C++ for C Programmers

by JT Kalnay

This book is dedicated to Dennis Ritchie and to Steve Jobs.

To Dennis for giving us the tools to program.

To Steve for giving us a reason to program.

Published by jt Kalnay

Discover other titles by jt Kalnay at:
www.jtkalnay.com

1

About This Book

This book is not organized in a traditional chapter format. Instead I have chosen to include example programs that illustrate the important points of C++ in an evolutionary manner. Where appropriate, I have provided C code that would accomplish the same things that the C++ code would do to illustrate the specific advantage of C++. This comparison is useful as both a teaching tool and a motivational tool.

The programs that I present are not, by themselves, complete applications. The programs are “single-issue teaching programs”. Experienced programmers who are learning a new language have told me time and time again that they mainly want to see the functionality of the new syntactic and semantic elements. The programmers tell me that they will be able to think of the applicability of the feature to their project. When necessary, I provide a sample application to give a feel for how the new element might be employed.

The programs are presented in an order that presents the simplest, most straightforward aspect of a new element first. Subsequent programs present the more subtle or confusing aspects of a new element. This is a proven pedagogical approach for teaching C++ that I have presented to over 1,000 professionals and college students.

This book assumes that you are already a GOOD C programmer and are able to learn well on your own. Simple (and peripheral) C++ concepts such as the cout/cin i/o mechanisms and how they replace printf/scanf ... are left to the end or for the reader. To ease the learning curve, and to focus on the compelling differences between C and C++, many C functions are used in place of the identical corresponding C++ rewrites.

Good luck in your study of C++.

Table Of Contents

What is C++?

C++, at its essential core, is C with additional syntax to implement objects through the mechanism of classes. C++ upgrades the struct keyword with several new features;

data hiding
encapsulation
polymorphism.

The extensions to the struct keyword are significant enough that the keyword "class" has been added to the language to replace and/or augment "struct". A class in C++ is the mechanism which facilitates the writing of Object Oriented programs.

Why you should learn C++.

1) Data hiding and Functional Implementation hiding via private data and publicinterface coding.

Skilled C programmers utilize techniques to implement data hiding. C++ formalizes and thus facilitates this practice.

2) Being able to define operators for your data structures without having to introduce new names into the programmers vocabulary.

In C, you define data structures with the struct keyword. The only operation defined for structs is =, a memberwise copy. If you want to add two instances of a struct, or compare two instances of a struct, you have to write a subroutine to perform the function and you have to advertise the name of the subroutine to the programmer that will be using it. This is silly. You already have the + and == operators to add and test for equivalency. Why can't you use them for user defined types? In C++, you can define the data structure and what any of the standard operators (+ ++ - - = == * / ...) mean with respect to that data structure. Having operators for your data structures simplifies the programmer's job. They don't have to learn as many subroutine names to be able to use your data structure.

3) User defined variable initialization and deallocation code.

In C, when someone declares an instance of one of your data structures, the system executes code that is responsible for assigning initial values to that data structure. The user may specify initial values, however it is still system code that runs.
In C++, you have the option of writing what is known as constructor code. When someone declares an instance of one of your data structures, your constructor code can be executed. This can be extremely useful for error checking, data validation, user validation, auditing, billing, and tracking reasons.

4) Anything you can do in C you can do in C++.
Not everything you can do in C++ can be done in C.

5) Many new commercially available code libraries will only work with C++.

Section 2

This section covers the parts of C++ that are building blocks for classes. These topics can be studied before or after learning about classes. However, to be able to fully exploit the power of classes, an understanding of these topics is very valuable.

ch1p1.cpp	C++ commenting mechanisms C++ and the if then else of C C++ and the for, while, do of C C++ and the printf of C and the cout of C++
vardefs.cpp	C++ and the int, float, char of C C++ and the new possible locations of variable declarations
scope.cpp	C++ and the scope of variables The C++ Scope Resolution Operator (sro) ::
scope1.cpp	The sro extended to include subroutine code
protos.c	C and its traditional weak function prototype header checking
protos.cpp	C++ and tighter function prototype header checking C++ needs tighter prototype checking because it allows for polymorphic functions. Several functions may have the same name, as long as they have different argument lists. The functions are distinguishable by their argument lists and thus the error checking performed by the prototype header checking becomes very important.
protos1.cpp	A polymorphic function
protos2.cpp	Polymorphic functions taken to a ridiculous, yet pedagogically valuable extreme.
reversi.cpp	A programmer friendly usage for polymoprhic functions.
defaults.cpp	C++ allows a function to have default arguments defined for the incoming argument list. This way, if a function is called with less than the required number of arguments in the calling list, the missing arguments can be filled in using the defaults.
polydef.cpp	There can be a side effect when using polymoprhic functions and functions with default argument lists. This program illustrates the potential pitfall.
refs.cpp	C++ and the reference feature. A reference is not a pointer. This program points that out.
refs1.cpp	References in action.
refs2.cpp	Extremely clever thing you can do with references.

```
// ch1p1.cpp
// this program introduces C++ comments, variables, decisions and loops
// the double slashes are an addition to C++
// they are a commenting mechanism
// a double slash comment can begin anywhere on a line and continues
// until the end of that line and that line only

/* the slash asterisk, asterisk slash comment method */
/* is still
	in use */

#if 0
	to comment out long passages
	of code, the #if 0 and the #ifdef mechanisms
	are still available
#endif

#include <stdio.h>			// get access to printf scanf
#include <iostream.h>			// get access to cout and cin

// main is still the program entry point and by default still returns an int
main()
{
	// the { is still the start of scope operator
	// the } is still the end   of scope operator

	int i = 5;				// int is still a keyword
	float f = 7.2;				// float is still a keyword
	char x = 'a';				// char is still a keyword

	// all the decision structures are the same as in C, if, switch, goto
	if ( i < 7 )	// the relational operators are the same,  < <= > >= != ==
	{
		printf("i was less than 7\n");		// if is identical
		printf("i was %i\n",i);			// printf is identical, although often replaced with cout
	}
	else
	{
		cout << "i was greater than or equal to 7\n";		// cout is new
		cout << "i was " << i << endl;				// it can replace printf
	}

	// all the looping structures are the same as in C  for, while, do
	for ( i = 10; i < 13; i++ )
		printf("%i squared is %i \n",i, i*i);

}

```

```
// vardefs.cpp
// This program illustrates that C++ can declare variables the same as C.
// It also illustrates that in C++ you may declare variables anywhere you want.
// This is an extension to variable declaration in C.
// A common C++ programmer trick is illustrated, along with its side-effect.
// The C programmer who is used to declaring all their variables at the
// beginning of a program may continue to do so, however, there are
// advantages to waiting to declare a variable until it is needed, especially
// in programs that will run for hours or days
#include <stdio.h>
main()
{
	int i;				// declare i, don't initialize it
	int j = 5;			// declare and initialize j
	printf("i is %i j is %i \n",i,j);	// show i and j

	int k = 7;			// declare another variable after first executable
	printf("k is %i \n",k);

	for ( i = 0; i < 3; i++ )			// i already exists
	{
		int i = 0;				// in a new scope, a new i is created
		printf("INSIDE LOOP i is %i\n",i);	// this is the inside i
	}					// the inside i goes out of scope and
						// is deallocated each time around loop
	printf("After loop i is %i\n",i);

	// we can create a new variable anywhere, l is local to main
	// l is not local to the loop, the l declared inside the scope is local
	// to the loop

	for ( int l = 0;  l < 3; l++ )			// l is created once, at start of loop
	{
		int l = 0;				// in a new scope, a new l is created
		printf("inside loop l is %i \n",l);	// this is the inside l
	}					// the inside l goes out of scope and
						// is deallocated each time around loop
	printf("After loop l is %i\n",l);

#ifdef BUG
	// this would be a redefinition of l, remember l is local to main
	// l is not local to the loop, many programmers confuse this issue
	for ( int l = 0;  l < 3; l++ )			// l is created once, at start of loop
	{
		int l = 0;				// in a new scope, a new l is created
		printf("inside loop l is %i \n",l);	// this is the inside l
	}					// the inside l goes out of scope and
						// is deallocated each time around loop
	printf("After loop l is %i\n",l);
#endif
}
```

Output From Running Program

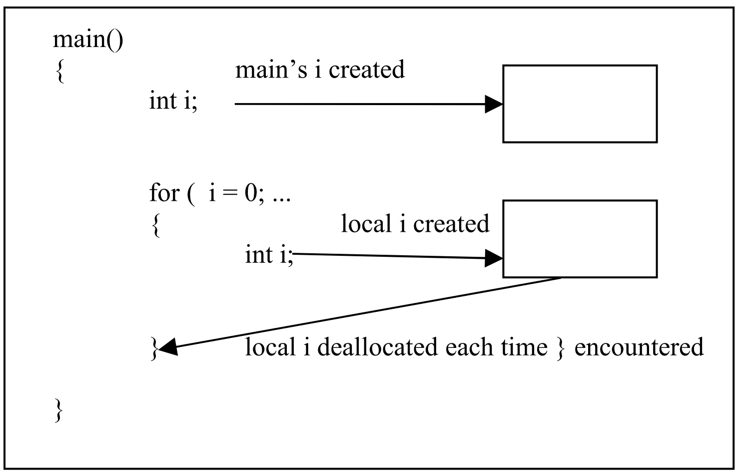
i is 0 j is 5
k is 7

```
INSIDE LOOP i is 0
INSIDE LOOP i is 0
INSIDE LOOP i is 0
After loop i is 3
```

Notice that each time around the loop, the internal I was re-created, and re-initialized to 0

```
inside loop l is 0
inside loop l is 0
inside loop l is 0
After loop l is 3
```

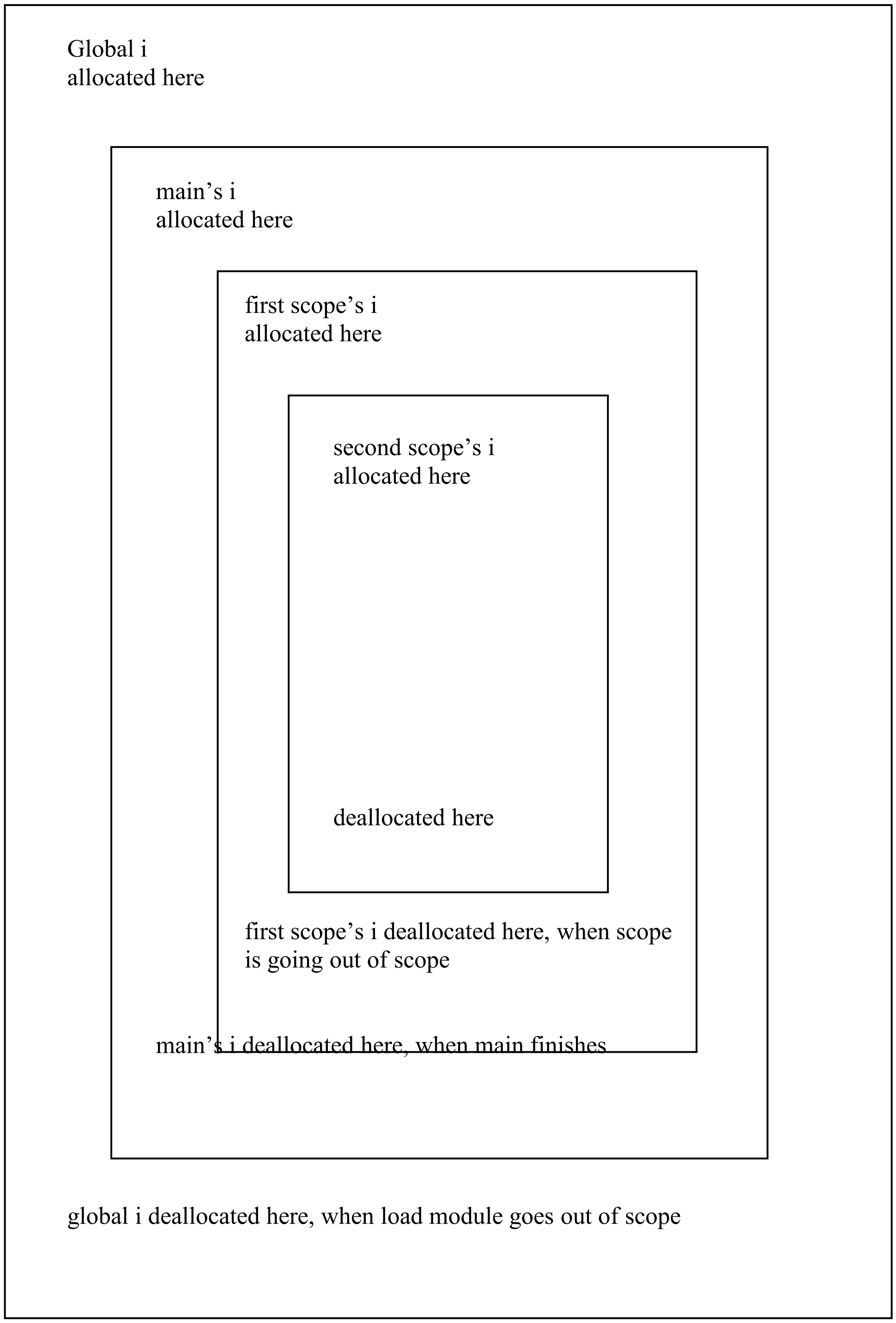
Notice that each time around the loop, the internal l was created, and re-initialized to 0



C++ allows you to declare variables the same way you did in C.
C++ also allows you to declare variables anywhere you want, even after executable statements.

```
// scope.cpp
// C and C++ share variable scoping rules
// C++ provides a new operator, the scope resolution operator ::
// this operator can be used to distinguish between a local variable
// and a global variable by the same name.
#include <stdio.h>

int i = 5;      // i is a global variable, we'll call it global i, we can access it anywhere via ::i
int j = 7;      // j is also a global variable, we'll call it global j, we can access it anywhere via ::j
main()
{
        int i = 50;                  // i is a local variable, we'll call it main's i
        printf("i is %i \n",i);               // the local  variable will be printed
        printf("::i is %i \n",::i);           // the global variable will be printed

        printf("j is %i \n",j);               // the local  variable will be printed
        printf("::j is %i \n",::j);           // the global variable will be printed

        // start a new scope
        {
                int i = 500;                  // this i is local to the scope
                printf("\n\tIN FIRST SCOPE\n");
                printf("\ti is %i \n",i);       // local  var printed
                printf("\t::i is %i \n",::i);   // global var printed
                printf("\tj is %i \n",j);       // local  var printed
                printf("\t::j is %i \n",::j);   // global var printed

                // start another new scope
                {
                        int i = 5000;           // local to scope
                        int j = 7000;           // local to scope
                        printf("\n\t\tIN INNERMOST SCOPE\n");
                        printf("\t\ti is %i \n",i);          // local  printed
                        printf("\t\t::i is %i \n",::i);      // global printed
                        printf("\t\tj is %i \n",j);          // local  printed
                        printf("\t\t::j is %i \n",::j);      // global printed
                }
                printf("\n\tAFTER INNERMOST SCOPE\n");
                printf("\ti is %i \n",i);                    // local  var printed
                printf("\t::i is %i \n",::i);                // global var printed
                printf("\tj is %i \n",j);                    // local  var printed
                printf("\t::j is %i \n",::j);                // global var printed
        }
        printf("\n\tAFTER FIRST SCOPE\n");
        printf("i is %i \n",i);               // local  var printed
        printf("::i is %i \n",::i);           // global var printed
        printf("j is %i \n",j);               // local  var printed
        printf("::j is %i \n",::j);           // global var printed
}
```

```
Output From Running Program
                                        At this point in the program there is the global i and
                                        there is main's i
i is 50                                 i all by itself refers to the i local to the scope
::i is 5                                ::i refers to the GLOBAL i
j is 7
::j is 7

        IN FIRST SCOPE                  now there are threei's, therefore i and ::i are different
        i is 500                        i all by itself refers to the i local to the scope
        ::i is 5                        ::i refers to the GLOBAL i
        j is 7
        ::j is 7

               IN INNERMOST SCOPE       now there are four i's, therefore i and ::i are different
               i is 5000                i all by itself refers to the i local to that scope
               ::i is 5                 ::i refers to the GLOBAL i, not the i one scope out
               j is 7000
               ::j is 7

        AFTER INNERMOST SCOPE           now we are back to just three i's
        i is 500                        i refers to the i local to that scope
        ::i is 5                        ::i refers to the GLOBAL i
        j is 7
        ::j is 7

        AFTER FIRST SCOPE
i is 50
::i is 5
j is 7
::j is 7
```

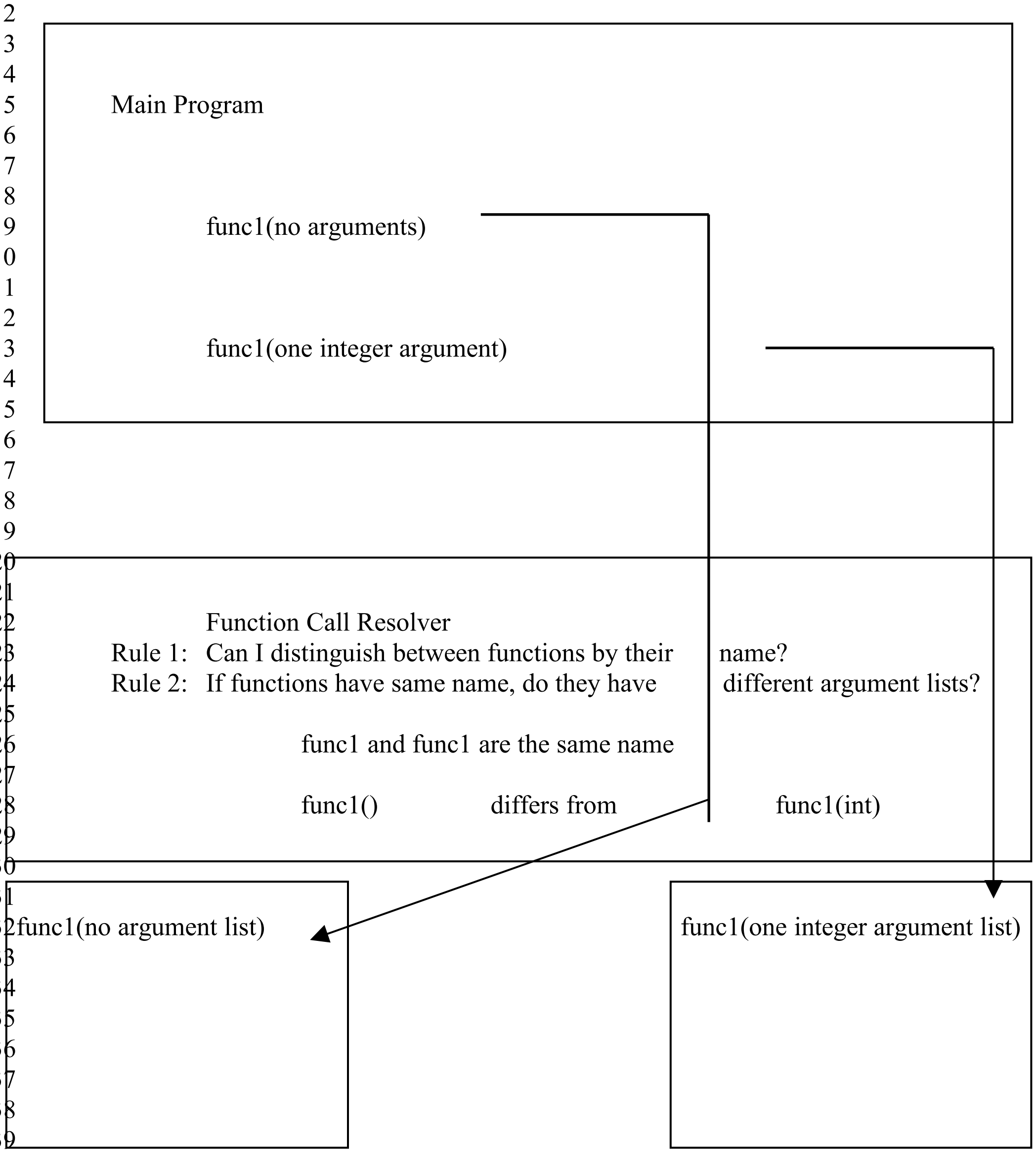

Global i
allocated here
main’s i
allocated here
first scope’s i
allocated here
second scope’s i
allocated here
deallocated here
first scope’s i deallocated here, when scope
is going out of scope
main’s i deallocated here, when main finishes
global i deallocated here, when load module goes out of scope

```
// scope1.cpp
// the scope resolution operator applies in subroutines as well
// a subroutine may have a local instance of a variable and still reference a global variable with
// the same name via the :: operator

#include <stdio.h>
#include <iostream.h>

int i = 5;              // this is a global variable
int j = 7;              // this is also a global variable

void func1(void)
{
        // declare a local i
        // do not declare a local j
        int i = 25;
        cout << "IN func1   i is " << i << endl;          // i, all by itself, referes to the local i
        cout << "IN func1 ::i is " << ::i << endl;        // ::i referes to the GLOBAL i, not main's i

        cout << "IN func1   j is " << j << endl;          // j, all by itself, referes to the global j in this case
        cout << "IN func1 ::j is " << ::j << endl;        // because there is no local j
        return;
}

main()
{
        int i = 50;             // this i is local to main, it is not visible to the subroutine
        printf("i is %i \n",i);         // the local  variable will be printed
        printf("::i is %i \n",::i);     // the global variable will be printed

        printf("j is %i \n",j);         // the local  variable will be printed
        printf("::j is %i \n",::j);     // the global variable will be printed

        // call the function
        func1();
}

Output From Running Program

IN func1   i is 25              // the local i in the subroutine was accessed
IN func1 ::i is 5               // the GLOBAL i, not main's i, was accessed via ::i
IN func1   j is 7               // there was no local j, therefore the GLOBAL j was accessed
IN func1 ::j is 7               // ::j also referred to the global j

i is 50                         // the local i in main was accessed
::i is 5                        // the GLOBAL i was accessed
j is 7                          // the global j was accessed because there is no j local to main
::j is 7                        // the global j was accessed
```

```
1// protos.c
2// C++ tightens the requirement for function prototype headers
3// In C, you can get away without having the header, sometimes with
4// disastrous results.  In C++ you have to have the header.
5// This program could be compiled as a C program
6// This program could not be compiled as a C++ program because of missing prototype headers
7main()
8{
9       printf("IN main\n");
10      func1();
11}
12
13int func1(void)
14{
15      printf("IN Func1\n");
16      return 1;
17}
18
19Output From Running Program as a C Program
20IN main
21IN Func1
22
23Sample Output of Trying To Compile as a C++ Program
24CC protos.c
25"protos.c"        line 13:  error:     undefined function func1 called
26Compilation failed
```

```
// protos.cpp
// C++ tightens the requirement for function prototype headers
// In C, you can get away without having the header, sometimes with
// disastrous results.  In C++ you have to have the header.
// This program could be compiled as a C++ program

#include <stdio.h>          // these two lines are required to compile this
int func1(void);            // program as a C++ program
                            / without them you would get errors on the use of
                            // printf and on the use of func1
                            // C++ REQUIRES the function prototype headers

main()
{
        printf("IN main\n");
        func1();
}

int func1(void)
{
        printf("IN Func1\n");
        return 1;
}


Output From Running  Program
IN main
IN Func1
```

```
// protos1.cpp
// C++ tightens the requirement for function prototype headers
// The reason it tightens the function prototype header rules
// is related to the feature called polymorphism that C++ provides
// examine the TWO FUNCTIONS func1
// they have the same name, but take different argument lists
// we will examine this closer in later programs

#include <stdio.h>      // these two lines are required to compile this
int func1(void);
int func1(int);

main()
{
        int i = 0;

        printf("IN main\n");
        printf("Before call i is %i\n\n",i);
        i = func1();
        printf("After first call i is %i\n\n",i);
        i = func1(i);
        printf("After second call i is %i\n\n",i);
}


function name:    func1                          function name is func1    called via func1()
return type:      int
argument list:    void
int func1(void)
{
        printf("IN func1, no argument version \n");
        return 1;
}

function name:    func1                          function name is func1    called via func1(int)
return type:      int
argument list:    int
int func1(int)
{
        printf("IN func1, integer argument version \n");
        return 253;
}

Output From Running Program

IN main
Before call i is 0

IN func1, no argument version
After first call i is 1

IN func1, integer argument version
After second call i is 253
```


