```

produces these compiler error messages

```
type.C: In function `int main()':
type.C:6: argument passing to `char *' from `float'
```

Functions with no arguments in C and C++

In C, a function prototype with an empty argument list

```
extern int freak();
```

indicates that the argument list of the declared function is not prototyped and could have <u>any number of arguments</u>.

In C, a function prototype with a void argument list

```
extern int freak(void)
```

indicates that the function has <u>no arguments</u>.

Because C++ maintains such strict argument checking, an empty argument list means there are literally no arguments. So in C++, the two above *declarations are equivalent*.

For example, consider the freak function declared in its own file freack.C:

```
int freak(int a) {
    return(a+5);
}
```

and a main program in a file called tstfrk.C that use the freak function,

```
#include <stdio.h>
extern int freak();
main() {
  int x=23;
  printf("%d\n",freak(x));
}
```

When compiled with the following command (for example)

```
gcc tstfrk.C freak.C
```

the following compile-time error messages are produced:

```
tstfrk.C: In function `int main()':
tstfrk.C:2: too many arguments to function `int freak()'
tstfrk.C:5: at this point in file
```

In C, the code in these two file would have run successfully.

Type-safe Linkage

C++, takes type checking to further level beyond ANSI C, by comparing function prototyped in the "main" file and actually defined in a separate file.

Consider this function colum defined in a file called colum.c

```
double colum(double a, double b) {
    return(a/2.0*(b+42.0));
}
```

and further consider the main source file (tstcol.c) which uses the colum function:

```
#include <stdio.h>
extern double colum(int a, double b);
main() {
  double a=5.5,b=6.7,c;
  c=colum(a,b);
  printf("%f\n",c);
}
```

In tstcol.c the function prototype incorrectly declares the first dummy argument to be an int instead of a double. <u>A C compiler will not find this error and incorrect value for c will be produced</u>.

NOTE: C programmers who get in the habit of putting function prototypes in their own include file can generally avoid this error.

If this same code is compiled using a <u>C++ compiler it will find the error</u> and prevent the creation of an incorrect executable. This is accomplished at link-time through a process known a **name-mangling**. The function names actually used by the linker are encoded with argument-typing information.

For example, using the code on the previous page, the C++ compile command

```
gcc tstcol.C colum.C
```

produces the following compiling system messages:

```
tstcol.C: In function `int main()':
tstcol.C:5: warning: `double' used for argument 1 of `colum(int, double)'
Undefined                     first referenced
 symbol                             in file
colum__Fid                   /var/tmp/cca001cg1.o
ld: fatal: Symbol referencing errors. No output written to a.out
```

Default Arguments

In C++, the programmer can set **default values for dummy arguments** in function definitions. If no actual value is passed to that dummy argument, the default value is used. Useful when a certain argument almost always have the same value when the function is use. In addition, this approach can reduce the number of actual arguments in the function reference. On the other hand, should comment on why/how the default value is used.

As illustration, consider the following program and its output:

```
#include <iostream.h>
void symm(float x, int k=13) {
    cout << "Argument 1 is " << x << endl;
    cout << "Argument 2 is " << k << endl;
}
main() {
  symm(3.2,167);
  symm(5.4);
}
-----------------------
Argument 1 is 3.2
Argument 2 is 167
Argument 1 is 5.4
Argument 2 is 13
```

Function Overloading

Often, a function needs to perform the same operation, but using arguments of different types. In C++, multiple versions of a function with the **same name** but different type arguments can be defined. In the main program, the "correct" version of the function is actually used by the compiler by examining the argument types. (Internally, this is done through name mangling again...).Function overloading avoids having multiple functions with slightly different baroque names which essentially perform the same work.

In the following program, a swap function is overloaded to work with both integers and doubles.

```
#include <iostream.h>
void swap(int* i, int* j) {
    int temp;
    cout << "int swap called" << endl;
    temp=*i;
    *i=*j;
    *j=temp;
}

void swap(double* x, double* y) {
    double temp;
    cout << "double swap called" << endl;
    temp=*x;
    *x=*y;
    *y=temp;
}
```

```
main() {
  int a=5,b=23;
  double r=34.5,s=1245.78;
  swap(&a,&b);
  swap(&r,&s);
  cout << "a is " << a << "  b is " << b << endl;
  cout << "r is " << r << "  s is " << s << endl;
}
----------------------------------
int swap called
double swap called
a is 23  b is 5
r is 1245.78  s is 34.5
```

Inlined Functions

In C++, functions can be declared as inlined. Inlining is a general optimization technique which causes any reference to a function to be **replaced by the actual code** that makes up the function. The function body in put *in* the *line* where the function reference occurs.The user programs as they wish and the compiler does the inlining automatically.(Especially useful for small functions referenced often).

The advantages to inlining are:

-Don't pay the overhead time-cost of the function call itself

-Unlike macros (the alternative approach), inlined functions can be prototyped and thereby type-checked.

-Less prone to errors than macros.

The disadvantage of inlining (and macros) is the assembly language "text bloat" which will occur. In addition, the actual code must be known to the compiler, thus functions in a run-time library cannot be inlined.

The following C code shows <u>macros can produce incorrect results</u> due to their literal substitution of arguments:

```
#include <stdio.h>
#define MUL(a,b) a*b
main () {
  int x,y,z;
  x=10; y=100;
  z=MUL(x*3+2,y/4);
  printf("z is %d \n",z);
}
-------------------------
z is 80
```

The correct result we expected -(10*3+2) *(100/4) =800 was not calculated because the macro substitution resulted in the expression

x*3+2*y/4

which due to operator precedence gives the value 80.

This problem is solved in the following C++ program using inlining:

```
#include <iostream.h>
inline int mul(int a,int b) {
      return a*b;
     }
main () {
 int x,y,z;
 x=10; y=100;
 z=mul(x*3+2,y/4);
 cout << z << endl;
}
-------------------
800
```

Call-by-Reference

In C, the only way actual arguments are passed to dummy arguments when using a function is **call-by-value**. The value of the actual argument is passed into the dummy argument. Any changes the function makes to the dummy arguments will not affect the actual arguments.

C does provide an indirect method for changing the value of the actual arguments, namely **call-by-address**. Addresses are passed from the actual to the dummy argument and the indirection operator * is used to change the contents of the actual argument.

The following C code demonstrates the use of call-by-address

```
#include <stdio.h>
struct item {
  char* name;
  float price;
  int   quantity;
};
void increase(struct item* p) {
     (*p).price=1.10*(*p).price;
     p->quantity=p->quantity+10;
}
main() {
   struct item chicken={"Holly Farms",5.00,20};
   increase(&chicken);
   printf("New price is %0.2f\n",chicken.price);
}
----------------------------
New price is 5.50
```

The drawbacks to the C call-by-address approach is that the function body is difficult to interpret because of the necessary * and -> operators. In addition, one must remember to use the address operator & for the actual argument in the function reference itself.

For these reasons, C++ has developed a true **call-by-reference**. If a dummy argument is declared as a reference to the actual argument- by using the reference operator & (again!) - it picks up the address of the actual argument (automatically) and can change the contents of that address. *Conceptually, the referenced actual argument just gets a new name in the function: whatever is done to the dummy argument is also done to the actual argument.*

The C++ version of the previous code using call-by-reference is:

```
#include <iostream.h>
struct item {
 char* name;
 float price;
 int   quantity;          read as "reference to an item structure"
};
void increase(item& thing) {
     thing.price=1.10*thing.price;
     thing.quantity=thing.quantity+10;
}
main() {
  item chicken={"Holly Farms",5.00,20};
  increase(chicken);
  cout << "New price is " << chicken.price << endl;
}
------------------------------------------------
New price is 5.5
```

Operator Overloading

One of the more powerful features of C++ is that programmers can define the basic C++ **built-in operators** - such as +, *, [], !, for example- to **work with user-defined types**. This capability makes for a sensible and more readable main programs and is similar, in spirit, to function overloading. Of course, the operators continue to work as originally designed in the language definition for basic types: thus the term overloading. Keep in mind also that although you can use operator overloading to define an operator to do anything you want, it is **best to make the new operator definition is some way analogous to the normal use of the operator**.

The syntax for operator overloading looks exactly like the syntax for a function definition except that the function name is replaced by the name

operator**<***operator symbol***>**

A sample operator overload prototype might look as follows:

```
clock operator+(const clock& a, const clock& b);
```

where clock is some user-defined structure and the body of this prototype will define what "+" will do to two report structures (perhaps, add the hours members to correct for different time zones). Note the use of the const keyword in operator overload prototype. This is to insure that the operator cannot change the value of its operands, only produce a result. Note also the use of call-by- reference arguments. Both noted procedures are suggested, not required.

It should be pointed out that operator overloading **only defines what the operators does for user defined types**. It does not change the operator's precedence, direction of associativity, or number of operands. In addition, you cannot make up your own new operator symbols.

The following program on the next page overloads the *,+, and ! operators to work with complex number in the "natural" way:

```
#include <iostream.h>
struct complex {
  double real;
  double img; };
complex operator+(const complex& a, const complex& b) {
        complex res;
        res.real=a.real+b.real;
        res.img=a.img+b.img;
        return(res); }
complex operator*(const complex& a, const complex& b) {
        complex res;
        res.real=a.real*b.real-a.img*b.img;
        res.img=a.real*b.img+a.img*b.real;
        return(res); }
complex operator!(const complex& a) {
        complex res;
        res.real=a.real;
        res.img=-a.img;
        return(res); }
main() {
  static complex x={1.0,2.0},y={3.0,4.0},z;
  z=x+y;
  cout << z.real << " +i " << z.img << endl;
  z=x*y;
  cout << z.real << " +i " << z.img << endl;
  z=x*(!x);
  cout << z.real << " +i " << z.img << endl;
}
--------------------------------
4 +i 6
-5 +i 10
5 +i 0
```

OOP C++ Features: Table of Contents

C++ Classes

A class is a user-defined data type. You can think of it as an extended and improved structure data type. **An object is variable of a certain class type**. It is often said that an object is an instantiation of the class.

Intrinsic data-types, float, for example have values they can contain (real numbers) and a set of operations (+,/,*,etc) that can be applied to variables of type float.

These same concepts are carried over to class types. The values a class can contain are called its **data members**. The set of operations that can be applied to a certain class object are called its **member functions**.

After a class has been declared, we will see in the rest of the course how its objects can be treated in the same manner and syntax as "normal" data-type variables.

In the following code, a class Triangle is declared:

```
class Triangle {
   double base;                    data members
   double height;
 public:
   void set(double a, double b) {            function members
     base=a; height=b;
   }
   void display()  {
    cout << "base=" << base << " height=" << height << endl;
   }
   double area() { return(0.5*base*height); }
};        common error: forget this semicolon!
```

In this declaration of the class Triangle (it is customary the capitalize the first letter of a class type), there are two data members- base and height- and three member functions: set, display and area. The member functions represent three typical tasks one might want to perform with a Triangle object.In many ways the declaration of a class look like the declaration of a structure but with functions now being allowed as members

Here is an example of some main code that would use the Triangle class data type:

```
main () {
  Triangle t;          t is an object of class Triangle
  t.set(2.0,3.0);
  t.display();
  cout << "area is " << t.area() << endl;
}
-----------------------------------------
base=2 height=3
area is 3
```

It should be noted in this code that the same "dot" operator to access a member of a structure is also used to access the member of class. Specifically, **member functions are invoked with the traditional "dot" operator.**

Access to Class Members

The user has complete control over which parts of a program can alter and/or use the members (data and functions) of a class. There are three types of member access allowed:

Public -> Accessible to all functions in the program (everyone)

Private -> Accessible only to member functions of the same class

Protected -> Accessible only to functions of same *or* derived classes

<u>The default access classification for classes is private</u>. As in the class Triangle previously declared, once a access-specifier -like public- appears, all members following the public keyword are declared public, until a different access-specifier is typed in. Thus, all three member functions of Triangle are public.

Controlling the access to class members facilitates the safe reuse of classes in different programs, and avoids improper addressing and name collisions of data. The practice of using public member functions to indirectly set, change, and use private data members is called **Data Hiding**.

Consider the following class Access:

```
class Access {
  int x;
 public:
  float y;
  int z;
  void set(int a, float b, int c) {
   x=a; y=b; z=c; }
  void display() {
   cout << x << " " << y << " " << z << endl;   }
};
```

which contains a mixture of public and private data members. The proper and improper accessing of these members is illustrated in the following main code:

```
main () {
  Access w;
  Access *p;
  p=&w;
  w.set(1,2.45,3);
  w.display();
  w.y=4.343434;
  w.display();
// w.x=6;     ILLEGAL! Private member
  p->z=32;
  w.display();
}
-------------------------------------
1 2.45 3
1 4.34343 3
1 4.34343 32
```

Member Functions

As we have already seen, and interesting feature of C++ classes (unlike C structures) is that they can contain functions as members. In the class examples we have shown so far the member functions have been defined within the declaration of the class itself. The actual requirement is less severe: **member functions must be declared within the class declaration**, that is, their prototype must appear there. They **can actually be defined outside the class declaration**. One advantage of defining the member function inside the class is that it will automatically be inlined.

In terms of access, member functions - wherever they are defined- have access to all private, protected, and public class members (data & function). When the member function is defined outside the class definition the **scope resolution operator** - **::** - must be used with the class name to indicate that the declared function belongs to that class.

In the following class declaration the member function setx is defined in the declaration of class Play, while the member functions showx & incx are only declared there:

```
class Play {
    int x;
  public:
    void setx (int a) {
      x=a;
    }
    void showx ();
    void incx (int del);
};
```

The following program uses the Play class and shows the syntax for defining "outside" member functions:

```
#include <iostream.h>
class Play {
   int x;
  public:
   void setx (int a) {
     x=a;
   }
   void showx ();
   void incx (int del);
};
void Play::showx() {
      cout << "x is " << x << endl; }
inline void Play::incx(int del) {
             x += del;
}
main () {
  Play fun;
  fun.setx(5);
  fun.showx();
  fun.incx(4);
  fun.showx();
}
---------------------
x is 5
x is 9
```

The Constructor Function

Consider what happens when the following code is executed:

double x=4.567989;

The basic-type variable x is brought into scope, enough memory for a double-type is allocated to x and that memory is initialized with a value. Given that an underlying philosophy of C++ is that derived-types can and should be treated the same (conceptually and syntactically) as intrinsic types, one would expect a similar initialization sequence for class variables. This "construction" of a class variable is accomplished through a **constructor function**.

Here are the basic properties of a constructor

- Special member function
- Has the same name as the class
- Is invoked automatically each time a new variable of its class is created (declared, dynamically-allocated, type conversion to the class)
- Cannot be invoked explicitly
- Has no return type (not even void)
- <u>Typically uses its arguments (if any) to initialize the new objects data members</u>
- Can have default arguments and be overloaded
- Default (no-op) constructor is provided by the compiler if you do not provide one
- **ADVICE:** Always define your own default constructor (and destructor) - even if they are empty- you may need to fill them later...

The following program and its output demonstrate the use and properties of constructors:

```
#include <iostream.h>
class Bookshelf {
    float width; float height; int numshelves;
  public:
    Bookshelf(float i=4.0,float j=6.0,int k=5) {
      width=i; height=j; numshelves=k;
      cout << "Another Bookshelf made: "
      << width << "x" << height << " with " <<
      numshelves << " shelves" << endl; }
};
main () {
  Bookshelf normal;
  Bookshelf wide(8.0);
  Bookshelf tall(4.0,9.0);
  Bookshelf* custom;
  custom=new Bookshelf(6.0,3.0,3);
  for (int i=10; i<=15; i+=5) { Bookshelf paperback(4.0,6.0,i); }
}
-------------------------------------
Another Bookshelf made: 4x6 with 5 shelves
Another Bookshelf made: 8x6 with 5 shelves
Another Bookshelf made: 4x9 with 5 shelves
Another Bookshelf made: 6x3 with 3 shelves
Another Bookshelf made: 4x6 with 10 shelves
Another Bookshelf made: 4x6 with 15 shelves
```

The Destructor Function

The complementary function to the constructor. The **destructor** allows the user to perform any necessary "clean-up" after a class variable is destroyed.

Destructors have the following properties:

- Another special member function
- Has the same name as its class, preceded by a tilde (~)
- Implicitly invoked each time a class object is destroyed (goes out of scope [local variable], dynamically de-allocated, temporary object made during a type conversion no longer needed)
- Has no return type (not even void)
- <u>Has no arguments and cannot be overloaded</u>
- Default (no-op) destructor is provided by compiler (if you don't)
- **Typically used for classes whose member functions perform dynamic memory allocation: destructor releases the memory.**

On the next two pages, two programs and their output are shown. The first program demonstrates how and when constructors and destructors are invoked.

The second program shows how a destructor can *and should* be used to free up no longer needed memory.

Constructor/Destructor Sample Program