Main Program
func1(no arguments)
func1(one integer argument)
Function Call Resolver
Rule 1: Can I distinguish between functions by their name?
Rule 2: If functions have same name, do they have different argument lists?
func1 and func1 are the same name
func1() differs from func1(int)
func1(no argument list)
func1(one integer argument list)

```
// protos2.cpp
// a function may have the same name as another function as long as it differs in the argument list
// this program takes this to the extreme

#include <stdio.h>          // these two lines are required to compile this
#include <iostream.h>

int func1(void);            // prototype header for func1 with no argument list
int func1(int);             // prototype header for func1 with one integer in argument list
int func1(float);           // prototype header for func1 with one float in argument list
int func1(int,int);         // prototype header for func1 with two ints in argument list
int func1(int,int,int);     // prototype header for func1 with three ints in argument list
int func1(int,int,int,int); // prototype header for func1 with four ints in argument list


main()
{
        int i = 0;
        float f = 1.1;

        printf("IN main\n");
        printf("Before call i is %i\n\n",i);
        i = func1();
        printf("After no arg    call i is %i\n\n",i);

        i = func1(i);
        printf("After one int   call i is %i\n\n",i);

        i = func1(f);
        printf("After float     call i is %i\n\n",i);

        i = func1(i,i);
        printf("After two int   call i is %i\n\n",i);

        i = func1(i,i,i);
        printf("After three int call i is %i\n\n",i);

        i = func1(i,i,i,i);
        printf("After four int  call i is %i\n\n",i);
}
```

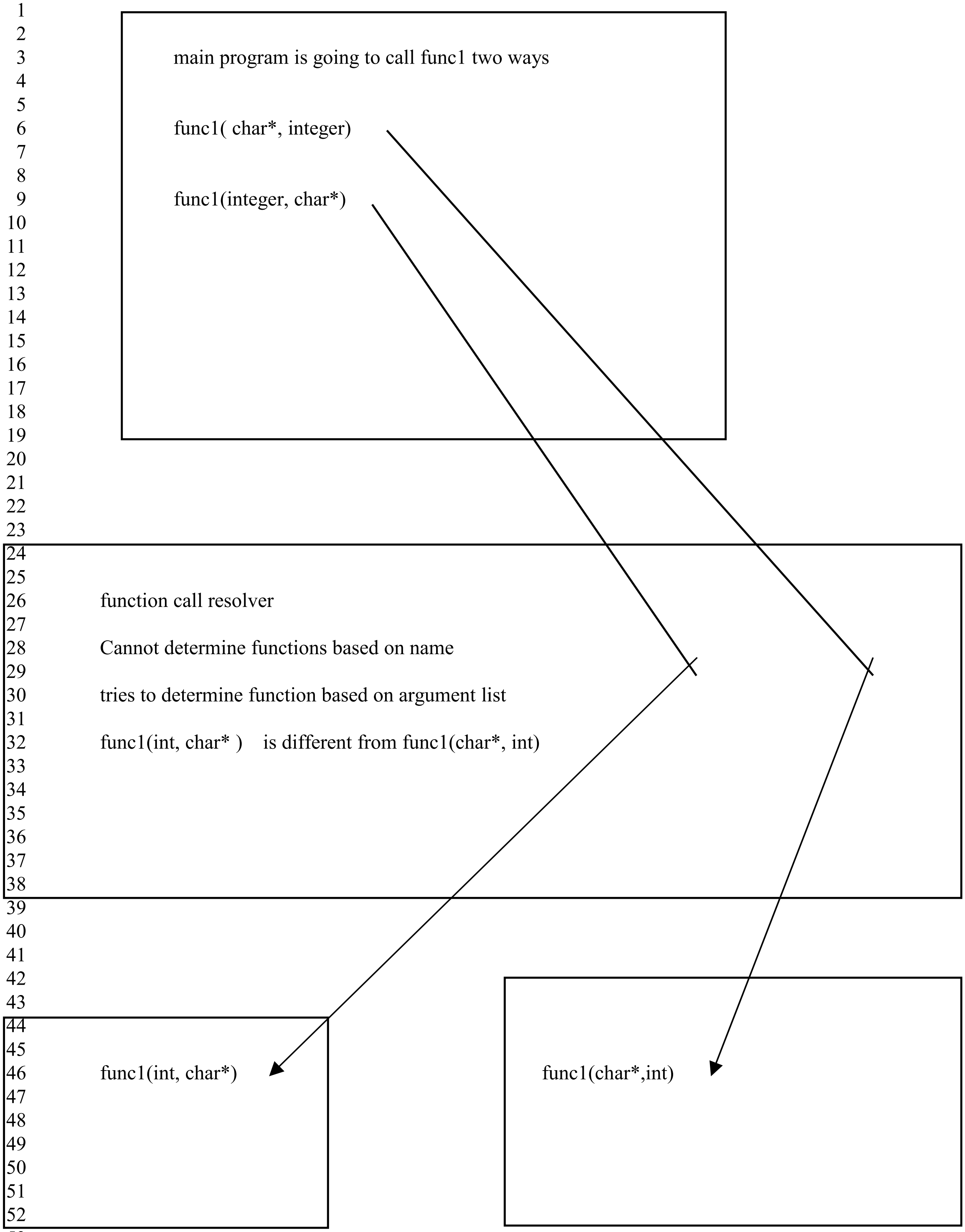
```
// The ridiculous list of functions all with the same name
// Having one function with one name is good, having two functions with the same name is okay
// Having three functions with the same name is acceptable, but FOUR functions or more with the same
// name?  Are they really all doing the same thing?  Maybe you need a different function name.  Maybe not

int func1(void) {
	printf("IN func1, no argument version \n");
	return 1; }

int func1(int) {
	printf("IN func1, integer argument version \n");
	return 2; }

int func1(float) {
	printf("IN func1, float argument version \n");
	return 3;  }

int func1(int,int) {
	printf("IN func1, two integer argument version \n");
	return 4; }

int func1(int,int,int) {
	printf("IN func1, three integer argument version \n");
	return 5; }

int func1(int,int,int,int) {
	cout << "In func1, four integer argument version " << endl;
	return 6; }
```

Output From Running Program

```
In func1, four integer argument version
IN main
Before call i is 0

IN func1, no argument version
After no arg    call i is 1

IN func1, integer argument version
After one int   call i is 2

IN func1, float argument version
After float     call i is 3

IN func1, two integer argument version
After two int   call i is 4

IN func1, three integer argument version
After three int call i is 5
After four int  call i is 6
```

The function call resolver can tell all these functions apart by their argument lists

```
// reversi.cpp
// this program shows a handy feature of C++ polymorphic functions
// When you have a function that takes two arguments of two different
// types and a lot of people are going to use that function
// you can make the function easier to use by allowing the programmer
// to call the function with the arguments in either order

// For example, if you have a function that requires an age and a name
// you could allow the function to be called with the name first and age
// second or the age first and the name second.

void func1(char *, int);            // function prototype header for name first, age second
void func1(int, char* );            // function prototype header for age first, name second

#include <iostream.h>
main()
{
        // call the function the first way
        func1("bill", 32);

        // call the function the second way
        func1(89,"frank");

}

function name:    func1
return type:      void
argument list:    pointer to character, integer
void func1(char * ptr, int age)
{
        cout << ptr << " is " << age << " years old " << endl;
}

function name:    func1
return type:      void
argument list:    integer, pointer to character
void func1(int age, char * ptr)
{
        cout << ptr << " is " << age << " years old " << endl;
}


// MAINTENANCE POINT
// you have to decide whether to use polymorphic names for a function or
// whether to use default arguments for a function
// you create a chicken and egg problem if you use both polymorphic names
// for a function and default arguments for a function, see the next program

Output From Running Program
bill is 32 years old
frank is 89 years old
```


main program is going to call func1 two ways
func1(char*, integer)
func1(integer, char*)
function call resolver
Cannot determine functions based on name
tries to determine function based on argument list
func1(int, char*) is different from func1(char*, int)
func1(int, char*)
func1(char*,int)

```
// defaults.cpp
// C++ allows you to write functions that have default arguments.
// This is handy when you want to call a function that needs four
// arguments but two or three of them are almost always the same.
// Any arguments that you do supply to a function with default arguments
// will fill in the argument list fields from left to right.
// Imagine a function that requires the date passed in, with default
// arguments, the function could assume that we are in the 20th century unless
// told otherwise.

// what you do is specify what the arguments are supposed to be if no value is provided.

void func1(int = 1);                          // if called with no args, arg set to 1
void func2(int, int = 2);                     // if called with no args, error
                                              // if called with one arg, second arg set to 2

void func3(int, int, int = 27);               // if called with no args, error
                                              // if called with one arg, error
                                              // if called with two args, third arg set to 27

void func4(int = 1, int = 2, int = 3, int = 4);   // if called with no args,
                                              // first arg set to 1, second arg set to 2
                                              // third arg set to 3, foruth arg set to 4
#include <iostream.h>
main()
{
    int i = 100, j = 200, k = 300, l = 400;

        // what will be printed by each of these calls??
        func1();
        func1(i);
        func1(j);
        cout << endl;

        // func2();  THIS WOULD BE A COMPILER ERROR.  Can you answer why?
        func2(i);
        func2(i,j);
        cout << endl;

        // BOTH OF THESE WOULD BE COMPILER ERRORS.  Can you answer why?
        // func3();
        // func3(i);
        func3(i,j);
        func3(i,j,k);
        cout << endl;

        func4();
        func4(i);
        func4(i,j);
        func4(i,j,k);
        func4(i,j,k,l);
        cout << endl;
}
```

```
void func1( int a )
{ cout << "a is " << a << endl; }

void func2( int a, int b )
{ cout << "a is " << a << "   b is " << b << endl; }

void func3( int a, int b, int c )
{ cout << "a is " << a << "   b is " << b << "   c is " << c << endl; }

void func4( int a, int b, int c, int d )
{ cout << "a is " << a << "   b is " << b << "   c is " << c << "   d is " << d << endl; }
```

In the program remember that these variables had these values

i = 100
j = 200
k = 300
l = 40

Output From Running Program

Output Produced	**Function Call That Was Made**
a is 1	func1()
a is 100	func1(i)
a is 200	func1(j)
a is 100 b is 2	func2(i)
a is 100 b is 200	func2(i,j)
a is 100 b is 200 c is 27	func3(i,j)
a is 100 b is 200 c is 300	func3(i,j,k)
a is 1 b is 2 c is 3 d is 4	func4()
a is 100 b is 2 c is 3 d is 4	func4(i)
a is 100 b is 200 c is 3 d is 4	func4(i,j)
a is 100 b is 200 c is 300 d is 4	func4(i,j,k)
a is 100 b is 200 c is 300 d is 400	func4(i,j,k,l)

main calls func1, can call it with one int or with not ints

main calls func2, can call it with one int or two ints
cannot call it with no ints

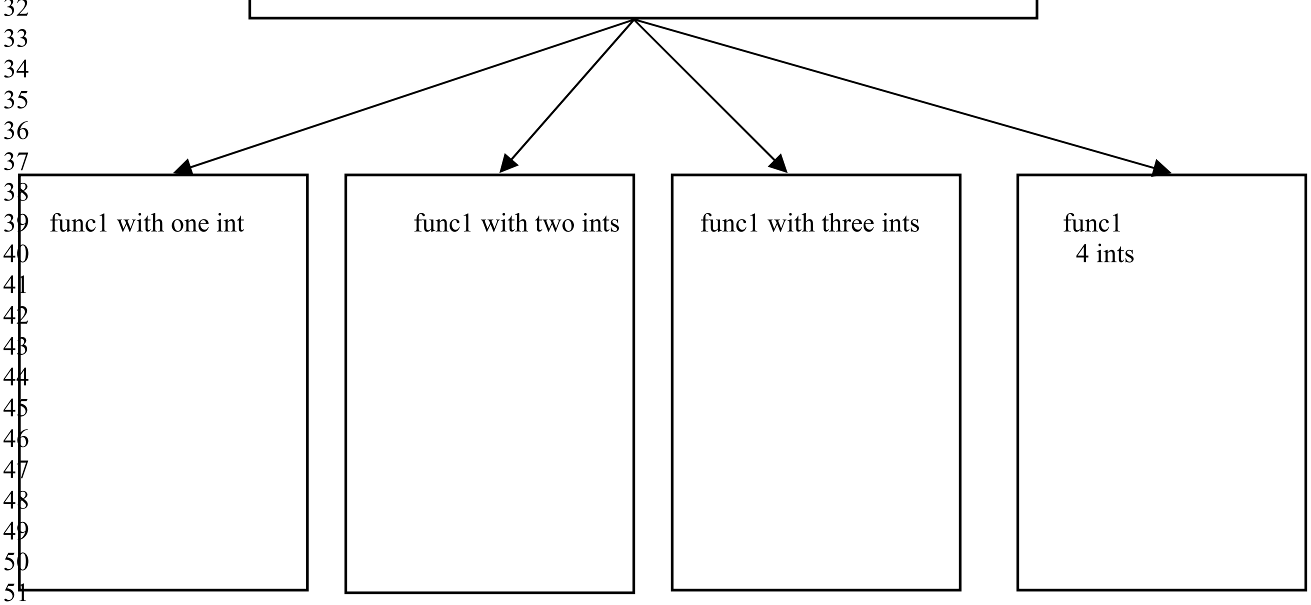
main calls func3 with two ints or three ints
cannot call it with no or one ints

main calls func4 with zero, one, two three or four ints

Function Call Resolver

Can tell functions apart by their names

But what does it do when it gets called with func1 and no ints?
it fills in the one default and calls func1 with one int!
What does it do when it gets called with func4 and two ints?
it fills in two more ints and calls func4 with four ints!



```
1// polydef.cpp
2
3// this program illustrates a potential problem using polymorphic names
4// for a function where you also supply default arguments for a function
5
6void func1(int, float = 2.34 );          // this could be called sending one int
7void func1(int);                         // this could also be called sending one int
8                                         // There would be no way for the function call resolver to figure
9                                         // out which one had been called
10
11#include <iostream.h>
12main()
13{
14      int i = 5;
15      float f = 9.99;
16
17      // call func1 sending one int, which gets called, the
18      // func1 with a single int arg or the func1 with an int
19      // and float default?
20      func1(i);
21
22      // call func1 sending an int and a float
23      func1(i,f);
24
25}
26
27function name:    func1
28return type:      void
29argument list:    one integer, no defaults
30void func1(int a)
31{
32      cout << "In one int arg func1 a = " << a << endl;
33}
34
35function name:    func1
36return type:      void
37argument list:    one integer and one float   OR
38                  one integer, with float added in as default by function call resolver
39void func1(int a, float b)
40{
41      cout << "In two arg func1 with float default " << endl;
42      cout << "int a is " << a << " float b is " << b << endl;
43}
44
45
46Output From Trying To Compile This Program
47
48CC polydef.cpp
49"polydef.cpp", line 18: error:  two exact matches for func1();  void(int)  and void(int,float)
50Ambiguous Function Call
51Compilation Failed
```

```
// refs.cpp
// C++ introduces the concept of a reference
// A reference IS NOT a pointer, it has no memory of its own
// A pointer is a pointer, it has memory of its own in which it
// stores the address of the thing it is pointing at

// In this program we introduce the basic syntax for declaring and using
// a reference

// Subsequent programs, especially refs2.cpp, show the use of a reference

#include <iostream.h>

main()
{
	int a = 1;		// a is an integer, it has its own memory
	int & c = a;		// c is NOT an integer, it does not have memory
				// the syntax   int &   identifies c as a reference
	int * ptr = &a;		// ptr is a pointer to an integer
				// it has the address of a stored in it

	cout << "value of a is " << a << endl;
	cout << "value of c is " << c << endl;
	cout << "value of ptr is " << ptr << endl << endl;

	cout << "address of a is " << &a << endl;
	cout << "address of c is " << &c << endl;
	cout << "address of ptr is " << &ptr << endl << endl;

	a = 3;			// this changes the value of a
	cout << "value of a is " << a << endl;
	cout << "value of c is " << c << endl;
	cout << "value of *ptr is " << *ptr << endl << endl;

	c = 7;			// this also changes the value of a
	cout << "value of a is " << a << endl;
	cout << "value of c is " << c << endl;
	cout << "value of *ptr is " << *ptr << endl << endl;

	*ptr = -32;		// this also changes the value of a
	cout << "value of a is " << a << endl;
	cout << "value of c is " << c << endl;
	cout << "value of *ptr is " << *ptr << endl << endl;
}
```

```
Output From Running Program
value of a is 1
value of c is 1
value of ptr is 0x1833fff4

address of a is 0x1833fff4
address of c is 0x1833fff4
address of ptr is 0x1833fff2

value of a is 3
value of c is 3
value of *ptr is 3

value of a is 7
value of c is 7
value of *ptr is 7

value of a is -32
value of c is -32
value of *ptr is -32
```

Variable	Memory	Address
a c	1	0x1833fff4
ptr	0x1833fff4	0x1833fff2

```cpp
// refs1.cpp
// This program shows the use of reference variables in calling a function
// This notation will drive C programmers CRAZY because it seems to be
// in direct conflict with how C does things.  I agree.  This is confusing

#include <iostream.h>

// a copy of the value of the calling argument is stored in x
void call_by_value ( int x )
{
        cout << "call_by_value received " << x << endl;
        cout << "address of x is " << &x << endl;
        x++;
        cout << "call_by_value generated " << x << endl;
}

// the name of the calling argument is stored in x
// x will not be a local variable, x will BE the calling argument
// x will not have to be dereferenced to access the contents
void call_by_reference ( int& x )
{
        cout << "call_by_reference received " << x << endl;
        cout << "address of x is " << &x << endl;
        x++;
        cout << "call_by_reference generated " << x << endl;
}

// the address of the calling argument is stored in x
// x will be a local variable of type pointer to int
// x will need to be dereferenced to access the contents
void call_by_pointer ( int* x )
{
        cout << "call_by_pointer received " << x << endl;
        cout << "call_by_pointer points at " << *x << endl;
        cout << "address of x is " << &x << endl;
        *x = *x + 1;
        cout << "call_by_pointer generated " << *x << endl;
}
```

```
main()
{
	int i = 1;
	cout << "ADDRESS OF MAIN i IS " << &i << endl;
	cout << "i before call_by_value(i) " << i << endl;
	call_by_value(i);				// only name of variable is used
	cout << "i after call_by_value(i) " << i << endl << endl;

	cout << "i before call_by_reference(i) " << i << endl;
	call_by_reference(i);				// only name of variable is used
	cout << "i after call_by_reference(i) " << i << endl << endl;

	cout << "i before call_by_pointer(i) " << i << endl;
	call_by_pointer(&i);				// need to generate address of variable
	cout << "i after call_by_pointer(i) " << i << endl << endl;
}
```

Output From Running Program	**variable**	**memory**	**address**
ADDRESS OF MAIN i IS 0xd37fff4	main's i	1	0xd37fff4
	call_by_value		
i before call_by_value(i) 1			
call_by_value received 1	local x	1	0xd37fff2
address of x is 0xd37fff2			
call_by_value generated 2			
i after call_by_value(i) 1			
i before call_by_reference(i) 1			
call_by_reference received 1	**call_by_reference**		
address of x is 0xd37fff4			
call_by_reference generated 2	no local x	Main's i manipulated directly	
i after call_by_reference(i) 2			
i before call_by_pointer(i) 2	**call_by_pointer**		
call_by_pointer received 0xd37fff4			
call_by_pointer points at 2	ocal x	0xd37fff4	0xd37fff2
address of x is 0xd37fff2	pointer to int		
call_by_pointer generated 3			
i after call_by_pointer(i) 3 manipulates main's i through *x			

```cpp
// refs2.cpp
// This program shows an EXTREMELY interesting way in which references can be utilized
// It draws on the fact that a reference can appear on the LEFT of an
// assignment statement or on the RIGHT of an assignment statement
// Also, a reference can be the return value of a function

#include <iostream.h>

// this function receives references to two variables
// It decides which one is largest and RETURNS THE REFERENCE
int & largest(int& a, int& b)
{
	if ( a > b )
		return a;
	else
		return b;
}

main()
{
	int i = 10;
	int j = 100;
	int x = 0;

	// this chunk shows how we EXPECT to use a function
	cout << "x is " << x << endl;
	x = largest(i,j);
	cout << "x is " << x << endl << endl;

	// by having the function return a reference, we can do this
	// We are adding one to whichever of the two is larger
	cout << "i is " << i << endl;
	cout << "j is " << j << endl;
	largest(i,j)++;
	cout << "i is " << i << endl;
	cout << "j is " << j << endl << endl;

}
```

Output From Running Program

```
x is 0
x is 100

i is 10
j is 100
i is 10
j is 101

largest(i,j)        resolves to either the variable i or j
                    it does not resolve to the value of i or j
                    it does not resolve to the address of i or j
                    it resolves to the variable i or j

therefore           largest(i,j)++       resolves to either i++  or j++
```

Section 3

Class Mechanism
Private Data
Public Member Functions

Programs use data structures. C provides the struct keyword for the specification of user defined data structures. Programs use subroutines to manipulate the data structures. If the data structures are only to be manipulated by the subroutiens, and the subroutines only manipulate the data structures, then why are they semantically separate entities?

Suppose that you have produced a data structure. You write subroutines to manipulate instances of this data structure. A user of your data structure may decide not to use the subroutiens that you wrote. They write their own, and make many mistakes with your data structure. Who is going to be held accountable? Why you of course, the poor sot that created the data structure that provided to be impossible to use! What is preventing the programmer from bypassing the carefully constructed subroutines that you wrote to work with the data structure? In C, Nothing, nada, zilch, zippo. Your data structures are wide open to the marauding hordes of hackers. In C++, you can prevent users from using any subroutines except those that you provided to manipulate the data inside your data structure. This can be extremely useful in producing verifiably correct programs.

The C++ class mechanism formalizes the practices of skilled programmers by enforcing data hiding and function exclusivity. C++ allows subroutines to be declared and defined WITHIN the context of a data structure. C++ also allows the programmer to specify what subroutines may act on what data structures.

ch1p2.cpp	Demonstrates C++ using a traditional C structure and a subroutine to work on instances of the structure
ch1p3.cpp	Demonstrates C++ using a class private data and public member functions to access that data are shown
struct1.cpp	A C programmer ruining a program through invalid data put in YOUR data structure
struct2.cpp	C++ preventing the programmer from inserting invalid data through data hiding
strctev1.cpp	Rules, features usage of and ramifications of the public, protected and private sections of a class
strctev2.cpp	CONSTRUCTORS, putting initial values into class instances
strctev3.cpp	Polymorphism and constructors
strctev4.cpp	Efficient constructors
static1.cpp	Static data in a class
static2.cpp	More static data in a class
strctev5.cpp	Error checking in the constructor
strctev6.cpp	Softer error checking in the constructor
strctev7.cpp	Multiple polymorphic constructors
strctev8.cpp	Polymorphism in a class
strctev9.cpp	External routines used by a class
destrct1.cpp	Simple constructors and destructors
destrct2.cpp	Multiple scopes and when things are destructed
destrct3.cpp	Arrays of objects being constructured and destructed
destrct4.cpp	Pointers in destructors and what gets destructed the pointer, or the thing being pointed at
destrct5.cpp	Pointers in destructors, a different strategy
destrct6.cpp	Pointers in destructors, static data controlling destruction of pointed to area
destrct7.cpp	Does the destructed memory actually get destructed? or do we just lose our pointer to it?
ch1p4.cpp	The role of public, protected and private in function visibility
strct10.cpp	Internalizing the error checking routines of strct9.cpp
strct11.cpp	What you do after the constructor is done.

```
// ch1p2.cpp
#include <stdio.h>                  // get access to printf scanf
#include <iostream.h>               // get access to cout and cin

// a structure definition, the keyword struct works the same way as in C
struct a
{
	int a;                          // a field, legal values are 0 to 10
};

// seta is a subroutine with error checking to ensure that a legal value is
// put into the field in the structure of type struct a.
// the question is,  What is FORCING the user to use this routine?
// Since they have unrestricted access to the data elements of the
// structure, they can set a value into the field without using your
// error checking routine, defeating the purpose of having the subroutine.
void seta(struct a * sa, int ina)
{
	while ( ina < 0 || ina > 10 )
	{
		printf("%i is invalid, please enter number in range 0,10\n");
		scanf("%i",&ina);
	}
	sa->a = ina;
}

main()
{
	struct a a1;                    // create an instance of the data type struct a

	// these first two sections use the variable a1 of type structa
	// in conjunction with the subroutine seta that was designed to error
	// check values going into an instance of a struct a
	seta(&a1,3);;                   // this will call the subroutine that does the error
	                                // checking for the value you want to put in a1
	printf("field a of a1 is %i \n",a1.a);

	seta(&a1,-7);                   // this will call subroutine that does error check
	printf("field a of a1 is %i \n",a1.a);

	// this code shows how a programmer can IGNORE the routine you wrote
	// to put the values into instances of struct a
	// in the first case, no harm is done
	// in the second case, an illegal value arrives in the data structure
	// EVEN THOUGH THERE IS A ROUTINE TO PREVENT THIS, NOTHING ENFORCED IT
	a1.a = 10;                      // programmer has direct access to fields of variables
	                                // of type struct a
	printf("field a of a1 is %i \n",a1.a);

	a1.a = -2;                      // programmer has direct access to fields of variable
	printf("field a of a1 is %i \n",a1.a);
}
```

Output From Running Program

```
field a of a1 is 3

0 is invalid, please enter number in range 0,10
23
23 is invalid, please enter number in range 0,10
98
98 is invalid, please enter number in range 0,10
5
field a of a1 is 5
```

User uses subroutine and experiences error checking the way we would hope they would

```
field a of a1 is 10
```

User uses direct access but luckily puts in a valid value

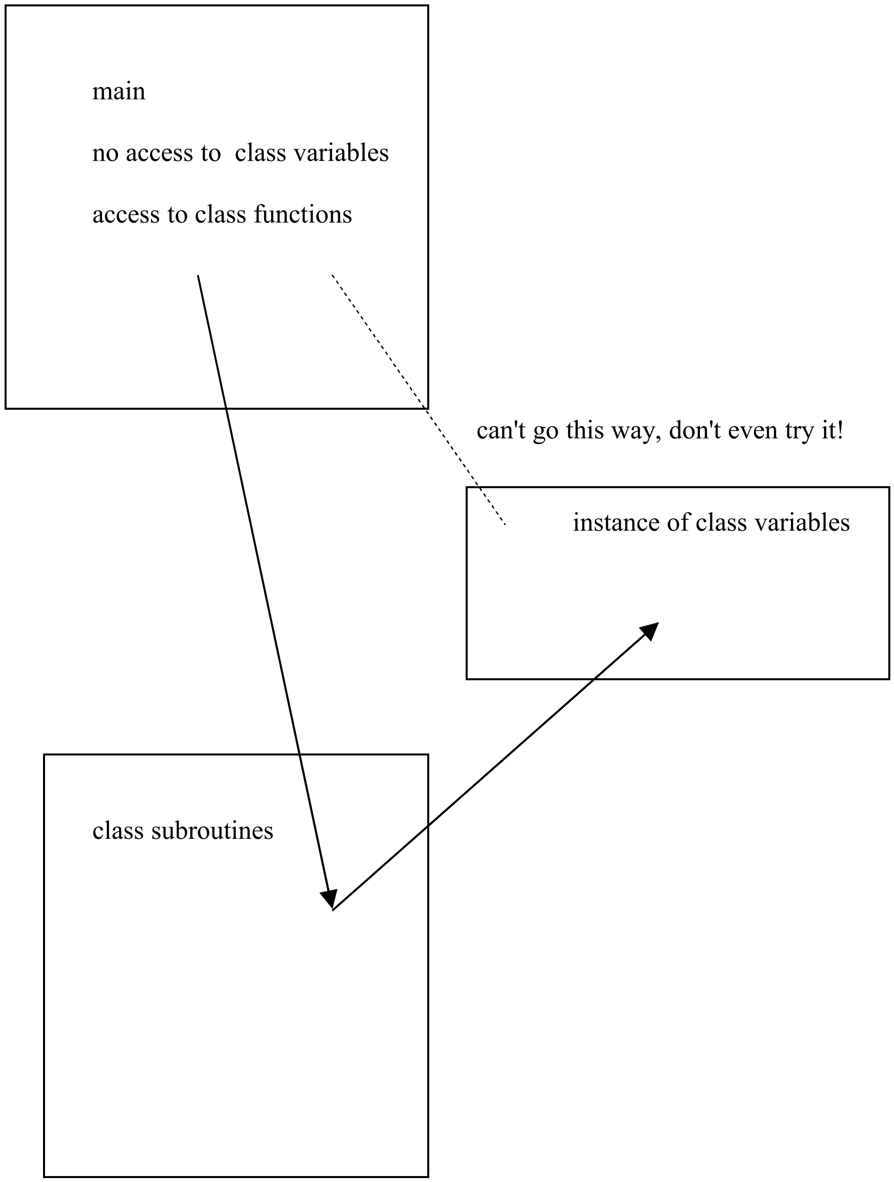
```
field a of a1 is -2
```

User uses direct access and puts in an invalid value

THERE WAS NOTHING STOPPING THE PROGRAMMER FROM DIRECTLY ACCESSING THE DATA IN THE STRUCTURE. THEY WERE ABLE TO WANTONLY BYPASS ERROR CHECKING!!

```
// ch1p3.cpp
#include <stdio.h>                    // get access to printf scanf
#include <iostream.h>                 // get access to cout and cin

// a class definition instead of a structure definition
// the keyword private indicates that the field listed in the private area
// can only be manipulated by the functions listed in the public area
// the programmer does not have direct access to the fields in the private area
// only the coder of the class has access to the fields
// if the user wants to put a value in field a, they HAVE to go through  the member function seta
class a {
        private:
                int a;                        // a field, legal values are 0 to 10
        public:
                void seta(int );              // a public member function to set a
                void printa(void);            // a public member function to print a
};

// the function name is now expanded from seta to a::seta
// the a:: identifies that fact that this is the seta function that belongs to the class a
// note that unlike the seta from ch1p2.cpp, that no pointer to a struct a
// is passed in by the programmer, the system passes a pointer for you
// when a function is a member of a class, an instance of that class is BOUND
// to the function call by the calling syntax, the BOUND variable is called
// the INVOKING variable and its address is stored in the pointer variable
// "this"

// a sample call to this function would be  a1.seta(-3);
// the variable a1 would be BOUND to the call and its address would be stored
// in a pointer     "this"

return     class     function        argument
type       name      name            list



void a::seta(int ina)
{
        while ( ina < 0 || ina > 10 )
        {
                printf("%i is invalid, please enter number in range 0,10\n");
                scanf("%i",&ina);
        }
        // this was the pointer that got the address of the INVOKING variable
        this->a = ina;
}

void a::printa(void)
{
        printf("The value of field a is %i\n",this->a);
        return;
}
```

```
main()
{
	// when declaring variables that are of a user defined class,
	// the keyword class is not required
	a a1;			// create an instance of the data type class a

	// these first two sections use the variable a1 in conjunction with
	// the subroutine seta that was designed to error check values going
	// into an instance of a struct a
	a1.seta(3);		// this will call the subroutine that does the error
				// checking for the value you want to put in a1
	// because the field a is private, you can't print it from this code
	// it is only available to member functions of the class
	// this we have to use the member function printa
	// printf("field a of a1 is %i \n",a1.a);
	a1.printa();


	a1.seta(-7);		// this will call subroutine that does error check
	a1.printa();

	// these lines will not compile, because a is a private field
	// the error checking that you built into seta is ENFORCED
	// remove the #if 0 and the #endif from the code and try to compile
	// you will see the compiler error
	// a::a is not accessible in function main()
#if 0
	a1.a = 10;
	a1.a = -2;
#endif
}
```


main
no access to class variables
access to class functions
can't go this way, don't even try it!
instance of class variables
class subroutines

```
// struct1.cpp
// We have seen that unrestricted programmer access to structure fields is BAD
// Because a structure can be made to take on invalid values, every subroutine
// that deals with the structure type must have error checking included in it.
// This is wasteful.  It would be better if there was no way for an invalid value ever to get into
// an instance of a data structure that you created

#include <stdio.h>      // get access to printf and scanf
#include <stdlib.h>     // get access to exit

// account for the zero indexing, users typically say January is month 1
char * months[13] =
{
        "fill", "January", "February", "March", "April", "May", "June",
        "July", "August", "September", "October", "November", "Dec"
};

// Don't worry about leap year in this example
// account for the zero indexing, users typically say January is month 1
int days_per_month[13] = { 100, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

struct d { int month, day, year; };     // typical struct definition

void inc_date( struct d * );            // function to move to next date
void print_date( struct d * );          // function to print what month it is

main()
{
        // what will be the initial values for d1 and d2??  Whatever the SYSTEM decides.
        struct d d1;                    // create an instance of an d
        struct d d2;                    // create another instance of an d

        d1.month = 12;
        d1.day = 25;
        d1.year = 1994;                 // a valid date has been initialized, okay

        print_date(&d1);                // display valid date
        inc_date(&d1);                  // call f1 with a valid date
        print_date(&d1);                // call f2 with a valid date

        d1.month = 98;                  // make a valid date into an invalid one
                                        // there is nothing preventing the programmer from doing this
        inc_date(&d1);          // call the function with an invalid date, error checking required in routine
        print_date(&d1);        // call the function with an invalid date, error checking required in routine
}
```

```
// this function is supposed to move the date to the next day
// it first has to check to see that the date passed to it is valid
void inc_date ( struct d * ptr1 )
{
	// FORCED TO ERROR CHECK IN EVERY ROUTINE THAT DEALS WITH STRUCT D
	if ( ptr1->year < 0 )   // make sure our year is valid
	{
		printf("Invalid year field, cannot print, exiting\n");
		exit(-1);
	}
	// make sure our month is valid
	if ( ptr1-> month < 1 || ptr1->month > 12 )
	{
		printf("Invalid month field, cannot print, exiting\n");
		exit(-2);
	}
	// make sure our day is valid
	if ( ptr1->day > days_per_month[ptr1->month] || ptr1->day < 1 )
	{
			printf("Invalid day field, cannot print, exiting\n");
			exit(-3);
	}
	// now that we have a valid date, update it, wrapping if necessary
	ptr1->day++;
	if ( ptr1->day > days_per_month[ptr1->month] ) {
		ptr1->day = 1;    ptr1->month++;
		if ( ptr1->month == 13 ) {
			ptr1->month = 1;
			ptr1->year++;
		}
	}
	return;
}

void print_date( struct d * ptr1 ) {
	// FORCED TO ERROR CHECK IN EVERY ROUTINE THAT DEALS WITH STRUCT D
	if ( ptr1->year < 0 )   // make sure our year is valid{
		printf("Invalid year encountered \n");
		exit(-4);
	}
	if ( ptr1-> month < 1 || ptr1->month > 12 ) {
		printf("Invalid month encountered \n");
		exit(-5);
	}
	if ( ptr1->day > days_per_month[ptr1->month] || ptr1->day < 0 )  {
			printf("Invalid day encountered\n");
			exit(-6);
	}
	// only now that we have error checked the whole thing can we print
	printf("%i of %s in %i\n",ptr1->day,months[ptr1->month],ptr1->year);
	return;
}
```

```
Output From Running Program

25 of Dec in 1994
26 of Dec in 1994
Invalid month field, cannot print, exiting
```

```
// struct2.cpp
#include <stdio.h>      // get access to printf
#include <stdlib.h>     // get access to exit

// the set_vals routine will deal with the zero indexing, we don't need a bogus entry for months[0]
char * months[12] =
{
        "January", "February", "March", "April", "May", "June",
        "July", "August", "September", "October", "November", "December"
};

// set_vals will deal with zero indexing, don't need a bogus entry in days_per_month[0]
int days_per_month[12] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

class d
{
        // only functions declared as part of class d are given access to
        // month, day or year fields of instances of the class
        private:
                int month, day, year;

        public:
                void set_vals(int,int,int);  // function to access fields
                void inc_date( void );       // function to increment date
                void print_date( void );     // function to print fields
};

main()
{
        // what will be the initial values for d1 and d2??
        d d1;                           // create an instance of an d

        // these statements are illegal in C++ because month day and year
        // are private elements of the struct d type of data
        // d1.month = 12;
        // d1.day = 25;
        // d1.year = 1994;

        // to set a date into d1, you have to use set_vals
        d1.set_vals(12,25,1994);   // apply the set_vals function to d1

        d1.print_date();                // display valid date using f2 function of the variable d1

        d1.inc_date();                  // move d1 ahead to the next date
        d1.print_date();                // display the new date

        // try to set an invalid date into d1, it won't work!
        d1.set_vals(123,456,-987);
        d1.print_date();              // display valid date
}
```

```
// this function does NOT HAVE TO ERROR CHECK because there is no
// way for an instance of type class d to get into an invalid state
void d::inc_date()
{
        NO ERROR CHECKING REQUIRED HERE!!
    this->day++;
    if ( this->day > days_per_month[this->month] )
    {
        this->day = 1;
        this->month++;
        if ( this->month > 12 )
        {
            this->month = 0;
            this->year++;
        }
    }
    return;
}

// this function does not have to error check because there is no way
// for a struct d instance to get invalid values
void d::print_date( )
{
        NO ERROR CHECKING REQUIRED HERE!!!
    printf("%i of %s in %i\n",this->day,months[this->month],this->year);
    return;
}
```

```
// void is the return type
// set_vals is the name of the function
// a,b, and c are the expected parameters
// we want to identify that this code is the code that implements the
// function set_vals for the struct d data type, the d:: does this

return  class   function        argument
type    name    name            list






void d::set_vals(int a, int b, int c )
{
        printf("Setvals got %i %i %i \n",a,b,c);
        // when called via a call like d1.set_vals(1,2,3)
        // C++ creates a pointer, "this", which is local to this subroutine
        // this, in the call d1.set_vals(1,2,3) will end up with the addr of d1
        // if set_vals were called via a call like d2.set_vals(3,4,5)
        // then this would take on the address of d2
        if ( c < 0 )   // make sure our year is valid
        {
                printf("Invalid year encountered\n");
                exit(-1);
        }

        // make sure our month is valid
        if ( a < 1 || a > 12 )
        {
                printf("Invalid month encountered\n");
                exit(-2);
        }

        // make sure our day is valid
        if ( b > days_per_month[a - 1] || b < 1 )
        {
                printf("Invalid day encountered\n");
                exit(-3);
        }

        // if we get to here, then all the values checked out
        this->month = a - 1;  // account for array indexing starting at 0
        this->day   = b;
        this->year  = c;
}
```

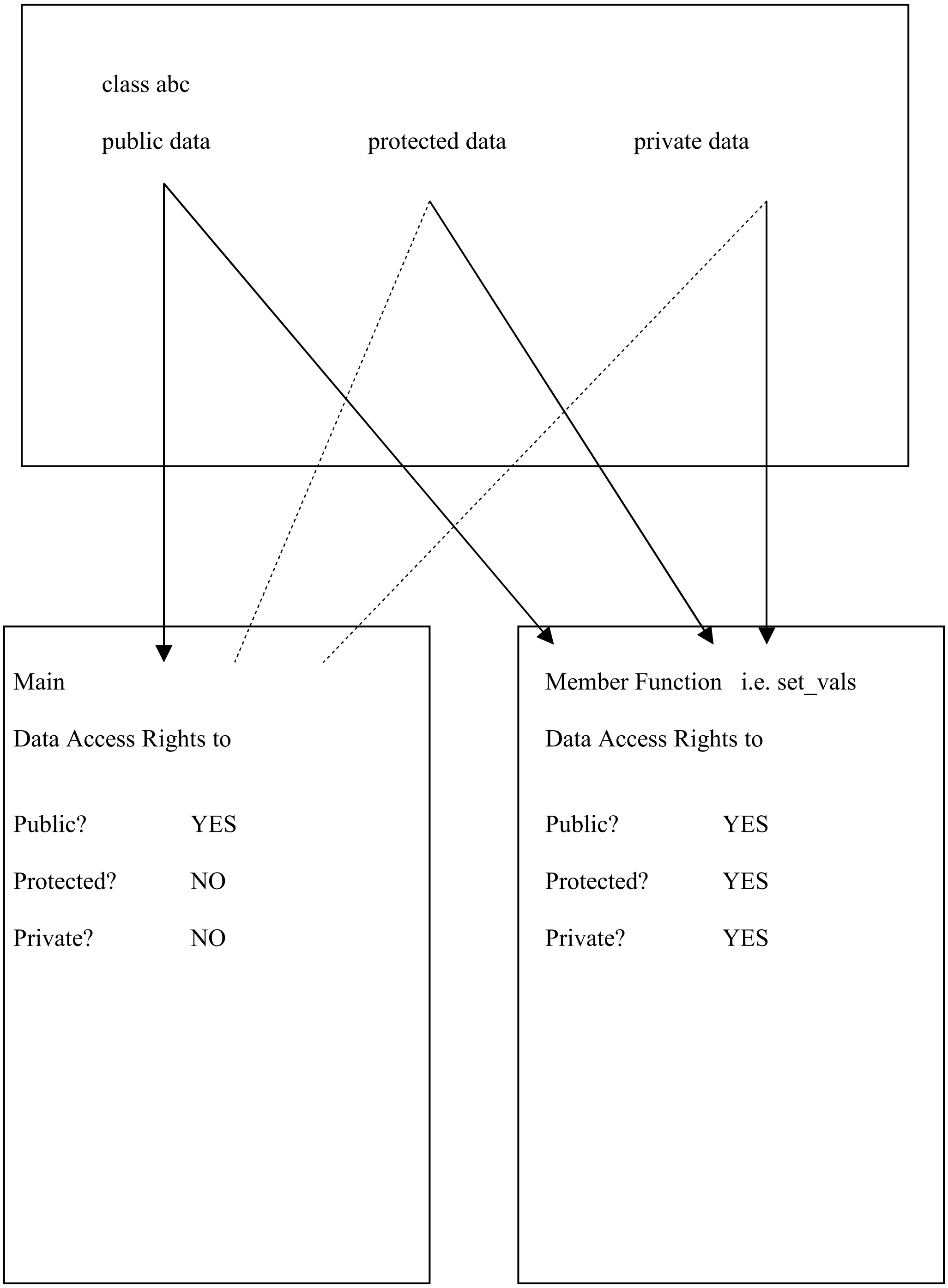
```
Output From Running Program

Setvals got 12 25 1994
25 of December in 1994
26 of December in 1994

Setvals got 123 456 -987
Invalid year encountered
```

```
// strctev1.cpp
// Now that we have seen the 2 rudiments of the class keyword,
// the private and public sections, let's examine, in a somewhat exhuastive
// manner, the rules, features, usage of and ramifications of the public,
// private and protected sections of a class

// A C++ class may have up to three sections:
//          public   protected          private

// each of these sections has different properties with respect
// to the visibility of data fields and member functions
// Let's examine the visibility of data fields first
// There are two locations where a data field's visibility needs to be checked
// First, within the code of a member function of the class
// Second, within the code found in main, code that is NOT IN A MEMBER FUNCTION i.e. in main
#include <stdio.h>

class abc
{
        public:
                int a;                          // public data, visible everywhere
                void set_vals(int,int,int);     // public member function
        protected:
                int b;                          // protected data, visible only in member functions
        private:
                int c;                          // private data, visible only in member functions
};

void abc::set_vals(int A, int B, int C)
{
        printf("Address of bound object is %x \n",this);
        printf("Before setting: a = %i \t b = %i \t c = %i \t \n", a,b,c);
        this->a = A;                    // public    data visible to member function
        this->b = B;                    // protected data visible to member function
        this->c = C;                    // private   data visible to member function
        printf("After setting: a = %i \t b = %i \t c = %i \t \n", a,b,c);
}

main()
{
        abc a1;                         // create an instance of an abc, don't know what initial values will be

        // a is a public data field, this line will compile
        a1.a = 1;                       // attempt to individually assign values

        // b is a protected data field, this line WILL NOT compile
        // a1.b = 2;                    // if you uncomment this line you will see compiler error message
        // abc::b is not accessible in function main()

        // c is a private data field, this line WILL NOT compile
        // a1.c = 3;                    // if you uncomment this line you will see compiler error message
        // abc::c is not accessible in function main()
```

```
        a1.set_vals(5,6,7);     // this is how I could set the values of fields a b and c in a1 of class abc

        // What if I tried to create another variable a2 of type abc?
        // The question is, from this location, within main, not within
        // the code of a member function, can I provide intial values?
              // WITH WHAT WE KNOW NOW??
        // NO.  The field c is a private data element, therefore I cannot
        // directly address it from here

        // Question?  Am I then prevented from providing initial values for an instance of a variable?
        // NO.  See the next program for Constructor code
        // this line will not compile
        // abc a2 = { 5,6,7 };     // create another instance of an abc
        // if you uncomment this line you will see compiler error message
        // Classes cannot be initialized with {} in function main()
}
```