```
#include <iostream.h>
class Number {
    int i;
  public:
    Number(int a) {
        i=a;  cout << "Created Number " << i << endl; }
    ~Number()  {
        cout << "Destroyed Number " << i << endl; }
};
Number x(1);
main () {
  Number y(2);
  Number* np;
  np=new Number(3);
  delete np;
  for (int i=4; i<7; ++i) { Number z(i); }
}
---------------------
Created Number 1
Created Number 2
Created Number 3
Destroyed Number 3
Created Number 4
Destroyed Number 4
Created Number 5
Destroyed Number 5
Created Number 6
Destroyed Number 6
Destroyed Number 2
Destroyed Number 1
```

Deallocating Destructor Program

```
#include <iostream.h>
class Release {
   int* p;
  public:
   Release(int a) {
       p=new int; *p=a;
       cout << "Allocated an integer: it is " << *p << endl; }
   ~Release()  {
       delete p;
       cout << "Deallocated an integer: it was " << *p << endl; }
};
main () {
  Release x(1);
  Release* rp;
  rp=new Release(2);
  delete rp;
  for (int i=3; i<5; ++i) { Release z(i); }
}
-------------------------------
Allocated an integer: it is 1
Allocated an integer: it is 2
Deallocated an integer: it was 2
Allocated an integer: it is 3
Deallocated an integer: it was 3
Allocated an integer: it is 4
Deallocated an integer: it was 4
Deallocated an integer: it was 1
```

Copying Objects

Is it possible to copy one object of a class into another object of a class? The answer is yes and the assignment operator "=" has been designed so that the copying is done with "natural" syntax. If you use an assignment statement to copy one object to another a **data member by data member copy** is performed. That is, each member of the object is copied in turn. As in the following program:

```
#include <iostream.h>
class Bookshelf {
  float width; float height; int numshelves;
 public:
  Bookshelf(float i=4.0,float j=6.0,int k=5) {
     width=i; height=j; numshelves=k; }
  display () {
    cout << "Bookshelf is "
    << width << "x" << height << " with " <<
    numshelves << " shelves" << endl; }
};
main () {
 Bookshelf library(6.25,3.50,4);
 Bookshelf bedroom;
 bedroom.display();
 bedroom=library;
 bedroom.display(); }
--------------------------------
Bookshelf is 4x6 with 5 shelves
Bookshelf is 6.25x3.5 with 4 shelves
```

Potential problems with the member-by-member copying action of the assignment statement can arise when there are pointer data members. When pointer are included in the object, the data items that the pointers reference are not actually copied-only the addresses themselves. Because of this, a copied object can inadvertently alter the contents of the original object. Consider the following code:

```
#include <iostream.h>
class Ints {
   int x;
   int* p;
  public:
   void setp(int a) { *p=a; }
   Ints(int a=1) {x=a; p =new int; *p=a; }
   void display() {
     cout << "x=" << x << " *p=" << *p << endl;
   }
};
main () {
  Ints  s(3);
  Ints t;
  s.display();
  t.display();
  t=s;
  t.display();
  t.setp(8);
  t.display();
  s.display();
}
-----------------
x=3 *p=3
x=1 *p=1
x=3 *p=3
x=3 *p=8
x=3 *p=8
```

Contents of s changed when working with copy t!!!

Copy Constructor

The copy constructor is used to make a new class object **safely duplicate** an existing class object. The salient properties of a copy constructor are:

- Special type of constructor member function
- Has exactly one argument which a <u>reference</u> to the class type
- Invoked with the following syntax

  class-name new-object-name(*existing-object-name*)
- Typically defined when the class contains pointer members and the assignment operator is inappropriate
- The copy constructor produces what is referred to as <u>deep copying</u> whereas the assignment operator performs <u>shallow copying</u>

The program on the following page fixes the problem just presented with our Ints class and its shallow copying. The copy constructor shown on the next page actually creates a **new address** for the copied class's int pointer and **not the same address** as the original.

```
#include <iostream.h>
class Ints {
   int x;
   int* p;
  public:
   void setp(int a) { *p=a; }
   Ints(int a=1) {x=a; p =new int; *p=a; }
   Ints(Ints& r) {
     p=new int;
     x=r.x;
     *p=*(r.p);
   }
   void display() {
     cout << "x=" << x << " *p=" << *p << endl;
   }
};
main () {
  Ints s(3);
  s.display();
  Ints t(s);   // replaces t=s;
  t.display();
  t.setp(8);
  t.display();
  s.display();
}
---------------
x=3 *p=3
x=3 *p=3
x=3 *p=8
x=3 *p=3        // s unchanged by actions on t
```

Conversion Constructor

Yet another type of constructor **enables type conversion from other types to the class type**. It is, naturally, called a conversion constructor.The conversion constructor has exactly one argument of a *type other than the class type*. If the compiler encounters a "mixed-type " operation, the conversion constructor is used to make a temporary object of the class type and use this converted variable in the calculations. **Note**: the data members of classes involved in conversion operation often have to made public since they will be used in non-member functions.

Consider the following two classes for English and Metric distances and the conversion constructor to go from Metric units to English units:

```
class Metric {
 public:
  float cm;
  Metric(float x) { cm=x; }
};
class English {
 public:
  float in;
  English(float x) { in=x; }
  English(Metric m) {
   in=m.cm/2.54;
   cout << "Convert construct invoked \n";
  }
  English add(English y) { return(in+y.in); }
  void display() { cout << "Length is " << in << " inches\n"; }
};
```

Here is the main program that demonstrates how the conversion works:

```
main () {
  English d1(6.25);
  English d2(3.375);
  d1=d1.add(d2);
  d1.display();
  Metric m1(30.48);
  d1=d1.add(m1);
  d1.display();
}
-------------------------------
Length is 9.625 inches
Convert construct invoked
Length is 21.625 inches
```

Operator Overloading and C++ Classes

As we have seen previously, C++ offers the ability to overload built-in operators so that they can work with user-defined types. Can operator overloading be applied to C++ classes? Yes: all that is needed is to make the definition function for the operator be a member function of the class.

Consider in the previous program the definition of the function add for the English class which resulted in the somewhat awkward looking "addition" statement:

```
d1=d1.add(d2);
```

Here is the same program rewritten with the "+" operator overloaded to work with English class variables. Notice how much more readable and sensible the main program becomes:

```
#include <iostream.h>
class Metric {
  public:
    float cm;
    Metric(float x) { cm=x; }
};
class English {
  public:
    float in;
    English() { }
    English(float x) { in=x; }
    English(Metric m) { in=m.cm/2.54; }
    English operator+(const English& a) {
            English res;
            res.in=in+a.in;
            return(res); }
    void display() { cout << "Length is " << in << " inches\n"; }
};
```

```
main () {
  English d1(6.25);
  English d2(3.375);
  d1=d1+d2;
  d1.display();
  Metric m1(30.48);
  d1=d1+m1;
  d1.display();
}
----------------------
Length is 9.625 inches
Length is 21.625 inches
```

Scientific Application: Adding Stellar Magnitudes

Since the time of the ancient Greek astronomers, the brightness of a star has been measured by its magnitude m, a real number. The lower the magnitude, the brighter the star: in fact, the brightest stars have negative magnitudes. Stellar magnitudes actually measure on a logarithmic scale in which

$$m = 2.512 \times \log\frac{F_0}{F}$$

where F is the flux from the star and F_0 is the flux from a zero-magnitude star.

Often, astronomers will have two stars in their field of view and would like to know the magnitude of the stars combined. One simply cannot add together the magnitudes of the separate stars since it is only physically correct to add the <u>fluxes</u> from the stars.

In the program on the next page, the "+" operator is overloaded to correctly work for the addition of stellar magnitudes. Again, notice how readable and sensible the statements of the main program are.

```
#include <iostream.h>
#include <math.h>
class Star {
    double mag;
  public:
    Star() { }
    Star(double x) { mag=x; }
    Star operator+(const Star& a) {
      Star res;
      res.mag=2.512*log10(1.0/(pow(10.0,-mag/2.512)+
                                  pow(10.0,-a.mag/2.512)));
      return(res); }
    void display() { cout << "Magnitude is " << mag << endl; }
};
main () {
  Star Deneb(1.26);
  Star Aldebaran(0.86);
  Star Field;
  Field=Deneb+Aldebaran;
  Field.display();
}
--------------------
Magnitude is 0.285582
```

Mathematical Application: Array-bounds overflow

A long-standing and common error made by programmers is to reference an array element that is beyond the bounds of the array. The results are unpredictable depending on what binary pattern in actually located in the (incorrect) memory location that is used. This classic problem is demonstrated in the following C program:

```
#include <stdio.h>
main() {
  int x[20];
  int i;
  for(i=0;i<20;i++)
    x[i]=2*i;
  for(i=0;i<=30;i+=5)
    printf("At index %d: value is %d\n",i,x[i]);
}
------------------------------
At index 0: value is 0
At index 5: value is 10
At index 10: value is 20
At index 15: value is 30
At index 20: value is 5
At index 25: value is -268436436
At index 30: value is 0
```

In C++, one can redefine the array element reference operator "[ ]" to check to make sure that a correct index is used before actually getting an array element. Thus, we see that operators can be overloaded **not only to work normally** with classes and structures of a certain type, but that **their capabilities can be extended** to include safety and error-checking.

Consider the same program on the previous page, but written in terms of C++ classes and operator overloading:

```
#include <iostream.h>
class Intarray {
  int *data,size;
 public:
  Intarray (int sz = 1) {
    if (sz < 1) {
     cout << "Intarray: size of array must be >= 1, not " << sz << endl;
     exit (1);
    }
    size = sz;  data = new int [sz]; }
  }
  int& operator[] (int index) {
   if (index < 0 || index >= size) {
    cout << "\n\n Intarray: boundary overflow or underflow, index="
       << index << ", should range from 0 to " << size-1 << endl;
    exit (1);
    }
    return (*(data+index)); }
};
```

```
main () {
  Intarray x(20);
  for (register int i=0;i<20;i++)
    x[i]=2*i;
  for (register int i=0;i<=30;i+=5)
    cout << "At index " << i << ": value is " << x[i] << endl;
}
-------------------------------------------
At index 0: value is 0
At index 5: value is 10
At index 10: value is 20
At index 15: value is 30
At index 20: value is
```

Intarray: boundary overflow or underflow, index=20, should range from 0 to 19

Conversion operator functions

Recall that a Conversion Constructor can convert a variable to a class type from another type. To convert in the opposite direction: i.e., **from** the class type **to** another type, one has to use a special member operator function. In place of the operator symbol the name of the type to-be-converted to is used.

In the following program we revisit the English to Metric conversion program but this time with English class operator function which converts inches to cm:

```
#include <iostream.h>
class Metric {
 public:
   float cm;
   Metric() {};
   Matriculant x) {cm=x; }
   void display() { cout << "Length is " << cm << " cm\n"; }
   Metric operator+(const Metric& a) {
        Metric res;
        res.cm=cm+a.cm;
        return(res); }
};
class English {
  public:
   float in;
   English(float x) { in=x; }
   operator Metric() {
     Metric m;
     m.cm=in*2.54;
     cout << "Eng-to-Met conversion occurred\n";
     return(m); }
};
```

```
main () {
  Metric m1(30.50);
  Metric m2(2.60);
  m1=m1+m2;
  m1.display();
  English d(3.937);
  m1=m1+d;
  m1.display();
}
-------------------------
Length is 33.1 cm
Eng-to-Met conversion occurred
Length is 43.1 cm
```

Friend Functions

In general only public members of a class are accessible to functions that are not member functions. However, a class can declare a function to be a friend function in which case the **friend function has access to all members of the class** (including private). Friend status is granted *by* the class *to* the function and is a feature of the class design. Friend functions are often used when a calculation needs access to the private members of two objects of the same class.

To specify a friend function, declare the function in the member list of the class, preceded by the **keyword friend**. The function may actually be defined as a global or a member function.

```
#include <iostream.h>
#include <math.h>
class Point {
   double x,y,z;
  public:
   Point(double a, double b, double c) { x=a; y=b; z=c; }
   friend double distance(Point& r, Point& s);
  };
double distance(Point& r, Point& s) {
  return(sqrt(pow(r.x-s.x,2)+pow(r.y-s.y,2)+pow(r.z-s.z,2))); }
main () {
  Point center(1.3,5.6,9.8);
  Point edge(12.4,34.6,56.7);
  cout << "Distance between the points is " << distance(center,edge) <<
endl; }
-----------------------------
Distance between the points is 56.2478
```

Overloading the I/O stream operators

This section will describe how to overload the output stream operator **<<** to work with you own classes. A similar procedure is used for the input stream operator **>>**.

Before overloading **<<** for user-defined classes we need to have a little deeper understanding of how it works. There is a built-in C++ class defined for us called **ostream** (you can see the declaration of this class in the iostream.h include file.) In fact cout, and cerr are just objects of this class. So the following code

```
int i;
cout << i;
```

is telling us that **<<** is a binary operator where the right-hand operand is a basic type (in this case integer) and the left hand operator is an object of class ostream. Therefore, the prototype for our overload function must have the form

```
ostream&     operator<< (ostream&   argl, X   argr);
```

where X is the class name we wish to output in any way we wish. The reason this function returns a reference to an ostream class is to enable " chaining together" of **<<** operations:

```
cout  << X << "Just outputted my own object\n";
```

is executed with a L-R associativity:

```
(cout << X) << "Just outputted my own ojbect \n";
```

Lastly, since one usually wants to output private data member the overload function is typically made a friend of the class.

The code on the next page revisits our Point class and overloads both **<<** and **>>** for it.

```
#include <iostream.h>
class Point {
    double x,y,z;
  public:
    Point() {};
    Point(double a, double b, double c) { x=a; y=b; z=c; }
    friend ostream& operator<< (ostream& os,Point p);
    friend istream& operator>> (istream& is,Point& p);
};
ostream& operator<<(ostream& os, Point p) {
   os << '(' << p.x << ',' << p.y << ',' << p.z << ')' << endl;
   return(os); }
istream& operator>>(istream& is, Point& p) {
   cout << "Enter x coordinate of the point: " ;
   is >> p.x;
   cout << "Enter y coordinate of the point: " ;
   is >> p.y;
   cout << "Enter z coordinate of the point: " ;
   is >> p.z;
   return(is); }
main () {
  Point center;
  cin >> center;
  cout << center;
}
--------------------------------
Enter x coordinate of the point: 3.4
Enter y coordinate of the point: 8.9
Enter z coordinate of the point: 12.3
(3.4,8.9,12.3)
```

Static Data Members

A data member of a class can be declared static. Such a data member is **associated with the class as a whole**, rather than each particular object of the class. A static data member is often referred to as a **"class variable"** and is typically used to store values which characterize the class in some way. For example, a class variable may be used to keep track of the total number of objects that have been created.

As with all class data members and static one must be declared in the definition of the class. This brings the name of the class variable into scope, but <u>does not allocated memory</u> for the member. The allocation is done when the class variable is redeclared (and perhaps initialized) outside of the class definition. Because of this seemingly redundant declaration of the class variable it is actually accessible before the first class object is created.

A class variable can be referred to with two different syntaxes:

-With the class name   class_name::static_data

-With **any** object of the class   class_object.static_data

On the next page we present a graphics program is which various shapes can be drawn with relative to an origin placed anywhere on the screen. The absolute position on the origin is common piece of information needed to all the graphic objects and thus is a natural candidate for a class variable.