Output From Running Program

```
Address of bound object is fff0
Before setting:    a = 1      b = 314  c = 1456
After setting:     a = 5      b = 6      c = 7
```


class abc
public data
protected data
private data
Main
Data Access Rights to
Public? YES
Protected? NO
Private? NO
Member Function i.e. set_vals
Data Access Rights to
Public? YES
Protected? YES
Private? YES

```
// strctev2.cpp
// the previous program raised the question about how do we assign initial
// values to an instance of a class at initialization time?
// this program introduces the CONSTRUCTOR code

#include <stdio.h>

class abc
{
	public:
		int a;				// public data, visible everywhere
		void set_vals(int,int,int);	// public member function to assign
						// values to an existing variable
		// the name of the CONSTRUCTOR function must match the name of
		// the class, the implied, default and in fact only possible
		// return type for a CONSTRUCTOR function is a variable of the
		// class to which the constructor function belongs
		abc(int,int,int);		// public member function to assign
						// values to a variable being  created  CONSTRUCTOR FUNCTION
		void print_vals();		// public member function to print all three fields
	protected:
		int b;				// protected data, visible only in member functions
	private:
		int c;				// private data, visible only in member functions
};

// the return type is implied, defaulted, and fixed to be of type abc
// no return statement is used,  the use of "this" is not employed
class	function	argument
name	name		list




abc::abc(int A, int B, int C )
{
	// This is the ideal location to do error checking of initial values
	// see a subsequent program for this option
	a = A;				// assign to the public    field
	b = B;				// assign to the protected field
	c = C;				// assign to the private   field
}

void abc::set_vals(int A, int B, int C)
{
	// this is also a very good place to do error checking of update values
	// this will be done in a subsequent program
	// NOTICE that this subroutine uses the this pointer, the subroutine
	// above took advantage of "implied this" and saved typing this->
	// for each field
	this->a = A;			// public    data visible to member function
	this->b = B;			// protected data visible to member function
	this->c = C;			// private   data visible to member function
}
```

```
void abc::print_vals()
{
	printf("a = %i \t b = %i \t c = %i \n",this->a, this->b, this->c);
}

main()
{
	// Now I try to create a2 and assign initial values to all three
	// fields.  The function abc, which is a member function of the class
	// abc and which expects three integer arguments, will be called
	// Because abc(int,int,int) is a member function, it has access to
	// all three fields
	abc a2(5,6,7);		// create an instance of an abc class object giving three integers as arguments
	a2.print_vals();

	// abc a1;			// this line is now a problem
	// in the previous program, I provided NO CONSTRUCTOR
	// therefore C++ provided one for me
	// that DEFAULT NO ARGUMENT CONSTRUCTOR simply allocated the memory
	// for the variables
	// Now that I have provided a constructor function, I HAVE TO USE IT
	// the line
	// abc a1;
	// says to build an abc but it doesn't provide
	// the three integers that I said I would use to build one
	// SO what do you do?  See the next program
}

Output From Running Program
a = 5	b = 6	c = 7
```

```
// strctev3.cpp
// the previous program introduced a three argument constructor for the abc
// class.  We saw that this produced a problem, it caused the default no
// argument constructor to become unavailable for our usage.
// BUT what if WE WANTED a no argument constructor AND a three arguement constructor?
// IN C++ we can have two functions with the same name be members of the
// same class as long as they have different arguement lists
// So in this program, I give two functions named abc
// one is abc(int,int,int)
// the other is abc(void)
// both of these are CONSTRUCTOR functions
// one is called when we give three integers when creating an abc
// the other is called when we give NO arguments when creating an abc
#include <stdio.h>                    // get access to printf
#include <iostream.h>                 // get access to cout

class abc
{
    public:
        int a;                        // public data, visible everywhere
        void set_vals(int,int,int);   // public member function to assign
                                      // values to an existing variable
        // there are two functions with the name abc
        // they can be told apart by their argument lists
        abc(int,int,int):             // THREE ARGUMENT CONSTRUCTOR
                                      // public member function to assign values to a variable being
                                      // created  CONSTRUCTOR FUNCTION
        abc(void):                    // ZERO ARGUMENT CONSTRUCTOR
        void print_vals();            // public member function to print all three fields
    protected:
        int b;                        // protected data, visible only in member functions
    private:
        int c;                        // private data, visible only in member functions
};

// three argument constructor
abc::abc(int A, int B, int C )
{
    // would do error checking here
    cout << "Three Argument Constructor Called \n";
    a = A;                            // assign to the public    field
    b = B;                            // assign to the protected field
    c = C;                            // assign to the private   field
}
// zero argument constructor
abc::abc()
{
    // would do error checking here
    cout << "NO ARGUMENT CONSTRUCTOR CALLED \n";
    a = 1;                            // I just decided to put these three values into the fields
    b = 2;                            // since you get to write the constructor, you get to decide what
    c = 3;                            // initial values go in fields that otherwise would be set by the system
}
```

```
void abc::set_vals(int A, int B, int C)
{
	// would do error checking here
	this->a = A;			// public    data visible to member function
	this->b = B;			// protected data visible to member function
	this->c = C;			// private   data visible to member function
}

void abc::print_vals()
{
		// No need to do any error checking in this routine because all three avenues of approach
		// to an instance of an abc,   abc(int,int,int)     abc()    set_vals(int,int,int)   would be protected
		// by error checking
	printf("a = %i \t b = %i \t c = %i \n",this->a, this->b, this->c);
}

main()
{
	abc a1;				// this line is no longer a problem
					// because we have a zero argument constructor in this application

	a1.print_vals();		// show the initial values we assigned
	a1.set_vals(10,20,30);		// call the member function
	a1.print_vals();		// show the new values

	// this line will also work because we have a three argument
	// constructor for the class abc
	abc a2(5,6,7):			// create another instance of an abc
	a2.print_vals();		// show its initial values
	a2.set_vals(-34,21,-98234);	// call the member function
	a2.print_vals();		// show the new values
}

Output From Running Program

NO ARGUMENT CONSTRUCTOR CALLED
a = 1	b = 2	c = 3
a = 10	b = 20	c = 30

Three Argument Constructor Called
a = 5	b = 6	c = 7
a = -34	b = 21	c = -98234
```

```
// strctev4.cpp
// the previous program introduced polymorphic constructors
// this program introduces EFFICIENT constructors

#include <stdio.h>
#include <iostream.h>

class abc
{
	public:
		int a;				// public data, visible everywhere
		void set_vals(int,int,int);	// public member function to assign values to an existing variable

		// Three argument constructor
		abc(int,int,int);

		// Zero argument constructor
		abc(void);

		void print_vals();

	protected:
		int b;				// protected data, visible only in member functions

	private:
		int c;				// private data, visible only in member functions
};

// this constructor uses INVOCATION LINE INITIALIZATION
// all the memory for an instance of class abc is allocated before the first
// executable statement of the constructor is performed
// By using the a(A) syntax after the colon on the invocation line
// the value of A is passed to the built in one arg C++ integer constructor
// This is more efficient than assigning the value of A to a inside the
// constructor code because if you don't say a(A) then the no arg C++
// constructor is called and you are forced to do a = A.  That is two steps
// instead of one.  It's like ordering a car, taking whatever color they send
// you and then painting it red.  If you wanted a red car, why didn't you
// order a red car?

// C++ provides one arg constructors for all predefined types
// C++ provides on arg constructors for all user defined types
// As soon as you provide any constructor for a user defined type, the C++ no arg constructor goes away

```

```
                    // this is the efficient method to initialize

set variable a to value A    set variable b to value B    set variable c to value C
by using C++ built in        by using C++ built in        by using C++ built in
one arg integer              one arg integer              one arg integer
constructor                  constructor                  constructor








abc::abc(int A, int B, int C ) : a(A), b(B), c(C)
{
	cout << "Three Argument Constructor Called \n";
	// this would be the inefficient method
	// a = A;	// system would have already assigned value to a, you would be re-assigning a value
	// b = B;
	// c = C;
}

// the more efficient INVOCATION LINE INITIALIZATION method for the no arg constructor
// The ramifications of this type of initialization are small when you
// just have pre-defined data types as the fields in your class
// HOWEVER, when you have fields that themselves are classes, then the
// ramifications can be quite large.
abc::abc() : a(1), b(2), c(3)
{
	cout << "NO ARGUMENT CONSTRUCTOR CALLED \n";
	// a = 1;	// this is the inefficient way
	// b = 2;
	// c = 3;
}

// this is NOT a constructor function, therefore it CANNOT do invocation line
// initialization.  Invocation line initialization is reserved for constructors
void abc::set_vals(int A, int B, int C)
{
	this->a = A;  // public    data visible to member function
	this->b = B;  // protected data visible to member function
	this->c = C;  // private   data visible to member function
}

void abc::print_vals()
{
	printf("a = %i \t b = %i \t c = %i \n",this->a, this->b, this->c);
}
```

```
main()
{
      abc a1;                              // INVOKE THE ZERO ARGUMENT CONSTRUCTOR

      a1.print_vals();                     // show the initial values the constructor
                                           // assigned to the instance a1 of class abc
      a1.set_vals(10,20,30):               // call the member function
      a1.print_vals();                     // show the new values

      abc a2(5,6,7):                       // INVOKE THE THREE ARGUMENT CONSTRUCTOR

      a2.print_vals();                     // show initial values
      a2.set_vals(-34,21,-98234);          // call the member function
      a2.print_vals();                     // show the new values

}
```

Output From Running Program

```
NO ARGUMENT CONSTRUCTOR CALLED
a = 1      b = 2      c = 3
a = 10     b = 20     c = 30

Three Argument Constructor Called
a = 5      b = 6      c = 7
a = -34    b = 21     c = -98234
```

```
abc::abc(int A, int B, int C ) : a(A) , b(B), c(C)
{

        efficient way to initialize




}
```

memory for the three ints is allocated here.
We provide initial values.
This is analogous to saying   int a = A   int b = B;   int c = C;
when the integers are declared

```
abc::abc(int A, int B, int C)
{
a = A;          // you are resetting initial value provided by C++ no arg int constructor
ib = B;         // again, you are re-assigning a value into b
c = C;          // Too late, you are slow, you are bad, you are an
                        INEFFICIENT PROGRAMMER
```

why would you write:

```
int a;
a = 5;
```

when you could write

```
int a = 5;
```

```
// static1.cpp
// classes can have STATIC variables
// A static variable is SHARED amongst all instances of variables of that class
// The static variable should not be initialized in the constructor of a class
// because every object that is created would then be initializing the shared variable

// The shared variable should be initialized just before you start main
// The shared variable still obeys the public, protected, private rules of any other variable

// Shared variables are very useful for tracking how many objects were created
// for determining how many times some subroutine was called and so on
// for sharing information amongst every instance of a user defined type

#include <iostream.h>

class abc
{
	public:
		abc();				// no arg constructor
		void print_vals();
	private:
		static int num_of_abcs;		// static variable, private data, only visible to members
						// will keep track of how many abcs are created

		int a,b,c;			// private data, visible only in member functions
};

abc::abc()
{
	cout << "No arg constructor called\n";
	cout << "Current value of num_of_abcs is " << num_of_abcs << endl;
	num_of_abcs++;
	a = 1 * num_of_abcs;			// uses num_of_abcs to do something, your programs would
	b = 2 * num_of_abcs;		// likely have more interesting uses of static data than does
	c = 3 * num_of_abcs;			// this teaching program
}

void abc::print_vals()
{
	cout << "Values are: \n";
	cout << "a " << a
	     << "   b " << b
	     << "   c " << c << endl;

	cout << "static field num_of_abcs has: " << num_of_abcs << endl;
}

int abc::num_of_abcs = 0;			// initialize the static data
```

```
main()
{
	abc a1;		// create an instance of an abc
	a1.print_vals();

	abc a2;		// create a second one
	a2.print_vals();

	abc a3[4];	// create four more

	for ( int i = 0; i < 4; i++ )
		a3[i].print_vals();

		// you could not do this, because num_of_abcs was declared as private data
		// abc::num_of_abcs = 10;
}
```

Output From Running Program

```
No arg constructor called
Current value of num_of_abcs is 0
Values are:
a 1   b 2   c 3
static field num_of_abcs has: 1

No arg constructor called
Current value of num_of_abcs is 1
Values are:
a 2   b 4   c 6
static field num_of_abcs has: 2

No arg constructor called
Current value of num_of_abcs is 2
No arg constructor called
Current value of num_of_abcs is 3
No arg constructor called
Current value of num_of_abcs is 4
No arg constructor called
Current value of num_of_abcs is 5
Values are:
a 3   b 6   c 9
static field num_of_abcs has: 6
Values are:
a 4   b 8   c 12
static field num_of_abcs has: 6
Values are:
a 5   b 10   c 15
static field num_of_abcs has: 6
Values are:
a 6   b 12   c 18
static field num_of_abcs has: 6
```

```
// static2.cpp
// this program shows what happens if you initialize the shared variable
// in the constructor, which is the wrong place

#include <iostream.h>

class abc
{
	public:
		abc();				// no arg constructor
		void print_vals();
	private:
		static int num_of_abcs;	// static variable

		int a,b,c;			// private data, visible only in member functions
};

abc::abc()
{
	num_of_abcs = 0;			// num_of_abcs will be set to zero by EVERY abc that is created
	cout << "No arg constructor called\n";
	cout << "Current value of num_of_abcs is " << num_of_abcs << endl;
	a = 1 * num_of_abcs;
	b = 2 * num_of_abcs;
	c = 3 * num_of_abcs;
}

void abc::print_vals()
{
	cout << "Values are: \n";
	cout << "a " << a
	     << "   b " << b
	     << "   c " << c << endl;

	cout << "static field num_of_abcs has: " << num_of_abcs << endl;
}

main()
{
	abc a1;				// create an instance of an abc, initializes num_of_abcs to zero
	a1.print_vals();

	abc a2;				// create a second one, sets num_of_abcs back to zero
	a2.print_vals();

	abc a3[4];			// create four more, each one sets num_of_abcs back to zero

	for ( int i = 0; i < 4; i++ )
		a3[i].print_vals();
}
```

```
Output From Running Program

No arg constructor called
Current value of num_of_abcs is 0
Values are:
a 0   b 0   c 0
static field num_of_abcs has: 0
No arg constructor called
Current value of num_of_abcs is 0
Values are:
a 0   b 0   c 0

static field num_of_abcs has: 0
No arg constructor called
Current value of num_of_abcs is 0
No arg constructor called
Current value of num_of_abcs is 0
No arg constructor called
Current value of num_of_abcs is 0
No arg constructor called
Current value of num_of_abcs is 0

Values are:
a 0   b 0   c 0
static field num_of_abcs has: 0
Values are:
a 0   b 0   c 0

static field num_of_abcs has: 0
Values are:
a 0   b 0   c 0

static field num_of_abcs has: 0
Values are:
a 0   b 0   c 0

static field num_of_abcs has: 0
```

```
// strctev5.cpp
// The constructor provides the perfect place to error check any values
// that someone might want to put into a variable of your defined type
// For example, in C, if you have a date data type, is there anything
// keeping someone from creating a data variable with the values -1/-2/-3 ?
// NO
// In C++, we will be able to perform error checking in the constructor

char * months[12] =
{
	"January", "February", "March", "April", "May", "June", "July",
	"August", "September", "October", "November", "December"
};

int days_per_month[12] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

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

class date
{
	public:
		date(int,int,int);		// function with same name as data type is a CONSTRUCTOR
		void print_date();		// function to display a date
	private:
		int month;			// DOMAIN	1 - 12
		int day;			// DOMAIN  depends on month
		int year;			// DOMAIN	0 to whatever
};

main()
{
	date d1(12,23,1993);		// this is valid, it will work
	d1.print_date();

	date d2(-1,-3,-4);		// this is an attempt to create an invalid
					// instance of a date object, it will cause
					// program termination because the constructor
					// does error checking and the class designer
					// decided that an attempt to create an invalid
					// object was means for termination.  Softer
					// approaches to error handling are up to the
					// individual class designer.  That's part of
					// the whole point of being able to write
					// constructor code
}
```

```
// return type                    void
// complete function name         date::print_date
// argument list                  ()
void date::print_date()
{
        cout << "In member_print_date \n";
        cout << "  month is " << this->month <<
                "  day   is " << this->day <<
                "  year  is " << this->year << endl;
}

// return type                    none explicitly defined,  abc by definition
// complete function name         date::date
// argument list                  three integers for month, day, year
date::date(int m, int d, int y)
{
        // no pointer "this" is available for this function
        // because this function is creating the instance of the abc variable
        // this function implicitly returns a variable of the abc type
        cout << "The three argument constructor has been called " << endl;

        // error check for a valid year, if the year is invalid
        // TERMINATE the program.  If you were coding this constructor you
        // may have chosen a softer approach to error handling, this is
        // demonstrated in subsequent programs.
        if ( y < 0 )
        {
                cout << "Your desired year " << y << " is invalid " << endl;
                exit(-1);
        }
        if ( m < 1  || m > 12 )
        {
                cout << "Your desired month " << m << " is invalid " << endl;
                exit(-2);
        }
        if ( d < 1 || d > days_per_month[m - 1] )
        {
                cout << "Your desired day " << d << " is invalid " << endl;
                exit(-3);
        }
        // if we make it through all the checks assign the fields
        day   = d;
        month = m - 1;  // account for zero indexing
        year  = y;
        return;
}
```

Output From Running Program

```
The three argument constructor has been called
In member_print_date
        month is 11        day is 23          year is 1993

The three argument constructor has been called
Your desired year  -4  is invalid
```

class date
public
date — constructor
print_date — display function

private
month — only accessible through member functions
day
year

can't just assign values to these things
so
if you error check correctly at initialization time
and code your subroutines right
you don't have to error check anywhere else in program

constructor

error checks month

error checks day

error checks year

assigns the values

print_date subroutine

no need to error check

impossible to have invalid
values in a date instance

```
// strctev6.cpp
// In strctev5.cpp, if the value the user wanted to put into the
// date object to be constructed was invalid, we took drastic steps
// and exited the program.  Perhaps we would like to allow the user
// to re-enter the dates for an object of the class or perhaps we would
// just like to assign known valid values, either of these approaches
// has merits.  This program illustrates error checking loops in the constructor

char * months[12] =
{
	"January", "February", "March", "April", "May", "June", "July",
	"August", "September", "October", "November", "December"
};

int days_per_month[12] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

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

class date
{
	public:
		date(int,int,int);		// function with same name as
						// data type is a CONSTRUCTOR
		void print_date();		// function to display a date
	private:
		int month;			// DOMAIN	1 - 12
		int day;			// DOMAIN	depends on month
		int year;			// DOMAIN	0 to whatever
};

main()
{
	date d1(12,23,1993);		// this is valid, it will work
	d1.print_date();

	date d2(-1,-3,-4);		// this is invalid, it will cause the error checking to be invoked
	d2.print_date();
}

// return type				void
// complete function name		date::print_date
// argument list			()
void date::print_date()
{
	// this is the pointer that has the address of the object that was BOUND to this call
	cout << "In member_print_date \n";
	cout << "  month is " << this->month <<
		"  day   is " << this->day <<
		"  year  is " << this->year << endl;
}
```

```
// return type                          none explicitly defined,  abc by definition
// complete function name               date::date
// argument list                        three integers for month, day, year
date::date(int m, int d, int y)  {
        cout << "The three argument constructor has been called " << endl;
        cout << "Input arguments were " << m << " " << d << " " << y << endl;
        if ( y < 0 )
        {
                int tempy = y;      // create a local variable

                while ( tempy < 0 )
                {
                        cout << "Year " << tempy << " is invalid " << endl;
                        cout << "Please enter a year greater than 0 ";
                        cin  >> tempy;
                }
                cout << "Valid y of " << tempy << " accepted " << endl;
                y = tempy;
        }
        if ( m < 1  || m > 12 ) {
                int tempm = m;
                while ( tempm < 1 || tempm > 12 )
                {
                        cout << "Month " << tempm << " is invalid " << endl;
                        cout << "Please enter a month in the range 1 - 12 ";
                        cin >> tempm;
                }
                m = tempm;       // account for zero indexing
                cout << "Valid m of " << tempm << " accepted " << endl;
        }

        if ( d < 1 || d > days_per_month[m - 1] )
        {
                int tempd = d;
                while ( tempd < 1 || tempd > days_per_month[m - 1] )
                {
                        cout << "Day " << tempd << " is invalid " << endl;
                        cout << "Please enter a day in the range 1 - "
                             << days_per_month[m - 1] << " ";
                        cin >> tempd;
                }
                d = tempd;
                cout << "Valid d of " << tempd << " accepted " << endl;
        }

        // once we make it through all the checks assign the fields
        day   = d;               // could have been written this->day   = d
        month = m -1;            // could have been written this->month = m
        year  = y;               // could have been written this->year  = y
        return;
}
```

Output From Running Program

```
The three argument constructor has been called
Input arguments were 12 23 1993
In member_print_date
month is 11  day   is 23  year  is 1993
The three argument constructor has been called
Input arguments were -1 -3 -4
Year -4 is invalid
Please enter a year greater than 0 Valid y of 45 accepted
Month -1 is invalid
Please enter a month in the range 1 - 12 Month 15 is invalid
Please enter a month in the range 1 - 12 Month -7 is invalid
Please enter a month in the range 1 - 12 Valid m of 3 accepted
Day -3 is invalid
Please enter a day in the range 1 - 31 Day 42 is invalid
Please enter a day in the range 1 - 31 Valid d of 12 accepted
In member_print_date
month is 2  day   is 12  year  is 45
```

date class object has constructor and print_date functions defined for it

constructor does error checking loops
no other routine has to do error checking because constructor ensures that
only valid instances of dates are created

Error Checking Loop
for year

print_date
no error checking

Error checking loop
for month

Error checking loop
for days in month
Assign valid entries to date object

```
// strctev7.cpp
// We have already seen that it is possible to have a function with
// one name and two different argument lists.  This is known as
// a polymorphic function.  Can we have multiple polymorphic constructors?
// Yes!
// The program below will allow you create an instance of a date class variable
// by providing either no, one, two or three arguments.  Each of the
// constructors will either decide to do some error checking or to just
// assign some date to fill in the missing arguments.
// Notice that there is duplication of effort in the error checking.
// Perhaps we should consider making subroutines that all they do is error checking and allow
// the constructors to call these routines as they see fit.

// What your constructors would do would depend on you and your application
// Because you are writing the constructors for your class
// You can control what happens when a variable of your class is created
// This is a BIG ADVANTAGE of C++ over C

char * months[12] =
{
	"January", "February", "March", "April", "May", "June", "July",
	"August", "September", "October", "November", "December"
};

int days_per_month[12] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

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

class date
{
	public:
		date();			// zero  argument constructor
		date(int);		// one   argument constructor
		date(int,int);		// two   argument constructor
		date(int,int,int);	// three argument constructor

		void print_date();	// function to display a date
	private:
		int month;		// DOMAIN	1 - 12
		int day;		// DOMAIN	depends on month
		int year;		// DOMAIN	0 to whatever
};
```

```
main()
{
	date d1;			// create a var d1 of type date specifying no args
	d1.print_date();

	date d2(1994);			// create a var d2 of type date specifying one arg
	d2.print_date();

	date d3(3,1994);		// create a var d3 of type date specifying two args
	d3.print_date();

	date d4(3,1,1994);		// create a var d4 of type date specifying three args
	d4.print_date();
}
```

```
// return type                      void
// complete function name           date::print_date
// argument list                    ()
void date::print_date()
{
        cout << "In member_print_date \n";
        cout << "  month is " << this->month <<
                "  day   is " << this->day <<
                "  year  is " << this->year << endl << endl;
}

// no argument constructor, no error checking required, just assign some date
date::date()
{
        cout <<"Welcome to the no arg constructor " << endl;
        // this is the implementation of the no arg constructor
        // I have decided to put in today's date
        // so however your system retrieves the system date let's assume
        // those values ended up in the variable m,d,y okay?
        int m = 6, d = 6, y = 1944;
        month = m;      day = d;        year = y;
}

// one argument constructor, error checking for the year implemented
date::date(int y)
{
        // this is the one argument constructor
        // I have decided to interpret the one arg as the year
        // I have decided to error check it
        // I have decided to set the day and month to January first
        // Why, because I felt like it, you might decide to do something
        // different, that's the whole point of constructors
        // You get to decide on the initial values, not the system or someone
        // else, you get to be in control
        cout << "The one argument constructor has been called " << endl;
        if ( y < 0 )
        {
                int tempy = y;      // create a local variable

                while ( tempy < 0 )
                {
                        cout << "Year " << tempy << " is invalid " << endl;
                        cout << "Please enter a year greater than 0 ";
                        cin  >> tempy;
                }
                cout << "Valid y of " << tempy << " accepted" << endl;
                y = tempy;
        }
        month = 1;
        day = 1;
        year = y;
}
```

```
// two argument constructor with error checking
date::date(int m, int y)
{
	// this is the two argument constructor
	// I have decided to interpret the second arg as the year
	// I have decided to interpret the first arg as the month
	// I have decided to error check them both
	// I have decided to set the day to the first of the month
	cout << "The two argument constructor has been called " << endl;
	if ( y < 0 )
	{
		int tempy = y;	// create a local variable

		while ( tempy < 0 )
		{
			cout << "Year " << tempy << " is invalid " << endl;
			cout << "Please enter a year greater than 0 ";
			cin  >> tempy;
		}
		cout << "Valid y of " << tempy << " accepted" << endl;
		y = tempy;
	}
	if ( m < 1  || m > 12 )
	{
		int tempm = m;
		while ( tempm < 1 || tempm > 12 )
		{
			cout << "Month " << tempm << " is invalid " << endl;
			cout << "Please enter a month in the range 1 -12 ";
			cin >> tempm;
		}
		cout << "Valid m of " << tempm << " accepted" << endl;
		m = tempm;
	}
	month = m;
	day = 1;
	year = y;
}
```

```
// three argument constructor with error checking
date::date(int m, int d, int y)
{
      cout << "The three argument constructor has been called " << endl;
      if ( y < 0 )
      {
            int tempy = y;      // create a local variable

            while ( tempy < 0 )
            {
                  cout << "Year " << tempy << " is invalid " << endl;
                  cout << "Please enter a year greater than 0 ";
                  cin  >> tempy;
            }
            cout << "Valid y of " << tempy << " accepted" << endl;
            y = tempy;
      }
      if ( m < 1  || m > 12 )
      {
            int tempm = m;
            while ( tempm < 1 || tempm > 12 )
            {
                  cout << "Month " << tempm << " is invalid " << endl;
                  cout << "Please enter a month in the range 1 -12 ";
                  cin >> tempm;
            }
            cout << "Valid m of " << tempm << " accepted" << endl;
            m = tempm;
      }
      if ( d < 1 || d > days_per_month[m - 1] )
      {
            int tempd = d;
            while ( tempd < 1 || tempd > days_per_month[m - 1] )
            {
                  cout << "Day " << tempd << " is invalid " << endl;
                  cout << "Please enter a day in the range 1 - "
                     << days_per_month[m - 1] << " ";
                  cin >> tempd;
            }
            cout << "Valid d of " << tempd << " accepted" << endl;
            d = tempd;
      }

      // once we make it through all the checks assign the fields
      day   = d;               // this->day   = d
      month = m;               // this->month = m
      year  = y;               // this->year  = y
      return;
}
```

Output From Running The Program

```
Welcome to the no arg constructor
In member_print_date
month is 6  day   is 6  year  is 1944

The one argument constructor has been called
In member_print_date
month is 1  day   is 1  year  is 1994

The two argument constructor has been called
In member_print_date
month is 3  day   is 1  year  is 1994

The three argument constructor has been called
In member_print_date
month is 3  day   is 1  year  is 1994
```

One Arg Constructor

Error Checking Loop For Year

Two Arg Constructor

Error Checking Loop For Year

Error Checking Loop For Month

Three Arg Constructor

Error Checking Loop For Year

Error Checking Loop for Month

Error Checking Loop For Days

```
// strctev8.cpp
// Does polymorphism apply only to constructors for a data type?  No.
// This program has the same four constructors as strctev7.cpp but it
// also has three print_date routines
// To save space, I have removed the error checking code for the constructors
// in this application

char * months[12] =
{
	"January", "February", "March", "April", "May", "June", "July",
	"August", "September", "October", "November", "December"
};

int days_per_month[12] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

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

class date
{
	public:
		date();				// no    argument constructor
		date(int);			// one    argument constructor
		date(int,int);			// two   argument constructor
		date(int,int,int);		// three argument constructor

		void print_date();		// function to display a date
		void print_date(int);		// how many times to print date before printing the date
						// also print month instead of month #
		void print_date(char*);	// string to be displayed with date
	private:
		int month;			// DOMAIN	1 - 12
		int day;			// DOMAIN	depends on month
		int year;			// DOMAIN	0 to whatever
};

main()
{
	date d1;			// create a var d1 of type date specifying no args
	d1.print_date( (int) 3);	// call the one int arg print_date func
	d1.print_date();		// call the no arg print_date function
	d1.print_date("Today's date is "); // call the one char* arg func

	date d2(1994);			// create a var d2 of type date specifying one arg
	d2.print_date( (int) 4);	// call the one int arg print_date func
	d2.print_date();		// call the no arg print_date function
	d2.print_date("Au'jour d'hui c'est ");  // call the one char* arg func

	date d3(3,1994);		// create a var d3 of type date specifying two args
	d3.print_date(3);
	d3.print_date();
	d3.print_date("nichi wa ")
```

```
        date d4(3,1,1994);                  // create a var d4 of type date specifying three args
    d4.print_date(7);
    d4.print_date();
    d4.print_date("Hoyte ist ");
}

// return type                      void
// complete function name           date::print_date
// argument list                    ()
void date::print_date()
{
    cout << "In print_date \n";
    cout << "  month is " << this->month <<
            "  day   is " << this->day <<
            "  year  is " << this->year << endl << endl;
}

// return type                      void
// complete function name           date::print_date
// argument list                    (one integer )
void date::print_date(int x)
{
    cout << "In print_date(int) \n";

    for ( int i = 0; i < x; i++ )
            cout << months[this->month -1] << " " << this->day
            << " " << this->year << endl;
}

// return type                      void
// complete function name           date::print_date
// argument list                    ( one pointer to character )
void date::print_date(char * s)
{
    cout << "In print_date(char * ) \n";
    cout << s << months[this->month -1] << " " << this->day
         << " " << this->year << endl << endl;
}

the four date constructors are identical to strctev7.cpp
date::date()
date::date(int y)
date::date(int m, int y)
date::date(int m, int d, int y)
```

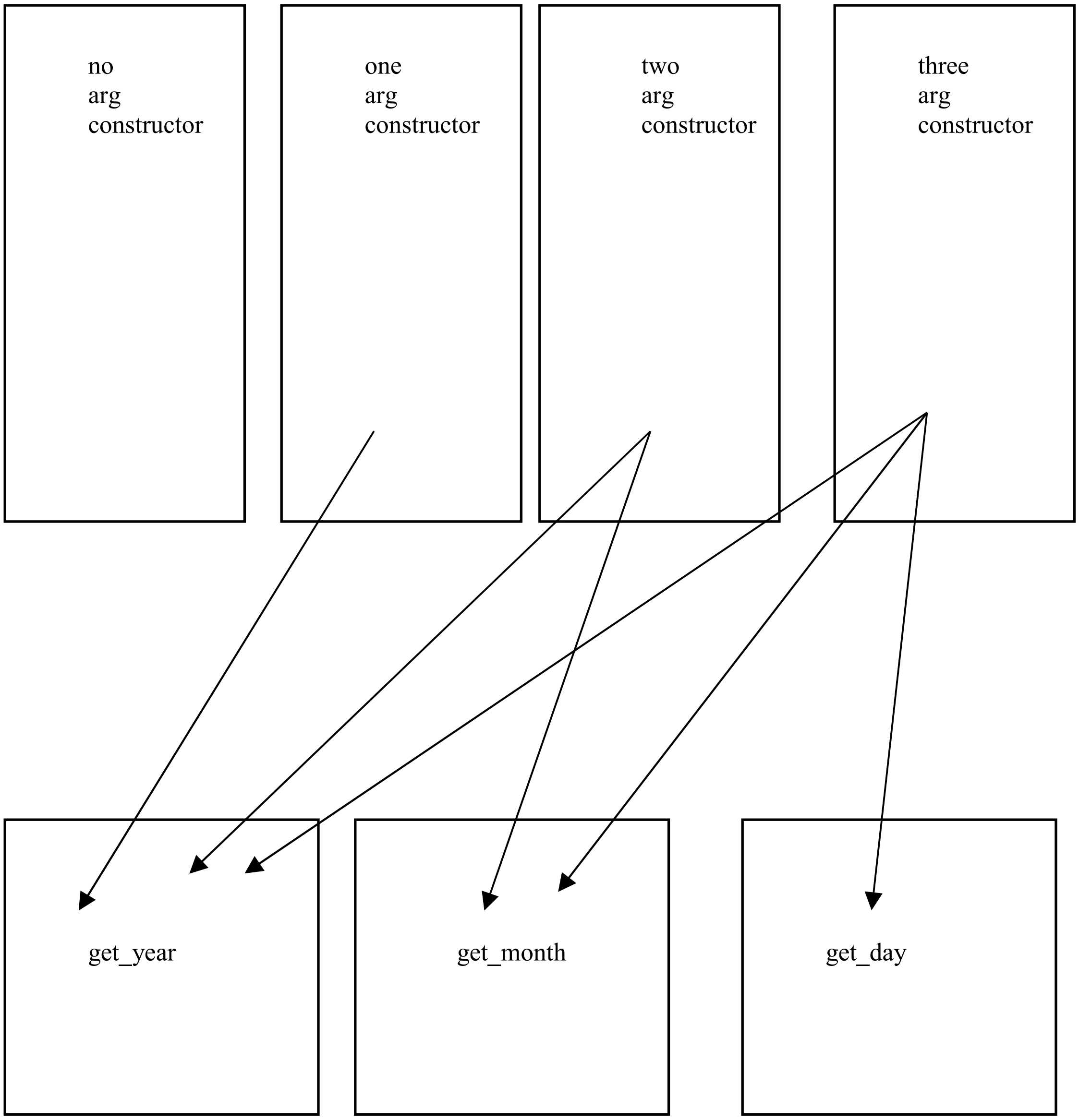
Output From Running Program

```
In print_date(int)
July 22 1961
July 22 1961
July 22 1961
In print_date
month is 7  day   is 22  year  is 1961

In print_date(char * )
Today's date is July 22 1961

In print_date(int)
January 1 1994
January 1 1994
January 1 1994
January 1 1994
In print_date
month is 1  day   is 1  year  is 1994

In print_date(char * )
Au'jour d'hui c'est January 1 1994

In print_date(int)
March 1 1994
March 1 1994
March 1 1994
In print_date
month is 3  day   is 1  year  is 1994

In print_date(char * )
nichi wa March 1 1994

In print_date(int)
March 1 1994
March 1 1994
March 1 1994
March 1 1994
March 1 1994
March 1 1994
March 1 1994
In print_date
month is 3  day   is 1  year  is 1994

In print_date(char * )
Hoyte ist March 1 1994
```

```
// strctev9.cpp
// Each of the constructors in strctev7.cpp did similar error checking
// Wouldn't it make sense to have common routines that could error check for
// the year, month or day fields that any of the constructors could call?
// Also, wouldn't those functions be useful to the set_vals routines?
// Yes.  This program shows that constructors can call subroutines
// Note that these subroutines aren't member functions of the date class
// They are regular subroutines like you are used to from C
// They DO NOT have an object bound to them
// Maybe they SHOULD be member functions.  Does anyone else have any need to call these functions?
// If no one else has a need to call them, then they should be member functions, furthermore,
// they should be PRIVATE member functions that can only be called by other member functions.

int get_year(int);              // accept a year, error check it, return valid year
int get_month(int);             // accept a month, error check it, return valid month
int get_day(int,int);           // accept a day and month, error check day, return

char * months[12] =
{
        "January", "February", "March", "April", "May", "June", "July",
        "August", "September", "October", "November", "December"
};
int days_per_month[12] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>

class date
{
        public:
                date();                 // no    argument constructor
                date(int);              // one   argument constructor
                date(int,int);          // two   argument constructor
                date(int,int,int);      // three argument constructor

                void print_date();      // function to display a date
        private:
                int month;              // DOMAIN       1 - 12
                int day;                // DOMAIN       depends on month
                int year;               // DOMAIN       0 to whatever
};
main()
{
        date d1;                        // no arg constructor invoked
        d1.print_date();
        date d2(1994);                  // call 1 arg constructor, valid values
        d2.print_date();
        date d3(-3,1994);               // call 2 arg constructor, invalid values
        d3.print_date();
        date d4(-3,121,-9);             // call 3 arg constructor, invalid values
        d4.print_date();
}
```

```
void date::print_date()
{
	cout << "  month is " << this->month <<
		"  day   is " << this->day <<
		"  year  is " << this->year << endl << endl;
}

// The three error checking logic loops are made into seperate routines that can be
// called from the constructors, this has the effect of making the overall
// amount of code smaller since the error checking doesn't have to be repeated
// in each of the constructors.  It does make it slower however, because there
// is the additional overhead of additional functional calls from within the

// Does it make sense to have these error checking subroutines visible to the
// user of the class which you are designing?  Perhaps not.
// In the next program, strct11.cpp, We show how these routines can be
// HIDDEN from the casual user but left visible to the class designer

int get_year(int y)
{
	if ( y < 0 )
	{
		int tempy = y;      // create a local variable

		while ( tempy < 0 )
		{
			cout << "Year " << tempy << " is invalid " << endl;
			cout << "Please enter a year greater than 0 ";
			cin  >> tempy;
		}
		cout << "Valid y of " << tempy << " accepted" << endl;
		y = tempy;
	}
	return(y);
}

int get_month(int m)
{
	if ( m < 1  || m > 12 )
	{
		int tempm = m;
		while ( tempm < 1 || tempm > 12 )
		{
			cout << "Month " << tempm << " is invalid " << endl;
			cout << "Please enter a month in the range 1 -12 ";
			cin >> tempm;
		}
		cout << "Valid m of " << tempm << " accepted" << endl;
		m = tempm;
	}
	return(m);
}
```

```
int get_day(int d, int desired_month)
{
	if ( d < 1 || d > days_per_month[desired_month - 1] )
	{
		int tempd = d;
		while ( tempd < 1 || tempd > days_per_month[desired_month - 1])
		{
			cout << "Day " << tempd << " is invalid " << endl;
			cout << "Please enter a day in the range 1 - "
			    << days_per_month[desired_month - 1] << " ";
			cin >> tempd;
		}
		cout << "Valid d of " << tempd << " accepted" << endl;
		d = tempd;
	}
	return d;
}

date::date()
{
	cout <<"Welcome to the no arg constructor " << endl;
	int m = 7, d = 22, y = 1961;
	month = m;	day = d;	year = y;
}

date::date(int y)
{
	cout << "The one argument constructor has been called " << endl;
	year = get_year(y);		// pass in the desired y, get a good one back
	month = 1;			// arbitrarily assign a month
	day = 1;			// arbitrarily assign a day
}

date::date(int m, int y)
{
	cout << "The two argument constructor has been called " << endl;
	year = get_year(y);		// pass the desired y, get a good one back
	month = get_month(m);		// pass the desired m, get a good one back
	day = 1;			// just arbitrarily assign a day
}

date::date(int m, int d, int y)
{
	cout << "The three argument constructor has been called " << endl;
	year = get_year(y);		// use the function that returns valid year
	month = get_month(m);		// use function that returns checked month
	day = get_day(d,month);		// use the error checked month
	return;
}
```


no
arg
constructor
one
arg
constructor
two
arg
constructor
three
arg
constructor
get_year
get_month
get_day

Output From Running Program

```
Welcome to the no arg constructor
month is 7  day   is 22  year  is 1961

The one argument constructor has been called
month is 1  day   is 1  year  is 1994

The two argument constructor has been called
Month -3 is invalid
Please enter a month in the range 1 -12 Valid m of 5 accepted
month is 5  day   is 1  year  is 1994

The three argument constructor has been called
Year -9 is invalid
Please enter a year greater than 0 Valid y of 1994 accepted
Month -3 is invalid
Please enter a month in the range 1 -12 Month 23 is invalid
Please enter a month in the range 1 -12 Month -9 is invalid
Please enter a month in the range 1 -12 Month 24 is invalid
Please enter a month in the range 1 -12 Valid m of 12 accepted
Day 121 is invalid
Please enter a day in the range 1 - 31 Valid d of 23 accepted
month is 12  day   is 23  year  is 1994
```

```cpp
// destrct1.cpp
// this program illustrates the destructor
// you write a destructor when you want to control how memory is deallocated for a class you created
// Most of the time, the default deallocation is fine
// However, if your class is pointing to a shared page of memory,
// there are issues concerning whether the deallocation of this object
// should cause the deallocation of the shared page or should only the
// deallocation of the last object of that class type cause the deallocation
// of the page.  There are other issues as well.

#include <iostream.h>

class abc
{
	private:
		int a,b,c;
	public:
		abc();	// the constructor function, name same as class
		~abc(); // the destructor function   notice the    ~
};

// This is the constructor function, its name matches the name of the class
abc::abc()
{
	cout << "The no arg constructor was invoked\n";
	a = b = c = 1;
}

// This is the destructor function, its name matches the name of the class and is preceeded
// by a tilde    ~
abc::~abc()
{
	cout << "The no arg DESTRUCTOR was invoked \n";

	// I'm not doing anything special because I'm not pointing at
	// anything shared, I will just allow the system to deallocate
	// the three integers, a,b,c in whatever way it sees fit
}

main()
{
	abc a1;		// this will cause the constructor to be called

}	 // having the program go out of scope will cause the destructor
	// to be invoked.  We VERY RARELY call the destructor functions
	// we just let them happen when they're supposed to happen, i.e.
	// when a variable or object goes out of scope

Output From Running Program

The no arg constructor was invoked			constructor called when var created
The no arg DESTRUCTOR was invoked			destructor called when var goes
							out of scope
```

```
// destrct2.cpp
// this program shows multiple scopes and when things go out of scope
#include <iostream.h>

class abc
{
	private:
		int a,b,c;
	public:
		abc();	// the constructor function, name same as class
		~abc(); // the destructor function   notice the    ~
};

// constructor function, called when instances of abcs are created
abc::abc()
{
	cout << "The no arg constructor was invoked\n";
	a = b = c = 1;
}

// destructor function, called when instances of abcs are destroyed, mostly by going out of scope
abc::~abc()
{
	cout << "The no arg DESTRUCTOR was invoked \n";

		// I'm not doing anything special because I'm not pointing at
		// anything shared, I will just allow the system to deallocate
		// the three integers, a,b,c in whatever way it sees fit
		// maybe in your program, you care about what order they are deleted in or
		// maybe you want to dump the values to an audit file before destroying them
		// that's the point of a destructor, you can do things when variables are being destroyed
}
```

```
main()
{
	cout << "Before abc a1";
	abc a1;		// this will cause the constructor to be called
	cout << "After abc a1\n";

	{
		cout << "\tA new scope has started\n";
		cout << "\tBefore abc a2  ";
		abc a2;
		cout << "\tAfter abc a2  ";
	}	// a2 will go out of scope here, the no arg destructor will be called
	cout << "\tThe scope with a2 just exited\n";

	{
		cout << "\tAnother new scope just started\n";
		cout << "\tBefore abc a3  ";
		abc a3;
		cout << "\tAfter abc a3  ";

		{
			cout << "\t\tYet another new scope started\n";
			cout << "\t\tBefore abc a4  ";
			abc a4;
			cout << "\t\tAfter abc a4  ";
		}		// at this point, a4 will go out of scope, destructor will be invoked
		cout << "\t\tInnermost scope ended\n";
	}	// at this point, a3 will go out of scope, destructor will be invoked
	cout << "\tOther scope ended\n";
	cout << "Program about to end\n";
}		// at this point, a1 will go out of scope, destructor will be invoked
```

Output From Running Program

```
Before abc a1	The no arg constructor was invoked
After abc a1
	A new scope has started
	Before abc a2	The no arg constructor was invoked
	After abc a2	The no arg DESTRUCTOR was invoked
	The scope with a2 just exited
	Another new scope just started
	Before abc a3	The no arg constructor was invoked
	After abc a3			Yet another new scope started
		Before abc a4	The no arg constructor was invoked
		After abc a4	The no arg DESTRUCTOR was invoked
		Innermost scope ended
		The no arg DESTRUCTOR was invoked
	Other scope ended
Program about to end
The no arg DESTRUCTOR was invoked
```

```
// destrct3.cpp
// this program illustrates the destructor and arrays

#include <iostream.h>

class abc
{
	private:
		int a,b,c;
	public:
		abc();		// the constructor function, name same as class
		~abc();		// the destructor function   notice the    ~
};
abc::abc()			// no argument constructor
{
	cout << "The no arg constructor was invoked\n";
	a = b = c = 1;
}
abc::~abc()			// no argument destructor
{
	cout << "The no arg DESTRUCTOR was invoked \n";
}

main()
{
	cout << "Before abc a1[5] \n";
	abc a1[5];		// how many times is constructor called?
	cout << "After abc a1[5] \n\n";

	{
		cout << "Before abc a2[3] \n";
		abc a2[3];
	}   // at this point, a2 goes out of scope, how many times is destructor invoked?
	cout << "After a2 has gone out of scope\n\n";
}  // at this point, a1 goes out of scope, how many times is destructor invoked?
```

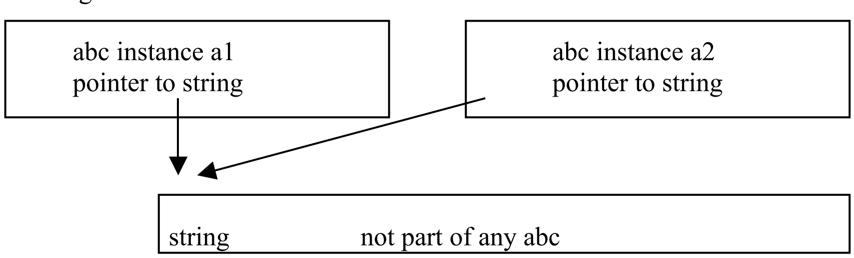
```
Output From Running Program

Before abc a1[5]
The no arg constructor was invoked
The no arg constructor was invoked
The no arg constructor was invoked
The no arg constructor was invoked
The no arg constructor was invoked
After abc a1[5]

Before abc a2[3]
The no arg constructor was invoked
The no arg constructor was invoked
The no arg constructor was invoked
The no arg DESTRUCTOR was invoked
The no arg DESTRUCTOR was invoked
The no arg DESTRUCTOR was invoked
After a2 has gone out of scope

The no arg DESTRUCTOR was invoked
The no arg DESTRUCTOR was invoked
The no arg DESTRUCTOR was invoked
The no arg DESTRUCTOR was invoked
The no arg DESTRUCTOR was invoked
```

```
// destrct4.cpp
// this program illustrates a practical use of a destructor
// We will have many instances of a class abc which are all pointing at one character string
// In the destructor, in this case, we will not delete the memory that is being pointed at
// We will let the system delete the three integers a,b,c
// We will let the system delete the integer size_of_string
// We will let the system delete the char * cptr
// We will then examine the memory pointed to by cptr to see if it still has the string in it

#include <iostream.h>

char s[] = { "This is the shared string they point at\n" };

class abc
{
	private:
		int a,b,c;
		int size_of_string;
		char * cptr;		// will point to shared character string
	public:
		abc();			// the constructor function, name same as class
		~abc();			// the destructor function   notice the    ~
		void print_abc();
};

abc::abc()
{
	cout << "The no arg constructor was invoked\n";
	a = b = c = 1;
	cptr = &s[0];
	size_of_string = sizeof(s);	// get the size stored
}

abc::~abc()
{
	cout << "The no arg DESTRUCTOR was invoked \n";
	// At this point I could do something to s if I wanted to
	// For this application I won't.  I will leave it alone
	// In destrct5.cpp and destrct6.cpp, I will do things to s
}

void abc::print_abc()
{
	cout << "In print_abc values are: \n";
	cout << a << "  " << b << " " << c << endl;
	cout.write(cptr,size_of_string);
}
```

```
main()
{
      // create an abc, it will have a pointer to s in it
      cout << "Before abc a1 s has " << s << endl;
      abc a1;                                         // this will cause the constructor to be called
      cout << "After  abc a1 s has " << s << endl;
      a1.print_abc();

      {
            // create another abc, it will also have a pointer to s
            cout << "Before abc a2 s has " << s << endl;
            abc a2;                                   // this will cause the constructor to be called
            cout << "After  abc a2 s has " << s << endl;
            a2.print_abc();
      }     // cause a2 to go out of scope, the destructor will be called
            // a1 still exists and presumably still points at the character string

      cout << "After a2 has gone out of scope, s has " << s << endl;
      a1.print_abc(); // see if a1 is still okay
}           // at this point, a1 will go out of scope, we still don't do anything to the string s
            // the string s was just something the instances of abc's were pointing at
            // the string s was not part of abc instances
            // so in this case it would make sense to delete the pointer to the string but not the string
```