```
#include <iostream.h>
class Pixel {
   int x,y;
  public:
   Pixel(int i, int j) { x=i; y=j; }
   static int xo,yo;
   int abs_x() { return(x+xo);}
   int abs_y() { return(y+yo);}
};
int Pixel::xo;
int Pixel::yo;
main () {
  Pixel a(17,42);
  Pixel b(55,75);
  Pixel::xo=5;
  Pixel::yo=6;
  cout << "The absolute coordinates of pixel a are ("
                  << a.abs_x() << "," << a.abs_y() << ")\n";
  cout << "The absolute coordinates of pixel b are ("
                  << b.abs_x() << "," << b.abs_y() << ")\n";
  a.xo=0; a.yo=0;
  cout << "The absolute coordinates of pixel b are ("
                  << b.abs_x() << "," << b.abs_y() << ")\n";
}
--------------------------------------
The absolute coordinates of pixel a are (22,48)
The absolute coordinates of pixel b are (60,81)
The absolute coordinates of pixel b are (55,75)
```

Inheritance

When a new class is declared in C++, you can specify that it should be a derived from an existing class. The new **derived class** inherits all data and function members from the **base class**, as if those class members had been declared explicitly within the new class declaration itself. The derived class modifies or extends the capabilities of the base class. It is through inheritance that C++ enables and facilitates software reuse. The reuse is even possible in cases where classes are made available to the user in the form of object libraries *without* accompanying member source code!

The syntax for making a derived class is to insert a semicolon after the name of the derived class followed by an access method keyword and the name of the base class. For example, the following line declared the derived class Land from the base class Vehicle:

```
class Land: public Vehicle { class-member-list };
```

The Land class inherits all members of Vehicle class except its constructors and destructor.

A member functions in the derived can have the same name as a corresponding member function of the base a class. When this occurs we say the derived class **overrides** (or redefines) the base member function. By default, the overridden function is used when an object of the derived class references it.

Any derived class can serve as a base class for other derived class and thus a multi-layer class hierarchy can be constructed.

The far-from-scientific program on the next page illustrates some of the basic features of derived classes.

```
class Golfer {
  public:
    int rounds_per_mo;
    void swing() { cout << "#&%*@!\n"; }
};
class Serious_Golfer : public Golfer {
  public:
    int handicap;
    void swing() { cout << "Ahhhhh! \n"; }
};
class Pro_Golfer : public Serious_Golfer {
  public:
    float income;
    void swing() { cout << "It's in the hole! \n"; }
};
main () {
 Golfer me;
 Serious_Golfer my_brother;
 Pro_Golfer john_cook;
 me.rounds_per_mo=1; me.swing();
 my_brother.rounds_per_mo= 8; my_brother.handicap=15;  my_brother.swing();
 john_cook.rounds_per_mo=20;  john_cook.income=800000.00;
 john_cook.swing();
 john_cook.Golfer::swing();
}
--------------------------
#&%*@!
Ahhhhh!
It's in the hole!
#&%*@!
```

Public Inheritance

You may have noticed that with all the derived class declarations seen so far the access method public has been used before the name of base class. Public inheritance is by far the most common form of class inheritance and is has the follow properties

- Public Inheritance is an "is-a" relationship. (A Pro_Golfer **is a** Serious _ Golfer also, but more)

- The inherited *public* and *protected* members of the base class become public and protected members of the derived class, respectively

-*Private* members of the base class are inherited but <u>are accessible only to member functions of the base class</u>. If a member function of the derived class needs to directly work with a data member of the base class, make the base class data member protected.

-Because of the "is-a" nature of public inheritance, an object of the derived class may appear unambiguously in any context requiring a base class object

-Pointers or references of the base class type also can point or refer to derived class objects.

The code on the next page demonstrates some of the points of public inheritance.

```
class Vehicle {
   int weight;
  public:
    int getweight() { return(weight); }
    void setweight(int i) { weight=i; } };
class Land : public Vehicle {
   float speed;
  public:
    float getspeed() { return(speed); }
    void setspeed(float x) { speed=x;} };
main () {
    Land car;
//  car.weight=2000;   Illegal! weight is private
    car.setweight(2000); car.setspeed(110.5);
    Vehicle v;
    v=car;
    cout << v.getweight() << endl;
    Vehicle *vp;
    vp=&car;
//  vp->setspeed(65.4);       Illegal! *vp only has Vehicle capabilities
    vp->setweight(1000);
    cout << car.getweight() << endl; }
----------------------
2000
1000
```

Since it can be said that an object of a derived class is at the same time an object of the base class, functions which use the base class will also work when derived-class objects are passed as arguments. The following code demonstrates this feature with out Vehicle-Land class hierarchy:

```
class Vehicle {
  int weight;
 public:
  int getweight() { return(weight); }
  void setweight(int i) { weight=i; } };
class Land : public Vehicle {
  float speed;
 public:
  float getspeed() { return(speed); }
  void setspeed(float x) { speed=x;} };
void check_wt(Vehicle& v) {
   if(v.getweight()>2000) cout << "Your vehicle is too heavy\n"; }
main () {
  Vehicle blob; blob.setweight(100);
  check_wt(blob);
  Land car; car.setweight(5000);
  check_wt(car);
}
---------------------
Your vehicle is too heavy
```

Constructors and Destructors of Derived Classes

When an object of derived class is declared, the constructor for the base class is invoked *first*, then the constructor for the derived class. It is the opposite order for destruction. When a derived-class object goes out of scope, the destructor for the derived class is invoked first, then the destructor for the base class.

Actual arguments to a derived-class constructor are passed in the usual way. To specify actual arguments to the base class constructor, insert a colon after the derived class constructor's formal argument list, followed by the name of base class and a parenthesized list of actual arguments. This technique is demonstrated in the following code:

```
class Widget {
   int wx;
  public:
   Widget(int a) { wx=a; }
   void display() { cout << "wx=" << wx << endl; }
};
class Gidget : public Widget {
   int gx;
  public:
   Gidget(int i, int j) : Widget(i) {gx=j;}
   void display() { cout << "gx=" << gx << endl; }
};
main () {
  Gidget sally(5,33);
  sally.display();
  sally.Widget::display();
}
----------------------
gx=33
wx=5
```

Mathematical Example: Averaging Data

In this C++ code the base class Data is responsible simply for inputting and outputting two arrays of data. The derived class- Average- extends and reuses the Data class. In Average, data is read in and averages of the data are also calculated.