Output From Running Program

```
Before abc a1 s has This is the shared string they point at
The no arg constructor was invoked
After  abc a1 s has This is the shared string they point at

In print_abc values are:
1  1 1
This is the shared string they point at
Before abc a2 s has This is the shared string they point at
The no arg constructor was invoked
After  abc a2 s has This is the shared string they point at

In print_abc values are:
1  1 1
This is the shared string they point at
The no arg DESTRUCTOR was invoked
After a2 has gone out of scope, s has This is the shared string they point at
In print_abc values are:
1  1 1
This is the shared string they point at
The no arg DESTRUCTOR was invoked
```



```
// destrct5.cpp
// In this version of the destructor program I am going to manipulate the string that the abc
// instances point at.  This may be a dumb thing to do, but the point is who is in control, not whether
// what you do is smart!!
#include <iostream.h>
#include <string.h>

char s[] = { "This is the shared string they point at\n" };

class abc
{
	private:
		int a,b,c;
		char * cptr;		// will point to shared character string
	public:
		abc();			// the constructor function, name same as class
		~abc();			// the destructor function   notice the    ~
		void print_abc();
};

abc::abc()
{
	cout << "The no arg constructor was invoked\n";
	a = b = c = 1;
	cptr = &s[0];
}

abc::~abc()
{
	cout << "The no arg DESTRUCTOR was invoked \n";
	// Now I am going to manipulate the shared string
	// This may be a stupid thing to do, but that's the point
	// When you write the destructor, you control what happens when
	// your objects are deallocated
	strcpy(cptr,"NEW STRING FROM DESTRUCTOR\n");
}

void abc::print_abc()
{
	cout << "In print_abc values are: \n";
	cout << a << "  " << b << " " << c << endl;
	cout.write(cptr,sizeof(s) );
}
```

```
main()
{
	// create an abc, it will have a pointer to s in it
	cout << "Before abc a1 s has " << s << endl;
	abc a1;					// this will cause the constructor to be called
	cout << "After  abc a1 s has " << s << endl;
	a1.print_abc();

	{
		// create another abc, it will also have a pointer to s
		cout << "Before abc a2 s has " << s << endl;
		abc a2;				// this will cause the constructor to be called
		cout << "After  abc a2 s has " << s << endl;
		a2.print_abc();
	}	 // cause a2 to go out of scope, this will cause destructor to be invoked
	    // remember, in the destructor, I have decided to manipulate s

	cout << "After a2 has gone out of scope, s has " << s << endl;

	a1.print_abc();				// see if a1 is still okay
}

Output From Running Program

Before abc a1 s has This is the shared string they point at

The no arg constructor was invoked
After  abc a1 s has This is the shared string they point at

In print_abc values are:
1  1 1
This is the shared string they point at

Before abc a2 s has This is the shared string they point at

The no arg constructor was invoked
After  abc a2 s has This is the shared string they point at

In print_abc values are:
1  1 1
This is the shared string they point at

The no arg DESTRUCTOR was invoked
After a2 has gone out of scope, s has NEW STRING FROM DESTRUCTOR

In print_abc values are:
1  1 1
NEW STRING FROM DESTRUCTOR			notice that a1 points to the start of the string
ey point at					but it's length field wasn't manipulated
						so left over stuff was included in the string
The no arg DESTRUCTOR was invoked
```

```
// destrct6.cpp
// In this program I am going to have a static variable keep track of how many instances are in
// existence.  When the last instance goes out of scope, I am then and only then going to manipulate
// the string that the instances of abcs are pointing at.  This of course begs the question, "What are
// you going to do if someone creates ANOTHER instance of an abc?"   Well... The, um, uh
// Oh Yeah, the static variable would have the value zero and I could check for that in the constructor
// and do something to the string area.  Whew.  Got out of that one.

#include <iostream.h>
#include <string.h>

char s[] = { "This is the shared string they point at\n" };

class abc
{
	private:
		static int count_of_active_objects;		// static variable
								// used to track how many of this type of object exist
		int a,b,c;
		char * cptr;					// will point to shared character string
	public:
		abc();					// the constructor function, name same as class
		~abc();					// the destructor function   notice the    ~
		void print_abc();
};

abc::abc()
{
	cout << "The no arg constructor was invoked\n";
	a = b = c = 1;
	cptr = &s[0];
	count_of_active_objects++;			// add one to the count stored in the static variable
}

abc::~abc()
{
	cout << "The no arg DESTRUCTOR was invoked \n";
	// only manipulate if you are the last one alive
	if ( count_of_active_objects == 1 )
	{
		strcpy(cptr,"NEW STRING FROM DESTRUCTOR\n");
	}
	else
	{
		cout << "I am not last suriving object\n";
		cout << "I am not going to hurt s \n";
	}
	count_of_active_objects--;			// decrement number of living objects
}
```

```
void abc::print_abc() {
	cout << "In print_abc values are: \n";
	cout << a << "  " << b << " " << c << endl;
	cout.write(cptr,sizeof(s) );
}

int abc::count_of_active_objects = 0;		// initialize the static variable

main() {
	// create an abc, it will have a pointer to s in it
	cout << "Before abc a1 s has " << s << endl;
	abc a1;					// this will cause the constructor to be called
	cout << "After  abc a1 s has " << s << endl;
	a1.print_abc();
	{
		// create another abc, it will also have a pointer to s
		cout << "Before abc a2 s has " << s << endl;
		abc a2; // this will cause the constructor to be called
		cout << "After  abc a2 s has " << s << endl;
		a2.print_abc();
	}	// cause a2 to go out of scope
	cout << "After a2 has gone out of scope, s has " << s << endl;
	a1.print_abc();				// see if a1 is still okay
}
```

Output From Running Program

```
Before abc a1 s has This is the shared string they point at

The no arg constructor was invoked
After  abc a1 s has This is the shared string they point at

In print_abc values are:
1  1 1
This is the shared string they point at
Before abc a2 s has This is the shared string they point at

The no arg constructor was invoked
After  abc a2 s has This is the shared string they point at

In print_abc values are:
1  1 1
This is the shared string they point at
The no arg DESTRUCTOR was invoked
I am not last suriving object
I am not going to hurt s
After a2 has gone out of scope, s has This is the shared string they point at

In print_abc values are:
1  1 1
This is the shared string they point at
The no arg DESTRUCTOR was invoked
```

```
// destrct7.cpp
// this program also uses a static variable to manipulate the string being pointed at
// However, along with having instances of abcs pointing at the string, I will also attempt to have
// non class variables pointing at and using the string.  So in this case, maybe the class destructor
// should have kept it's grubby paws off the string eh??

#include <iostream.h>
#include <string.h>

char s[] = { "This is the shared string they point at\n" };

class abc
{
	private:
		static int count_of_active_objects; // static variable used to track
							// how many of this type of object exist
		int a,b,c;
		char * cptr;				// will point to shared character string
	public:
		abc();					// the constructor function, name same as class
		~abc();					// the destructor function   notice the   ~
		void print_abc();
};

abc::abc()
{
	cout << "The no arg constructor was invoked\n";
	a = b = c = 1;
	cptr = &s[0];
	count_of_active_objects++;	// add one to the count
}

abc::~abc()
{
	cout << "The no arg DESTRUCTOR was invoked \n";
	// only manipulate if you are the last one alive
	if ( count_of_active_objects == 1 )
	{
		strcpy(cptr,"NEW STRING FROM DESTRUCTOR\n");
	}
	else
	{
		cout << "I am not last suriving object\n";
		cout << "I am not going to hurt s \n";
	}
	count_of_active_objects--;			// decrement number of living objects
}
void abc::print_abc()
{
	cout << "In print_abc values are: \n";
	cout << a << "  " << b << " " << c << endl;
	cout.write(cptr,sizeof(s) );
}
```

```
int abc::count_of_active_objects = 0;       // initialize the static variable

main()
{
        {
                // create an abc, it will have a pointer to s in it
                cout << "Before abc a1 s has " << s << endl;
                abc a1;                         // this will cause the constructor to be called
                cout << "After  abc a1 s has " << s << endl;
                a1.print_abc();

                {
                        // create another abc, it will also have a pointer to s
                        cout << "Before abc a2 s has " << s << endl;
                        abc a2;                 // this will cause the constructor to be called
                        cout << "After  abc a2 s has " << s << endl;
                        a2.print_abc();
                }                               // cause a2 to go out of scope
                cout << "After a2 has gone out of scope, s has " << s << endl;

                a1.print_abc();                 // see if a1 is still okay
        }
        // at this point, a1 and a2 have gone out of scope, there are no abc instances left
        // so can anyone else use s?  Or has the selfish abc class ruined it for everyone?
        cout  << "After a1 has gone out of scope, s has " << s << endl;
}
```

Output From Running Program

```
Before abc a1 s has This is the shared string they point at

The no arg constructor was invoked
After  abc a1 s has This is the shared string they point at
In print_abc values are:
1  1 1
This is the shared string they point at
Before abc a2 s has This is the shared string they point at

The no arg constructor was invoked
After  abc a2 s has This is the shared string they point at

In print_abc values are:
1  1 1
This is the shared string they point at
The no arg DESTRUCTOR was invoked
I am not last suriving object
I am not going to hurt s
After a2 has gone out of scope, s has This is the shared string they point at

In print_abc values are:
1  1 1
This is the shared string they point at
The no arg DESTRUCTOR was invoked
After a1 has gone out of scope, s has NEW STRING FROM DESTRUCTOR
```

```cpp
// ch1p4.cpp
// program to illustrate role of public, protected, private in function
// visibility.  There are two places to check, within member function code
// and outside of member function code, i.e. in main()

#include <iostream.h>
#include <stdio.h>

class abc
{
	public:
		void pubf1();
		void pubf2();
		abc();
	protected:
		void protf1();
		void protf2();
	private:
		void privf1();
		void privf2();
};

abc::abc() { cout << "No arg constructor invoked \n"; }

// from inside the member functions of a class you are able to call the
// public, protected and private member functions of that class
// PUBLIC FUNCTION
void abc::pubf1()
{
	cout << "In pubf1() \n";
	this->pubf2();  // try to call a public function, you will succeed
	this->protf2(); // try to call a protected function, will succeed
	this->privf2(); // try to call a private function, will succeed
}
// PROTECTED FUNCTION
void abc::protf1()
{
	cout << "In protf1() \n";
	this->pubf2();  // try to call a public function, you will succeed
	this->protf2(); // try to call a protected function, will succeed
	this->privf2(); // try to call a private function, will succeed
}
// PRIVATE FUNCTION
void abc::privf1()
{
	cout << "In privf1() \n";
	this->pubf2();  // try to call a public function, will succeed
	this->protf2(); // try to call a protected function, will succeed
	this->privf2(); // try to call a private function, will succeed
}
```

```
// Public Function
void abc::pubf2()
{
	cout << "IN PUBf2() \n";
}
// Protected Function
void abc::protf2()
{
	cout << "IN PROTf2() \n";
}
// Private Function
void abc::privf2()
{
	cout << "IN PRIVf2() \n";
}

// from non member function code, i.e. inside main
// you MAY CALL the public member functions of a class
// you MAY NOT CALL the protected or private member functions of a class
main()
{
	abc a1;				// create an instance of an abc
	a1.pubf1();			// try to call the public function
	cout << endl << endl;	// endl is a constant that == "\n"

	// this line will not compile
	// if you uncomment it you will see compiler error message
	// abc::protf1() is not accessible in function main()
	// a1.protf1();			// try to call the protected function

	// this line will not compile
	// if you uncomment it you will see compiler error message
	// abc::privf1() is not accessible in function main()
	// a1.privf1();			// try to call the private function

	a1.pubf2();			// try to call the other public function

	// this line will not compile
	// if you uncomment it you will see compiler error message
	// abc::protf2() is not accessible in function main()
	// a1.protf2();			// try to call the other protected function

	// this line will not compile
	// if you uncomment it you will see compiler error message
	// abc::privf2() is not accessible in function main()
	// a1.privf2();			// try to call the other private function
}
```

```
Output From Running Program

No arg constructor invoked
In pubf1()
IN PUBf2()
IN PROTf2()
IN PRIVf2()


IN PUBf2()
```

public
    pubf1()
    pubf2()

protected
    protf1()
    protf2()

private
    privf1()
    privf2()

public
member
function

can it access...
public fn? Yes
protected fn? YES
private fn? YS

protected
member
function

can it access...
public fn? YES
protected fn? YES
private fn? YES

private
member
function

can it access...
public fn? YS
protected fn? YES
private fn? YES

main

public function? YES
protected function? NO
private function? NO

```
// strct10.cpp
// In this program, the error checking routines of strctev9.cpp are made
//  PRIVATE MEMBER FUNCTIONS of the class
// this means that they will be visible only to the designer of the class
// Since it is the designer of the class who is carefully coding in the
// error checking properties of the class, it is the designer of the class
// who needs to be able to call the routines.  The user of the date class
// has no need to know or care that these routines even exist.
// This reduces the number of subroutines the user of the class needs
// to concern themselves with.  This reduces complexity for the class user

char * months[12] =
{
	"January", "February", "March", "April", "May", "June", "July",
	"August", "September", "October", "November", "December"
};

int days_per_month[12] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

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

class date
{
	public:
		date();			// no    argument constructor
		date(int);		// one    argument constructor
		date(int,int);		// two   argument constructor
		date(int,int,int);	// three argument constructor

		void print_date();	// function to display a date

	private:
		int month;		// DOMAIN	1 - 12
		int day;		// DOMAIN	depends on month
		int year;		// DOMAIN	0 to whatever

		// These are now PRIVATE MEMBER FUNCTIONS
		int get_year(int);// private member functions
		int get_month(int);	// may be called from member funcs
		int get_day(int,int);	// may not be called from main()
};
```

```cpp
main()
{
	date d1;			// create a var d1 of type date specifying no args
	d1.print_date();

	date d2(-4);			// create a var d2 of type date specifying one arg
	d2.print_date();

	date d3(23,1994);		// create a var d3 of type date specifying two args
	d3.print_date();

	date d4(3,32,1994);		// create a var d4 of type date specifying three args
	d4.print_date();
}

void date::print_date()
{
	cout << "  month is " << this->month <<   "  day   is " << this->day <<
		"  year  is " << this->year << endl << endl;
}

// in the previous program the header was
// int get_year(int y)
// the header changes because now the routine is a member function and
// the complete function name includes the class name

return	class	function	argument
type	name	name		list

int date::get_year(int y)
{
	if ( y < 0 )
	{
		int tempy = y;		// create a local variable

		while ( tempy < 0 )
		{
			cout << "Year " << tempy << " is invalid " << endl;
			cout << "Please enter a year greater than 0 ";
			cin  >> tempy;
		}
		cout << "Valid y of " << tempy << " accepted" << endl;
		y = tempy;
	}
	return(y);
}
```

```
int date::get_month(int m)
{
	if ( m < 1  || m > 12 )
	{
		int tempm = m;
		while ( tempm < 1 || tempm > 12 )
		{
			cout << "Month " << tempm << " is invalid " << endl;
			cout << "Please enter a month in the range 1 - 12 ";
			cin >> tempm;
		}
		cout << "Valid m of " << tempm << " accepted" << endl;
		m = tempm;
	}
	return(m);
}

int date::get_day(int d, int desired_month)
{
	if ( d < 1 || d > days_per_month[desired_month - 1] )
	{
		int tempd = d;
		while ( tempd < 1 || tempd > days_per_month[desired_month - 1])
		{
			cout << "Day " << tempd << " is invalid " << endl;
			cout << "Please enter a day in the range 1 - "
			    << days_per_month[desired_month - 1] << " ";
			cin >> tempd;
		}
		cout << "Valid d of " << tempd << " accepted" << endl;
		d = tempd;
	}
	return d;
}
```

```
// The four constructors are the same as in the last program
// But now they are going to use the Private Member Functions to do the error checking
// Instead of external error checking routines
date::date()
{
	cout <<"Welcome to the no arg constructor " << endl;
	int m = 7, d = 22, y = 1961;
	month = m;	day = d;	year = y;
}

date::date(int y)
{
	cout << "The one argument constructor has been called " << endl;
	// call the private member function get_year to do the error checking
	// this constructor is a member function, so it can call the private
	// member function get_year
	year = get_year(y);
	month = 1;	// arbitrarily assign a month
	day = 1;	// arbitrarily assign a day
}

date::date(int m, int y)
{
	cout << "The two argument constructor has been called " << endl;
	// call the private member functions get_year and get_month from
	// this public member function date(int,int)
	year = get_year(y);
	month = get_month(m);
	day = 1;
}

date::date(int m, int d, int y)
{
	cout << "The three argument constructor has been called " << endl;
	// call the private member functions get_year, get_month, get_day
	// from this public member function date(int,int,int)
	year = get_year(y);
	month = get_month(m);
	day = get_day(d,month);
	return;
}
```

```
Output From Running Program

Welcome to the no arg constructor
 month is 7  day   is 22  year  is 1961

The one argument constructor has been called
Year -4 is invalid
Please enter a year greater than 0 Valid y of 45 accepted
 month is 1  day   is 1  year  is 45

The two argument constructor has been called
Month 23 is invalid
Please enter a month in the range 1 - 12 Month -8 is invalid
Please enter a month in the range 1 - 12 Month 14 is invalid
Please enter a month in the range 1 - 12 Valid m of 7 accepted
 month is 7  day   is 1  year  is 1994

The three argument constructor has been called
Day 32 is invalid
Please enter a day in the range 1 - 31 Day 34 is invalid
Please enter a day in the range 1 - 31 Valid d of 2 accepted
 month is 3  day   is 2  year  is 1994
```

```
// strct11.cpp
// Constructors are good for putting INITIAL values into an instance of a date
// struct variable, but what if I want to CHANGE the values during run-time?
// We have already seen a routine that could put values into all the fields
// of an instance of the class
// What if we want to just put a value into one field at a time?

// Can we do it?
// Yes.

// We know that we cannot directly address the private data elements of a
// variable of type date struct from anyplace except a member function,
// therefore we are going to add member functions to set the data elements.
// We will add routines to set any individual field or to set all three
// fields at the same time.
// The set_year, set_month, and set_day functions will also be able to
// use the error_checking routines for the struct date data type

char * months[12] =
{
	"January", "February", "March", "April", "May", "June", "July",
	"August", "September", "October", "November", "December"
};

int days_per_month[12] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>

class date
{
	public:
		date();				// no    argument constructor
		date(int);			// one    argument constructor
		date(int,int);			// two   argument constructor
		date(int,int,int);		// three argument constructor
		void print_date();		// function to display a date
	       // These set routines will use the private member functions that do the error checking
		void set_year(int);		// function to change the year value
		void set_month(int);		// function to change the month value
		void set_day(int);		// function to change the day value
		void set_all_three(int,int,int);	// function to set all three
	private:
		int month;			// DOMAIN	1 - 12
		int day;			// DOMAIN	depends on month
		int year;			// DOMAIN	0 to whatever

		int get_year(int);// private member functions
		int get_month(int);		// to do the error checking routines
		int get_day(int,int);

};

```

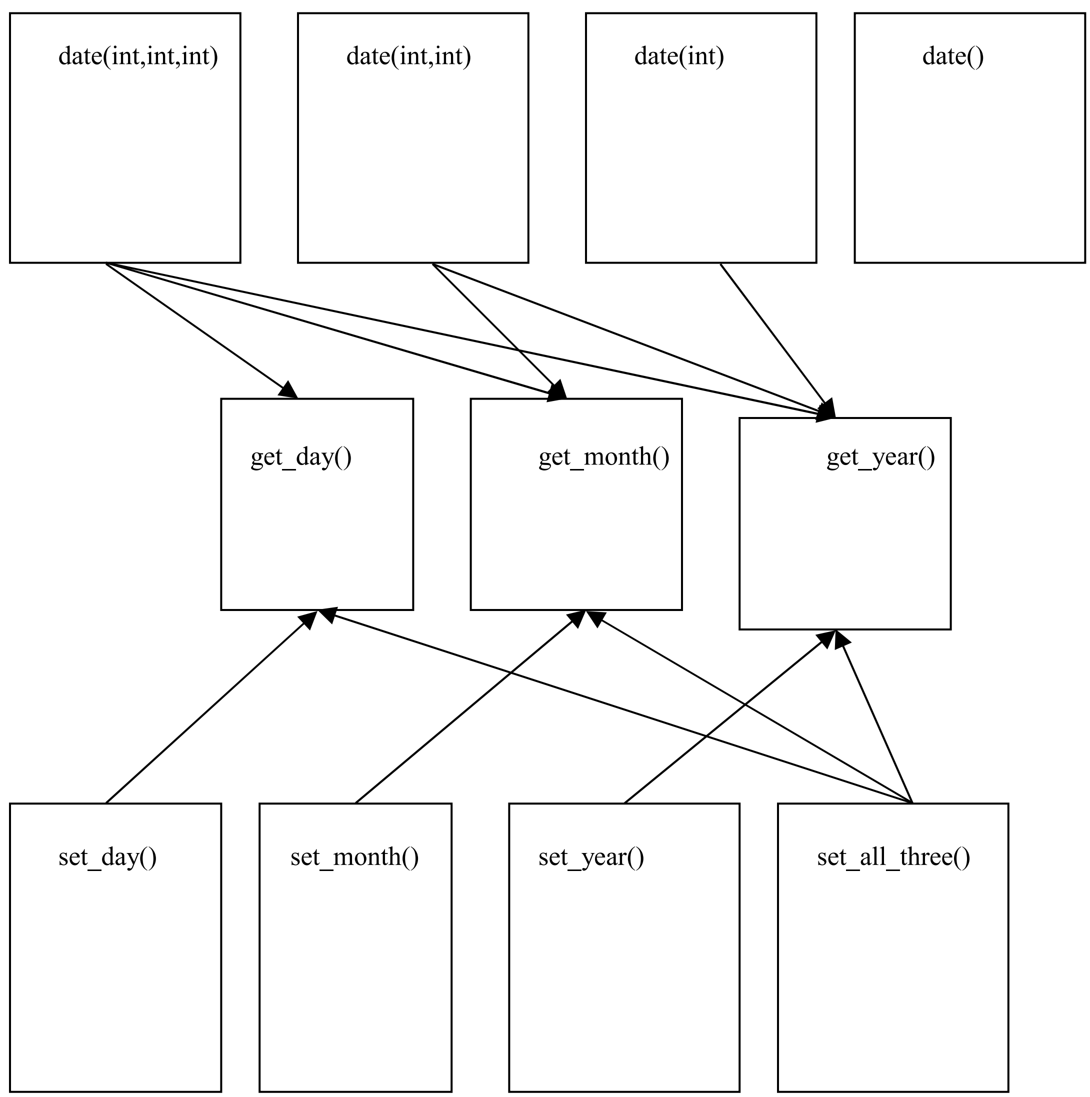
```
main()
{
	date d1;			// create a var d1 of type date specifying no args
	d1.print_date();

	d1.set_year(1993);		// valid call with valid month
	d1.print_date();

	d1.set_month(2);		// valid call with valid month
	d1.print_date();

	d1.set_day(33);			// this will cause error checking to happen
	d1.print_date();

	d1.set_all_three(12,25,1994);// call to set all three, all three are valid data, error checks will pass
	d1.print_date();

	d1.set_all_three(-3,89,-2);	// call to set all three, three error checks will occur
	d1.print_date();
}

void date::print_date()
{
	cout << "  month is " << this->month << "  day   is " << this->day <<
		"  year  is " << this->year << endl << endl;
}


// The get_year, get_month, get_day functions are the same as the last program
// The four constructors are the same as the last program
```

```
// all this subroutine has to do is call the get_year function
void date::set_year(int y)
{
        year = get_year(y);
        return;
}

// all this subroutien has to do is call the get_month function
void date::set_month(int m)
{
        month = get_month(m);
        return;
}

// all this subroutine has to do is call get_day
void date::set_day(int d)
{
    // we need to already have a valid month to be able to check the day
    // field of the struct date type variable that was BOUND to this call
    day = get_day(d,this->month);
    return;
}

// this subroutine will call each function in turn

void date::set_all_three(int m, int d, int y )
{
    year = get_year(y);
    month = get_month(m);
    day   = get_day(d,month);
    return;
}
```

Output From Running Programm

```
 month is 7  day   is 22  year  is 1961
 month is 7  day   is 22  year  is 1993
 month is 2  day   is 22  year  is 1993
Day 33 is invalid
Please enter a day in the range 1 - 28 Valid d of 23 accepted
 month is 2  day   is 23  year  is 1993
 month is 12  day   is 25  year  is 1994
Year -2 is invalid
Please enter a year greater than 0 Valid y of 1994 accepted
Month -3 is invalid
Please enter a month in the range 1 - 12 Month 89 is invalid
Please enter a month in the range 1 - 12 Month 0 is invalid
Please enter a month in the range 1 - 12 Valid m of 7 accepted
Day 89 is invalid
Please enter a day in the range 1 - 31 Valid d of 22 accepted
 month is 7  day   is 22  year  is 1994
```

date class
Private member functions
get_year
get_month
get_day
Public Member Functions
date() date(int) date(int,int) date(int,int,int)
set_year(int), set_month(int), set_dayint), set_all_three(int,int,int)



Section 4

This chapter is about user defined data structures and operators.
C++ allows you to "take over" what operators mean for your defined classes.
We are used to having to use subroutines to perform operations on our data structures.
This is bad. It introduces new subroutine names into the programmers name space without providing any new functionality. For example, if you make a data structure to model someone's age, you would probably want, at some point in time, to be able to add one to their age. In traditional languages you will be forced to write a subroutine and use the subroutine by name to add one to your data structure. In C++, you will still have to write the subroutine, however you will not have to call it by some ridiculous name like add_one_to_age. Instead you will be able to say

```
Age a1;
...
a1++;
```

This will be possible because you will be able to "take over," or overload, the operators for that class.
Taking over the operators for one class in no way affects how the operators behave for other classes.

You are able to take over:

- relational operators
- mathematical operators
- i/o operators.

Again, to visit our age class, if you wanted to compare two people's ages, in a non C++ language you would probably have to make some call like:

```
lower = compare_ages(a1,a2);
```

This is stupid.
In C++ you will be able to say

```
if ( a1 < a2 )
```

or any other relational operator that you have overloaded because you can take over the operators and redefine what they mean for your class.

The point of overloading operators is this:

- You are able to use your classes using the syntax you already know.
- You don't have to learn a gazillion (gumpism) new names to use a class.
- The amount of work that is done by the computer at run time is the same,
  - a subroutine is still called and run to do the work,
  - However, the programmer's job was made easier.

Section4.doc

relops.cpp	regular C relational operators and their limitations
relops1.cpp	relational operations on structures, the serious syntactical limitations
relops2.cpp	relational operations via operators on C++ classes
mathfunc.cpp	math operations on structures, the serious syntactical limitations
mathops.cpp	math operations via operators on C++ classes
mathops1.cpp	polymorphic math operators
ioops.cpp	taking over the i/o operators for a class

```
// relops.cpp
// C and C++ provide the programmer with a number of operators
// = == + += - -= * *= / /= < <= > >= !=  and so on
// these operators only have meaning to pre-defined types
// the only operator that has any meaning, by default, for user defined types
// is the = operator
// If the = operator is applied between two instances of a user defined type
// then a field wise copy will occur
// This is a serious limitation of C
// For example, if you have a data structure like we have been using
// to model the date, wouldn't it make sense to be able to say
// if ( d1 < d2 )     or          if ( d1 >= d2 )

// In C, a user is forced to write subroutines like
// if ( compare_days(d1,d2) )
// and introduce new names into the programmers name space

// The programmer has already spent the time mastering the C operators
// Can't we leverage their investment into user defined types? In C++ we can!

// This program is a simple review of C operators
// It illustrates how simple it is to use the relational operators for a  defined type

#include <iostream.h>
int main()
{
        int i = 0, j = 1, k = 1;

        if ( i == j )
                cout << "i is equivalent to j\n";

        if ( i != j )
                cout << "i is not equivalent to j\n";

        if ( i < j )
                cout << "i is less than j\n";

        if ( i <= j )
                cout << "i is less than or equal to j\n";

        if ( i > j )
                cout << "i is greater than j\n";

        if ( i >= j )
                cout << "i is greater than or equal to j\n";
}

Output From Running Program
i is not equivalent to j
i is less than j
i is less than or equal to j
```

C Relational Operators

int	<	int
int	<=	int
int	>=	int
int	>	int
int	==	int
int	!=	int
float	<	float
float	<=	float
float	>=	float
float	>	float
float	==	float
float	!=	float
char	<	char
char	<=	char
char	>=	char
char	>	char
char	==	char
char	!=	char
struct	no relational operators	struct

```
// relops1.cpp
// this program declares a simple time struct with three fields
// for the sake of listing compactness, error checking is left out
// this program demonstrates what a user would have to go through
// to be able to compare two instances of a time struct in the standard six relational ways

#define TRUE     1
#define FALSE    0

#include <iostream.h>

struct time
{
	int h,m,s;		// hours, minutes, seconds
};

// each of these routines returns TRUE if the condition is met
// each of these routines returns FALSE if the condition is not met
// NOTE that each of these functions needs two arguments
// relational operations are binary operations
int time_eq(struct time, struct time);
int time_ne(struct time, struct time);
int time_gt(struct time, struct time);
int time_ge(struct time, struct time);
int time_lt(struct time, struct time);
int time_le(struct time, struct time);

// to implement the equivalence operation we need to employ the == operator
// for each field of the struct which is of a pre defined type
// this routine would have been much more complicated if each field had
// been itself a user defined type
// In this case, I have defined equivalent to mean matching on every field
int time_eq(struct time a, struct time b)
{
	if ( (a.h == b.h)  && (a.m == b.m)  && (a.s == b.s) )
		return(TRUE);
	else
		return(FALSE);
}

// I have defined not equals to be true if any field doesn't match
int time_ne(struct time a, struct time b)
{
	if ( a.h != b.h )
		return(TRUE);
	if ( a.m != b.m )
		return(TRUE);
	if ( a.s != b.s )
		return(TRUE);
	return(FALSE);
}
```

```
// I have defined gt to mean if the clock time of a is later than the clock
// time of b as measured first on hours, then minutes, then seconds
int time_gt(struct time a, struct time b)
{
	if ( a.h > b.h )
		return(TRUE);
	if ( a.h < b.h )
		return(FALSE);

	// can only get here if a.h == b.h
	if ( a.m > b.m )
		return(TRUE);
	if ( a.m < b.m )
		return(FALSE);

	// can only get here if a.m == b.m
	if ( a.s > b.s )
		return(TRUE);

	return(FALSE);
}

// return true if time a is later than or equal to time b
int time_ge(struct time a, struct time b)
{
	if ( a.h > b.h )
		return(TRUE);
	if ( a.h < b.h )
		return(FALSE);

	// can only get here if a.h == b.h
	if ( a.m > b.m )
		return(TRUE);
	if ( a.m < b.m )
		return(FALSE);

	// can only get her if a.m == b.m
	if ( a.s > b.s )
		return(TRUE);
	if ( a.s < b.s )
		return(FALSE);

	return(TRUE);

}
```

```
// return TRUE if time a is earlier than time b
int time_lt(struct time a, struct time b)
{
	if ( a.h < b.h )
		return(TRUE);
	if ( a.h > b.h )
		return(FALSE);

	// can only get here if a.h == b.h
	if ( a.m < b.m )
		return(TRUE);
	if ( a.m > b.m )
		return(FALSE);

	// can only get here if a.m == b.m
	if ( a.s < b.s )
		return(TRUE);

	return(FALSE);
}

// return TRUE if time a is earlier than or equivalent to time b
int time_le(struct time a, struct time b)
{
	if ( a.h < b.h )
		return(TRUE);
	if ( a.h > b.h )
		return(FALSE);

	// can only get here if a.h == b.h
	if ( a.m < b.m )
		return(TRUE);
	if ( a.m > b.m )
		return(FALSE);

	// can only get her if a.m == b.m
	if ( a.s < b.s )
		return(TRUE);
	if ( a.s > b.s )
		return(FALSE);

	return(TRUE);
}
```

```
int main()
{
	struct time i = { 1,2,3 };
	struct time j = { 1,2,4 };

	// instead of being able to say i == j
	// I have to use the routine time_eq
	// if ( i == j )
	if ( time_eq(i,j) )
		cout << "i is equivalent to j\n";

	// if ( i != j )
	if ( time_ne(i,j) )
		cout << "i is not equivalent to j\n";

	// if ( i < j )
	if ( time_lt(i,j) )
		cout << "i is less than j\n";

	// if ( i <= j )
	if ( time_le(i,j) )
		cout << "i is less than or equal to j\n";

	// if ( i > j )
	if ( time_gt(i,j) )
		cout << "i is greater than j\n";

	// if ( i >= j )
	if ( time_ge(i,j) )
		cout << "i is greater than or equal to j\n";
}

Output From Running Program
i is not equivalent to j
i is less than j
i is less than or equal to j
```

Functions To Implement Relational Operations For User Defined Data Type in C

data type	function	data type	relational operation implemented
struct time	time_eq	struct time	(==)
struct time	time_ne	struct time	(!=)
struct time	time_gt	struct time	(>)
struct time	time_ge	struct time	(>=)
struct time	time_lt	struct time	(<)
struct time	time_le	struct time	(<=)

Wouldn't you rather use the relational operator than have to learn six new function names to compare a struct time to a struct time? In C++ you can.

```
// relops2.cpp
// this program redeclares a simple time struct with three fields
// AS A CLASS WHERE THE RELATIONAL OPERATORS HAVE BEEN TAKEN OVER
// by taking over the relational operators for our defined class
// we eliminate the six new names for the relational subroutines that
// the user would have to learn
// the same amount of work is still done by the executable, meaning that the
// same six subroutines are called, the point though is the intuitiveness of
// the interface to the defined class

#define		TRUE		1
#define		FALSE		0

#include <iostream.h>

// ANY operator functions that we declare, will by default
// have as the BOUND object, an instance of a class time
// the BOUND object will always appear to the immediate left of the operator
class time
{
	private:
		int h,m,s;			// hours, minutes, seconds

	public:
		time(int,int,int);		// three argument constructor

	// int		is the return type of the operation
	// operator	is a keyword that identifies which operator we are taking over
	// ==		is the operator we are taking over
	// (time)	is the argument that will be on the immediate right
	//		of the operator we are taking over if it is a binary operator
	//		by definition a time must be on the immediate left of the operator for the class
	//		for which the operator is being taken over
	int operator == (time);
	int operator != (time);
	int operator >  (time);
	int operator >= (time);
	int operator <  (time);
	int operator <= (time);
};

// when we make a statement like
// if ( i == j )
// the subroutine operator == will be invoked
// i, being on the left of the operator, will be BOUND to the call and will
// be accessible through the "this" pointer
// j, being on the right of the operator, will be PASSED as an argument to
// the subroutine call
```

```
// each of these routines returns TRUE if the condition is met
// each of these routines returns FALSE if the condition is not met

// return type                 int
// function name               operator ==
// class ownership             time
// argument list               object of type time
int time::operator == (time a)
{
        cout << "The == operator was invoked \n";   // teaching print statement to show subr was called
        if ( (this->h == a.h)  && (this->m == a.m)  && (this->s == a.s) )
                return(TRUE);
        else
                return(FALSE);
}

// I have defined != to be true if any field doesn't match
int time::operator !=(time b)
{
        cout << "The != operator was invoked \n";
        if ( this->h != b.h )
                return(TRUE);
        if ( this->m != b.m )
                return(TRUE);
        if ( this->s != b.s )
                return(TRUE);
        return(FALSE);
}

// I have defined > to mean if the clock time of a is later than the clock
// time of b as measured first on hours, then minutes, then seconds
int time::operator > (time b)
{
        cout << "The > operator was invoked \n";
        if ( this->h > b.h )
                return(TRUE);
        if ( this->h < b.h )
                return(FALSE);

        // can only get here if a.h == b.h
        if ( this->m > b.m )
                return(TRUE);
        if ( this->m < b.m )
                return(FALSE);

        // can only get here if a.m == b.m
        if ( this->s > b.s )
                return(TRUE);

        return(FALSE);
}
```

```
// return true if time a is later than or equal to time b
int time::operator >= (time b)
{
	cout << "The >= operator was invoked \n";
	if ( this->h > b.h )
		return(TRUE);
	if ( this->h < b.h )
		return(FALSE);

	// can only get here if a.h == b.h
	if ( this->m > b.m )
		return(TRUE);
	if ( this->m < b.m )
		return(FALSE);

	// can only get her if a.m == b.m
	if ( this->s > b.s )
		return(TRUE);
	if ( this->s < b.s )
		return(FALSE);

	// only get here if a.h == b.h, a.m == b.m and a.s == b.s
	return(TRUE);
}

// return TRUE if time a is earlier than time b
int time::operator < (time b)
{
	cout << "The < operator was invoked \n";
	if ( this->h < b.h )
		return(TRUE);
	if ( this->h > b.h )
		return(FALSE);

	// can only get here if a.h == b.h
	if ( this->m < b.m )
		return(TRUE);
	if ( this->m > b.m )
		return(FALSE);

	// can only get here if a.m == b.m
	if ( this->s < b.s )
		return(TRUE);

	return(FALSE);
}
```

```
// return TRUE if time a is earlier than or equivalent to time b
int time::operator <= (time b)
{
        cout << "The <= operator was called \n";
        if ( this->h < b.h )
                return(TRUE);
        if ( this->h > b.h )
                return(FALSE);

        // can only get here if a.h == b.h
        if ( this->m < b.m )
                return(TRUE);
        if ( this->m > b.m )
                return(FALSE);

        // can only get her if a.m == b.m
        if ( this->s < b.s )
                return(TRUE);
        if ( this->s > b.s )
                return(FALSE);

        return(TRUE);
}

// three argument constructor
time::time(int a, int b, int c) : h(a), m(b), s(c)
{
}
```

```
int main()
{
	time i(1,2,3);
	time j(1,2,4);

	// if ( time_eq(i,j) )
	// now I can use the operator == that the programmer already knows
	// I am not polluting their name space with a function name like time_eq
	if ( i == j )
		cout << "i is equivalent to j\n";

	// if ( time_ne(i,j) )
	if ( i != j )
		cout << "i is not equivalent to j\n";

	// if ( time_lt(i,j) )
	if ( i < j )
		cout << "i is less than j\n";

	// if ( time_le(i,j) )
	if ( i <= j )
		cout << "i is less than or equal to j\n";

	// if ( time_gt(i,j) )
	if ( i > j )
		cout << "i is greater than j\n";

	// if ( time_ge(i,j) )
	if ( i >= j )
		cout << "i is greater than or equal to j\n";
}
```

Output From Running Program

```
The == operator was invoked
The != operator was invoked
i is not equivalent to j

the < operator was invoked
i is less than j

The <= operator was invoked
i is less than or equal to j

The > operator was invoked
The >= operator was invoked
```

Overloading C Relational Operators for User Defined Types in C++

data type	function	data type	relational operation implemented
class time	==	class time	(==)
class time	!=	class time	(!=)
class time	>	class time	(>)
class time	>=	class time	(>=)
class time	<	class time	(<)
class time	<=	class time	(<=)

It is important to note that a subroutine is still being called. But, instead of having to use a subroutine name to call it, we can use the six relational operators that we already know.

```cpp
// mathfunc.cpp
// This program adds additional math oriented features to the time class from the relops
// programs.  This program uses three functions to implement the new math
// functionality

// As a general rule, when doing math on 2 instances of the same class, an instance of that class
// should be yielded as the result, and neither of the 2 instances involved in the operation should be
// affected

// For example,     when you type     x = i + j              you expect x to be of the same type as i and j
                                                             // and you expect i and j to be unaffected

// I want to define what it means to add one time to another
// To do this for my user defined types I have to write subroutines
// In C, I have to call the subroutines by name
// In C++ , in mathops.cpp we will see, that we can overload the math operators to call our subroutines

// For the time class I have now added a day field that will be set to:
// 1 if adding two times together would roll into the next day
// 0 if adding two times together would not roll into the next day
// -1 if subtracting two times would yield a time that is in yesterday

// For the + operation we add a time to a time and return a time
// For the ++ operation, we add one to the hour field and return a time
// For the += operation, we add a time to a time and return a time

#define         TRUE            1
#define         FALSE           0

#include <iostream.h>

struct time
{
        int day;                // 1, 0 or -1
        int h,m,s;              // hours, minutes, seconds
};


// return type                  none
// function name                inc_time
// argument list                struct time called by reference
// this is implementing the x++ syntax, we expect x to change
void inc_time( struct time * tptr )
{
        tptr->h++;
        if ( tptr->h >= 24 )
        {
                tptr->h -= 24;          // roll the hours back by twenty four
                tptr->day = 1;          // indicate that a day rolled over
        }
}
```

```
// return type                  none
// function name                add_time_to_time
// argument list                two time structs passed by reference
// this is implementing the x += j syntax
// we expect x to change, we do not expect j to change
void add_time_to_time( struct time * tptra, struct time * tptrb)
{
        tptra->s += tptrb->s;
        if ( tptra->s > 60 )
        {
                tptra->s -= 60;     // move the seconds field back by 60
                tptra->m++:         // move the minutes field ahead by one
        }

        tptra->m += tptrb->m;
        if ( tptra->m > 60 )
        {
                tptra->m -= 60;     // move the minutes field back by 60
                tptra->h++;         // move the hours   field ahead by one
        }

        tptra->h += tptrb->h;
        if ( tptra->h >= 24 )
        {
                tptra->h -= 24;     // roll the hours back by twenty four
                tptra->day = 1;     // indicate that a day rolled over
        }
}
```

```
// return type              struct time
// function name            add_two_times
// argument list            two struct times passed by reference
// this is implementing the x = i + j functionality
// we expect x to change, we don't expect i or j to change
struct time add_two_times(struct time * tptra, struct time * tptrb )
{
        struct time temp;
        int minute_carry = 0;
        int hour_carry = 0;

        temp.s = tptra->s + tptrb->s;
        if ( temp.s > 60 )
        {
                temp.s -= 60;           // move the seconds field back by 60
                minute_carry = 1;// move the minutes field ahead by one
        }

        temp.m = tptra->m + tptrb->m + minute_carry;
        if ( temp.m > 60 )
        {
                temp.m -= 60;   // move the minutes field back by 60
                hour_carry = 1; // move the hours   field ahead by one
        }

        temp.h = tptra->h + tptrb->h + hour_carry;
        if ( temp.h >= 24 )
        {
                temp.h -= 24;   // roll the hours back by twenty four
                temp.day = 1;   // indicate that a day rolled over
        }
        else
        {
                temp.day = 0;
        }
        return (temp):          // return the bound object
}

void print_time(struct time a)
{
        cout    << "The four fields are "
                << a.day
                << " "
                << a.h
                << " "
                << a.m
                << " "
                << a.s
                << endl;
        return;
}
```

```
int main()
{
	struct time i = {0,1,2,3};
	struct time j = {0,1,2,4};
	struct time k;

	cout << "i is ";
	print_time(i);		// print_time is going to do a field by field output
				// Question?  Could we overload <<stay tuned!

	cout << "j is ";
	print_time(j);
	cout << endl << endl;

	// To "add one" to a time, we have to use a subroutine like inc_time(), even though
	// we have the math operator ++ that we are used to using.  The problem is that ++ is only
	// defined for pre-defined types.  Not for user defined types.  That's why in this program we
	// need the subroutine inc_time.  In the next program, mathops.cpp, we will NOT use inc_time
	// We will overload ++ and use it instead.  The same subroutine will be called to do the exact
	// same work, however, the usage of the ++ operator will be more rememberable than the
	// subroutine inc_time
	inc_time(&i);
	cout << "i after inc_time(i) is ";
	print_time(i);
	cout << endl << endl;

	add_time_to_time(&i,&j);
	cout << "i after add_time_to_time(&i,&j) is ";
	print_time(i);
	cout << "j after add_time_to_time(&i,&j) is ";
	print_time(j);
	cout << endl << endl;

	k = add_two_times(&i,&j);
	cout << "k after k = add_two_times(&i,&j) is ";
	print_time(k);
	cout << "i after k = add_two_times(&i,&j) is ";
	print_time(i);
	cout << "j after k = add_two_times(&i,&j) is ";
	print_time(j);
	cout << endl << endl;
}
```

Output From Running Program

```
i is					The four fields are 0 1 2 3
j is					The four fields are 0 1 2 4

i after inc_time(i) is			The four fields are 0 2 2 3

i after add_time_to_time(&i,&j) is	The four fields are 0 3 4 7
j after add_time_to_time(&i,&j) is	The four fields are 0 1 2 4

k after k = add_two_times(&i,&j) is	The four fields are 0 4 6 11
i after k = add_two_times(&i,&j) is	The four fields are 0 3 4 7
j after k = add_two_times(&i,&j) is	The four fields are 0 1 2 4
```