```
class Data {
 protected:
  double *x,*y;
  int N;
 public:
  Data(int i=1) {
   N=i;
   x=new double [N];
   y=new double [N];
  }
  ~Data() { delete [] x; delete [] y; }
  void getdata();
  void showdata();
};
void Data::showdata() {
  for(int i=0;i<N;++i)
   cout << "i=" << i << " x=" << x[i] << " y=" << y[i] << endl;
}
void Data::getdata() {
  for(int i=0;i<N;++i)
   cin >> x[i] >> y[i];
}
```

```
class Average : public Data {
    double mux,muy;
  public:
    Average(int i) : Data(i) {};
    void averagex() {
        double sum=0.0;
        for(int i=0;i<N;++i)
          sum += x[i];
        mux=sum/N;      }
    void averagey() {
        double sum=0.0;
        for(int i=0;i<N;++i)
          sum += y[i];
        muy=sum/N;      }
    void display() {
       cout << "Average x: " << mux << endl;
       cout << "Average y: " << muy << endl;      }
};
main () {
  Average temp(9);
  temp.getdata();
  temp.averagex();
  temp.averagey();
  temp.display();   }
-----------------------
Average x: 5
Average y: 51.8556
```

Virtual Function Terminology

In more traditional programming languages, the act of invoking a function is static, which means that the compiler has sufficient information at compile-time to generate all the code necessary to implement the function call. This is called **early binding** of the function name to the its code.

A principal feature of C++ is that the <u>same function name</u> with the <u>same argument list</u> can perform different actions depending of the class of the object on which its is invoked. This property is referred to as **polymorphism**. In order to implement polymorphism, sometimes you must defer until run-time the association of a specific function name with the code that implements it. When this run-time connection occurs it is referred to as **late binding**. Finally, functions for which late binding is used are called **virtual functions**.

Virtual Functions: Syntax and Use

To declare a member function to be virtual, precede the function declaration with the keyword virtual. Say that a base class has a virtual function and that function is overridden in derived class. When the function name is invoked by an object of the derived class, the derived class version of the function is **always used**, regardless of how the derived-class object is referred to. In other words, when using virtual functions the meaning selected for the function depends on the *class* of the object rather than the way you *refer* to the object.

On the next two pages we have resurrected the Golfer class hierarchy to demonstrate the difference between the actions of non-virtual and virtual overridden functions.

```
#include <iostream.h>
class Golfer {
  public:
    int rounds_per_mo;
    void swing() { cout << "#&%*@!\n"; }
};
class Serious_Golfer : public Golfer {
  public:
    int handicap;
    void swing() { cout << "Ahhhhh! \n"; }
};
class Pro_Golfer : public Serious_Golfer {
  public:
    float income;
    void swing() { cout << "It's in the hole! \n"; }
};
main () {
  Pro_Golfer jc;
  Golfer *gp;
  gp=&jc;
  gp->swing();
}
------------
#&%*@!
```

```
#include <iostream.h>
class Golfer {
  public:
    int rounds_per_mo;
    virtual void swing() { cout << "#&%*@!\n"; }
};
class Serious_Golfer : public Golfer {
  public:
    int handicap;
    void swing() { cout << "Ahhhhh! \n"; }
};
class Pro_Golfer : public Serious_Golfer {
  public:
    float income;
    void swing() { cout << "It's in the hole! \n"; }
};
main () {
  Pro_Golfer jc;
  Golfer *gp;
  gp=&jc;
  gp->swing();
}
----------------
It's in the hole!
```

Pure Virtual Functions

The virtual function Golfer::swing() shown on this previous page is an example of a **simple virtual function** because it is both declared and defined in the base class. A **pure virtual function**, on the other hand, is *only* declared in the base class. The (rather odd!) syntax change for a pure virtual function is to append = 0 to the function declaration:

```
virtual   return-type   func-name(args) = 0;
```

Pure virtual functions in base classes have the following characteristics:

- No objects of the base class can be declared. Base class can only be used to derive other classes.
- **Must** be overridden in each derived class
- A pure virtual function establish a *function interface* inherited by all derived classes
- A class that declares one or more pure virtual functions is termed an **abstract base class**

The program on the next page makes the Golfer::swing function a pure virtual function and thus makes Golfer an abstract base class.

```
#include <iostream.h>
class Golfer {
  public:
    int rounds_per_mo;
    virtual void swing()=0;
};
class Serious_Golfer : public Golfer {
  public:
    int handicap;
    void swing() { cout << "Ahhhhh! \n"; }
};
class Pro_Golfer : public Serious_Golfer {
  public:
    float income;
    void swing() { cout << "It's in the hole! \n"; }
};
main () {
   Pro_Golfer jc;
// Golfer mike; Illegal! can't declare a variable of an abstract base class
   Golfer *gp;
   gp=&jc;
   gp->swing();
}
--------------
It's in the hole!
```