C Math Operators For Pre-defined Types

integer	+	
	+=	
	++	
	--	
	-	
	-=	
	/	
	/=	
	*	
	*=	
	<<	
	>>	
	%	
char	+	
	+=	
	++	
	--	
	-	
	-=	
	/	
	/=	
	*	
	*=	
	<<	
	>>	
	%	
float	+	
	+=	
	-	
	-=	
	/	
	/=	
	*	
	*=	
struct	=	= is the only arithmetic operator defined for user defined types

```
// mathops.cpp
// this program adds additional features to the time class from the relops
// programs.  In this program we take over the +  +=  ++   operators
// FOR THE TIME CLASS ONLY
// taking over the + or any other operator FOR A CLASS
// in no way influences what + means for any of the pre defined types
// or for any other class

// What we are doing is defining what it means to add one time to another
// Each class designer can define what it means to perform math between
// instances of the class of which they are are creating
// For some classes, it doesn't make sense to be able to add two instances
// of the class together
// For other classes it does
// For this example, focus on the mechanism of what I'm doing, not whether
// my definition of adding two times together makes sense or not

// For the time class I have now added a day field that will be set to:
// 1 if adding two times together would roll into the next day
// 0 if adding two times together would not roll into the next day
// -1 if subtracting two times would yield a time that is in yesterday

// For the + operation we add a time to a time and return a time
// For the ++ operation, we add one to the hour field and return a time
// For the += operation, we add a time to a time and return a time

// Note that the ++ operator is unary, there will be a BOUND object
// but no argument object
// the += operator is binary, there will be a BOUND object and an argument
// the BOUND object will be on the left of the operator
// the argument will be on the right of the operator
// the +  operator is binary, there will be a BOUND object and an argument
// the BOUND object will be on the left of the operator
// the argument will be on the right of the operator

// Note the difference in what is returned for the + ++ and += operations
// The ++ operation affects the BOUND object
// The += operation affects the BOUND object but not the argument object
// The +  operation affects neither the BOUND object nor the argument object
```

```
#define     TRUE     1
#define     FALSE    0

#include <iostream.h>

class time
{
    private:
        int day;          // 1, 0 or -1
        int h,m,s;        // hours, minutes, seconds

    public:
        time(int,int,int);        // three argument constructor
        time();                   // no argument constructor
        time operator ++ ();      // overload the mathematics operators
        time operator += (time);
        time operator +  (time);
        void print_time();
};


// return type          time
// function name        operator ++
// class ownershiptime
// argument list        none
time time::operator ++ ()
{
    cout << "The ++ operator was invoked \n";
    this->h++;
    if ( this->h >= 24 )
    {
        this->h -= 24;            // roll the hours back by twenty four
        this->day = 1;            // indicate that a day rolled over
    }
    return(*this);                // return the bound object
}
```

```
// return type                  time
// function name                operator +=
// class ownership              time
// argument list                an instance of a time class object
time time::operator += (time b)
{
        cout << "The += operator was invoked \n";

        this->s += b.s;
        if ( this->s > 60 )
        {
                this->s -= 60;      // move the seconds field back by 60
                this->m++;          // move the minutes field ahead by one
        }

        this->m += b.m;
        if ( this->m > 60 )
        {
                this->m -= 60;      // move the minutes field back by 60
                this->h++;          // move the hours   field ahead by one
        }

        this->h += b.h;
        if ( this->h >= 24 )
        {
                h -= 24;            // roll the hours back by twenty four
                day = 1;            // indicate that a day rolled over
        }
        return (*this);             // return the bound object
}
```

```
// return type             time
// function name           operator +
// class ownership         time
// argument list           an instance of a time class object
time time::operator + (time b)
{
        cout << "The + operator was invoked \n";

        int hour_carry = 0;
        int minute_carry = 0;

        time temp;      // can't destroy the BOUND object or the invoking object
                        // When you say i = j + k  you expect i to change
                        // you don't expect j or k to change

        temp.s = this->s + b.s;
        if ( temp.s > 60 )
        {
                temp.s -= 60;           // move the seconds field back by 60
                minute_carry = 1;// move the minutes field ahead by one
        }

        temp.m = this->m + b.m + minute_carry;
        if ( temp.m > 60 )
        {
                temp.m -= 60;           // move the minutes field back by 60
                hour_carry = 1;         // move the hours   field ahead by one
        }

        emp.h = this->h + b.h + hour_carry;
        if ( temp.h >= 24 )

        temp.h -= 24;                   // roll the hours back by twenty four
        temp.day = 1;                   // indicate that a day rolled over
        }
        return (temp);                  // return the newly created object
}

void time::print_time() {
        cout    << "The four fields are "
                << day
                << " "
                << h
                << " "
                << m
                << " "
                << s
                << endl;
                return;
}
// three argument constructor
time::time(int a, int b, int c) : h(a), m(b), s(c), day(0)
{
}
```

```
time::time() : h(0), m(0), s(0), day(0)
{
}

int main(){
        time i(1,2,3);      time j(1,2,4);      time k;

        cout << "i is ";    i.print_time();

        cout << "j is ";    j.print_time();
        cout << endl << endl;

        // increment i using the mathematics operator
        // You don't have to call some function like increment_time(i)
        i++;
        cout << "i after i++ is ";
        i.print_time();
        cout << endl << endl;

        // don't have to call some function like add_time_to_time(&i,&j);
        i += j;
        cout << "i after i += j is ";
        i.print_time();
        cout << "j after i += j is ";
        j.print_time();
        cout << endl << endl;

        // don't have to call some function like k = add_two_times(&i,&j)
        k = i + j;
        cout << "k after k = i + j is ";
        k.print_time();
        cout << "i after k = i + j is ";
        i.print_time();
        cout << "j after k = i + j is ";
        j.print_time();
        cout << endl << endl;
}
Output From Running Program

i is The four fields are 0 1 2 3
j is The four fields are 0 1 2 4

The ++ operator was invoked
i after i++ is The four fields are 0 2 2 3

The += operator was invoked
i after i += j is The four fields are 0 3 4 7
j after i += j is The four fields are 0 1 2 4

The + operator was invoked
k after k = i + j is The four fields are 0 4 6 11
i after k = i + j is The four fields are 0 3 4 7
j after k = i + j is The four fields are 0 1 2 4
```

C++ Taking Over Mathematics Operators

Take over	++	increment this object by some known amount defined to be "one"
		this object expected to change
	x++	x changes
		x stays the same type
Take over	+=	update this object by "adding" another instance of the same type
		expect this object to change
	x += y	x changes, y doesn't
		y of same type as x
Take over	+	create a new object by "adding" two objects of the same type as the desired yielded object
		expect new object to change, don't expect existing instances to change
	x = y + z	x changes, y doesn't, z doesn't
		y and z of same type as x

BUT: What If the thing(s) being added to your class aren't of the same type??
Could we do

Take over	+=	update this object by "adding" some other type instance to it?
		expect this object to change
	x += ABC	x of type xyz
		ABC of type ABC
		x changed but remains of type xyz

CAN WE DO THIS?
Yes!! See mathops1.cpp

```
// mathops1.cpp

// In this program we polymorphise the + and += operators
// Previously we had defined what it meant to add a time to a time
// Now, additionally,  we are going to define what it means to add an int to a time
// We are going to have a destructive add +=     time += int
// We are going to have a non_destructive add +  time  = time + int

#define         TRUE        1
#define         FALSE       0

#include <iostream.h>

class time
{
        private:
                int day;            // 1, 0 or -1
                int h,m,s;          // hours, minutes, seconds

        public:
                time(int,int,int);          // three argument constructor
                time();                     // no argument constructor

                // Math operators for times and times
                time operator ++ ();
                time operator += (time);  // add all the fields to each other
                time operator +  (time);

                // Math operators for times and ints
                time operator += (int);             // add to the hours field
                time operator +  (int);             // add to the hours field
                void print_time();
};


// return type              time
// function name            operator ++
// class ownership          time
// argument list            none
time time::operator ++ ()
{
        cout << "The time++ operator was invoked \n";
        this->h++;
        if ( this->h >= 24 )
        {
                this->h -= 24;                      // roll the hours back by twenty four
                this->day = 1;                      // indicate that a day rolled over
        }
        return(*this);      // return the bound object
}
```

```
// return type          time
// function name        operator +=
// class ownership      time
// argument list        an instance of a time class object
time time::operator += (time b)
{
        cout << "The time += time operator was invoked \n";

        this->s += b.s;
        if ( this->s > 60 )
        {
                this->s -= 60;                  // move the seconds field back by 60
                this->m++;                      // move the minutes field ahead by one
        }

        this->m += b.m;
        if ( this->m > 60 )
        {
                this->m -= 60;                  // move the minutes field back by 60
                this->h++;                      // move the hours   field ahead by one
        }

        this->h += b.h;
        if ( this->h >= 24 )
        {
                h -= 24;                        // roll the hours back by twenty four
                day = 1;                        // indicate that a day rolled over
        }
        return (*this);                         // return the bound object
}
```

```
// return type          time
// function name        operator +
// class ownership      time
// argument list        an instance of a time class object
time time::operator + (time b)
{
        cout << "The time + time operator was invoked \n";
        int     minute_carry = 0;
        int     hour_carry = 0;
        time temp;
        temp.s = this->s + b.s;

        if ( temp.s > 60 )
        {
                temp.s -= 60;           // move the seconds field back by 60
                minute_carry = 1;// move the minutes field ahead by one
        }

        temp.m = this->m + b.m + minute_carry;
        if ( temp.m > 60 )
        {
                temp.m -= 60;           // move the minutes field back by 60
                hour_carry = 1;         // move the hours   field ahead by one
        }

        temp.h = this->h + b.h;
        if ( temp.h >= 24 )
        {
                temp.h -= 24;           // roll the hours back by twenty four
                temp.day = 1;           // indicate that a day rolled over
        }
        return (temp);                  // return the newly created object
}
```

```
// THIS IS THE START OF THE POLYMORPHISED FUNCTIONS
// WE ARE NOW TAKING OVER +=  and + for a time on the left and an int on the right
// These are only taken over for this class
// += and + are not affected for any other class
// By convention, we expect an object back of the type that is on the left of the operator
// return type          time
// function name        operator +=
// class ownership      time
// argument list        one integer
time time::operator += (int b)
{
        cout << "The time += int operator was invoked \n";

        this->h += b;
        if ( this->h >= 24 )
        {
                h -= 24;            // roll the hours back by twenty four
                day = 1;            // indicate that a day rolled over
        }
        return (*this);             // return the bound object
}

// return type          time
// function name        operator +
// class ownership      time
// argument list        an int
time time::operator + (int b)
{
        cout << "The time = time + int operator was invoked \n";

        time temp;

        temp.h = this->h + b;
        if ( temp.h >= 24 )
        {
        temp.h -= 24;               // roll the hours back by twenty four
        temp.day = 1;               // indicate that a day rolled over
        }
        return (temp);              // return the newly created object
}

void time::print_time()     {
        cout    << "The four fields are "
                << day
                << " "
                << h
                << " "
                << m
                << " "
                << s
                << endl;
        return;
}
```

```
// three argument constructor
time::time(int a, int b, int c) : h(a), m(b), s(c), day(0) { }
time::time() : h(0), m(0), s(0), day(0)  { }

int main()
{
	time i(1,2,3);  time j(1,2,4);  time k;

	cout << "i is ";
	i.print_time();

	cout << "j is ";
	j.print_time();
	cout << endl << endl;

	// PROVE THAT THE OLD WAY STILL WORKS
	// increment i using the mathematics operator
	// You don't have to call some function like increment_time(i)
	i++;
	cout << "i after i++ is ";
	i.print_time();
	cout << endl << endl;

	// don't have to call some function like add_time_to_time(&i,&j);
	i += j;
	cout << "i after i += j is ";
	i.print_time();
	cout << "j after i += j is ";
	j.print_time();
	cout << endl << endl;

	// don't have to call some function like k = add_two_times(&i,&j)
	k = i + j;
	cout << "k after k = i + j is ";
	k.print_time();
	cout << "i after k = i + j is ";
	i.print_time();
	cout << "j after k = i + j is ";
	j.print_time();
	cout << endl << endl;

	// SHOW WHAT THE NEW WAY DOES
	// Now I am going to exercise the new polymorphic features
	i += 7;
	cout << "i after i += 7 is ";
	i.print_time();

	k = i + 4;
	cout << "k after k = i + 4 is ";
	k.print_time();
	cout << "i after k = i + 4 is ";
	i.print_time();
}
```

Output From Running The Program

i is The four fields are 0 1 2 3
j is The four fields are 0 1 2 4

The time++ operator was invoked
i after i++ is The four fields are 0 2 2 3

The **time += time** operator was invoked
i after i += j is The four fields are 0 3 4 7
j after i += j is The four fields are 0 1 2 4

The **time = time + time** operator was invoked
k after k = i + j is The four fields are 0 4 6 11
i after k = i + j is The four fields are 0 3 4 7
j after k = i + j is The four fields are 0 1 2 4

The **time += int** operator was invoked
i after i += 7 is The four fields are 0 10 4 7

The **time = time + int** operator was invoked
k after k = i + 4 is The four fields are 0 14 0 0
i after k = i + 4 is The four fields are 0 10 4 7

```
// ioops.cpp
// this program shows how the function print_time can be removed
// Just like we can take over the relational operators and arithmetic operators
// We can take over the << operator for any class that we define
// Again, the same amount of work gets done as when we had to call print_time
// But the interface is consistent with what the user of the class already
// applies to pre defined variables and other classes that use this mechanism

// taking over the << operator for this class does not affect the << operator for any other class
// or any pre-defined or user defined class

// this program introduces two new concepts
// FRIEND functions
// REFERENCES
// If you are unfamiliar with FRIEND functions or REFERENCES
// There are other examples programs in this book which illustrate how these concepts work

// For the evolutionary flow of taking over relational operators,
// then mathematics operators then io operators, I have chosen to cover
// friend fucntions and references next.  It was a real chicken and egg
// problem.  At least showing them in use provides justification for learning
// them!

#include <iostream.h>

class time
{
        private:
                int day;            // 1, 0 or -1
                int h,m,s;          // hours, minutes, seconds

        public:
                time(int,int,int);  // three argument constructor
                time();             // no argument constructor

                // return a reference to an output stream object
                // take over the << operator
                // An instance of a time class object must be immediately
                // to the right of the << operator
                // An instance of an ostream object must be immediately to the left of the << operator
                // Called via a call like     cout << t1;
                // friend says that the function is not a member of the class but may still access the
        private
                // data of the instances of the class
                friend ostream& operator << (ostream&,time);
};

```

```
// return type          ostream&
// function name        operator <<
// class ownershipnone, it is a friend function of two classes
// argument list        an instance of a time class object
ostream & operator << (ostream & os, time a)
{
        // no variable  this    is available in a friend function
        // all addresses need to be passed as arguments

        os << "The ostream << time operator was invoked \n";

        os      << "The four fields are "
        << a.day        // this << is between an ostream on the left and an int on the right
        << " "
        << a.h          // this << is between an ostream on the left and an int on the right
        << " "
        << a.m          // this << is between an ostream on the left and an int on the right
        << " "
        << a.s
        << endl;

        return(os);
}

// constructors
time::time(int a, int b, int c) : h(a), m(b), s(c), day(0)  {}
time::time() : h(0), m(0), s(0), day(0)  {}

int main()
{
        time i(1,2,3);      time j(1,2,4);      time k;

        cout << "i is \n";
        cout << i;          // don't call print_time,  use the << operator instead
                            // the same amount of, and in fact the EXACT same work gets done
                            // the difference is that the programmers already know <<
                            // the programmers don't already know print_time
                            // going with the overloaded operator, they don't need to learn any new
                            // function calls to use your class!!!
        cout << endl << endl;

        cout << "j is \n";
        cout << j;                  // taking over the << for an ostream on the left and a time on the right
        cout << endl << endl;       // does not affect any other calls to <<

        cout << "k is \n";          // this is a call to << with an ostream on the left
                                            //and a string on the right
        cout << k;                  // it is unaffected by our taking over for an ostream on the left and
        cout << endl << endl;       // a time on the right
}
```

```
Output From Running Program
i is
The ostream << time operator was invoked
The four fields are 0 1 2 3

j is
The ostream << time operator was invoked
The four fields are 0 1 2 4

k is
The ostream << time operator was invoked
The four fields are 0 0 0 0
```

Section 5

This section is about casting.
If you are any kind of C programmer you know how casting works.
You are able to cast from ints to floats and floats to doubles and so on.
But, in C, if you have a user defined data structure and you want to convert it into some other type of user defined data structure or even into one of the pre-defined data types, guess what? You have to write a subroutine to do the conversion and then use that subroutine by name. In C++ you will still have to write the subroutine to do the conversion, however, you won't have to use the subroutine by name, you will be able to invoke it using the same casting syntax that you are already familiar with. This has great value to the programmer because we are not polluting their mind with the names of subroutines. We are making use of the casting syntax they already know and love (hate?)

The point of creating and overloading casting operators is this:

- You are able to use your classes using the syntax you already know.
- You don't have to learn a gazillion (gumpism) new names to use a class.
- The amount of work that is done by the computer at run time is the same,
  - a subroutine is still called and run to do the work,
  - However, the programmer's job was made easier.

Any time that you can make the progammer's job easier, you are doing something good.
We want the programmers to focus on the problem to be solved and not on the syntax required to do that.

This section will show you how to provide casting operations between:

a user defined class and a pre-defined type	using a traditional function
a user defined class and a pre defined type	using casting operator
a pre defined type and a class	using a constructor

- a user defined class and the same user defined class
- a user defined class and a regular C structure
- a regular C structure and a user defined class

- a user defined class and a different user defined class where no inheritance relationship exists
- a user defined class and a different user defined class where an inheritance relationship does exist

castfunc.cpp	casting a class into a pre-defined type using functions
castops.cpp	casting a class into a pre-defined type using casting syntax
castcons.cpp	casting pre-defined types to classes using constructors
castcopy.cpp	building one instance of a class using another instance of the same class
cls2stct.cpp	casting a class instance into a struct instance
stct2cls.cpp	casting a struct instance into a class instance
cls2cls1.cpp	casting from one class to another class where no inheritance exists
cls2cls2.cpp	casting from one class to another where inheritance exists

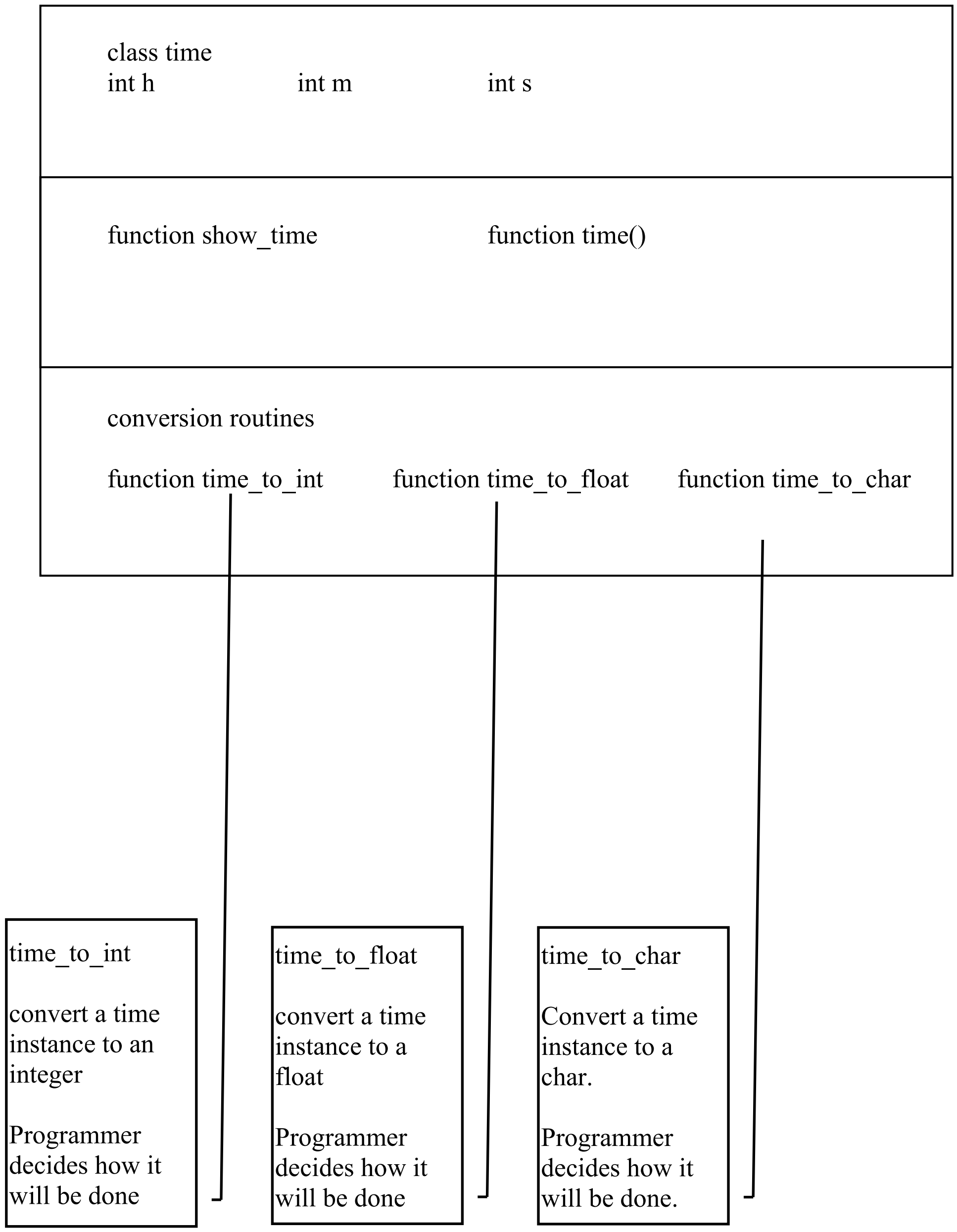
```
// castfunc.cpp

#include <iostream.h>

// This program shows how a C programmer would write subroutines
// to change an instance of a class time into the pre defined data
// types.  These conversions, called casting, may or may not make sense
// depending on your point of view.  What is important, is that the mechanism
// be explained so that if you wanted to do a conversion like this, you could define what it meant
class time
{
	private:
		int h,m,s;			// hours, minutes, seconds

	public:
		time(int,int,int);		// three argument constructor
		void show_time();		// show what time it is

	// int is the return type of the operation
	// operator is a keyword that identifies which operator we are taking over
	// int is the operator we are taking over
	// (time) is the argument that will be on the immediate right
	// of the operator we are taking over if it is a binary operator
	int   time_to_int   ();		// how to turn a time into an int

	// these are inconvenient to use.  If we are going to have conversion routines, we may as
	// well make them convenient to use.  We already have a casting syntax that we are familiar with
	// i.e. to turn a float into an int we would use		i  = (int) f;
	// so why don't we have casting routines for user-defined types?
	// the simple answer is that the compiler cannot anticipate what your user defined type will
	// have as fields and how you want to turn those fields into an int or float or other type
	// However, the compiler can anticipate our desire to do conversions, therefore we are provided
	// with a casting syntax  like   i = (int) f
	// and the (int)  will be an operator.  In C++ we can overload this operator just like we have
	// overloaded the relational and mathematical operators.  In the next program, castops.cpp
	// we do this and therefore don't need routines like time_to_float and time_to_char
	float time_to_float ();		// how to turn a time into a float
	char  time_to_char  ();		// how to turn a time into a char
};

void time::show_time()
{
	cout << "h is " << h << endl;
	cout << "m is " << m << endl;
	cout << "s is " << s << endl << endl;
}
```

return type | class membership | function name | argument list

```
int time::time_to_int( )
{
	cout << "time_to_int was called \n";
	return ( this->h ); // return the integer hour
	// you could have done whatever you wanted here
}

float time::time_to_float( )
{
	cout << "The time_to_float routine was called \n";

	// I have created some silly conversion from time to float
	// you could probably come up with one that made sense
	// whoever writes the cast controls what happens
	return ( (float) (this->h + this->m + this->s) );
}

char time::time_to_char ( )
{
	cout << "The time_to_char function was called \n";
	return ( 'a' );		// obviously you would do something more sophisticated
				// than just return the letter a.   The point is, that whatever happens, you are in
				// control of the conversion from class to defined type
}

// three argument constructor
time::time(int a, int b, int c) : h(a), m(b), s(c)
{
}
```

```
int main()
{
	time t(1,2,3);

	int i = 0;
	float f = 0.0;
	char c = 'C';

	cout << "Before casting i is " << i << endl;
	cout << "Before casting to int t is ";
	t.show_time();
	i = t.time_to_int();
	cout << "After casting i is " << i << endl;
	cout << "After casting to int t is ";
	t.show_time();

	cout << "Before casting f is " << f << endl;
	cout << "Before casting to float t is ";
	t.show_time();
	f = t.time_to_float();
	cout << "After casting f is " << f << endl;
	cout << "After casting to float t is ";
	t.show_time();

	cout << "Before casting c is " << c << endl;
	cout << "Before casting to char t is ";
	t.show_time();
	c = t.time_to_char();
	cout << "After casting c is " << c << endl;
	cout << "After casting to char t is ";
	t.show_time();
}
```

```
Output From Running Program

Before casting i is 0
Before casting to int t is h is 1
m is 2
s is 3

time_to_int was called
After casting i is 1
After casting to int t is h is 1
m is 2
s is 3

Before casting f is 0
Before casting to float t is h is 1
m is 2
s is 3

The time_to_float routine was called
After casting f is 6
After casting to float t is h is 1
m is 2
s is 3

Before casting c is C
Before casting to char t is h is 1
m is 2
s is 3

The time_to_char function was called
After casting c is a
After casting to char t is h is 1
m is 2
s is 3
```


class time
int h
int m
int s
function show_time
function time()
conversion routines
function time_to_int
function time_to_float
function time_to_char
time_to_int
convert a time instance to an integer
Programmer decides how it will be done
time_to_float
convert a time instance to a float
Programmer decides how it will be done
time_to_char
Convert a time instance to a char.
Programmer decides how it will be done.

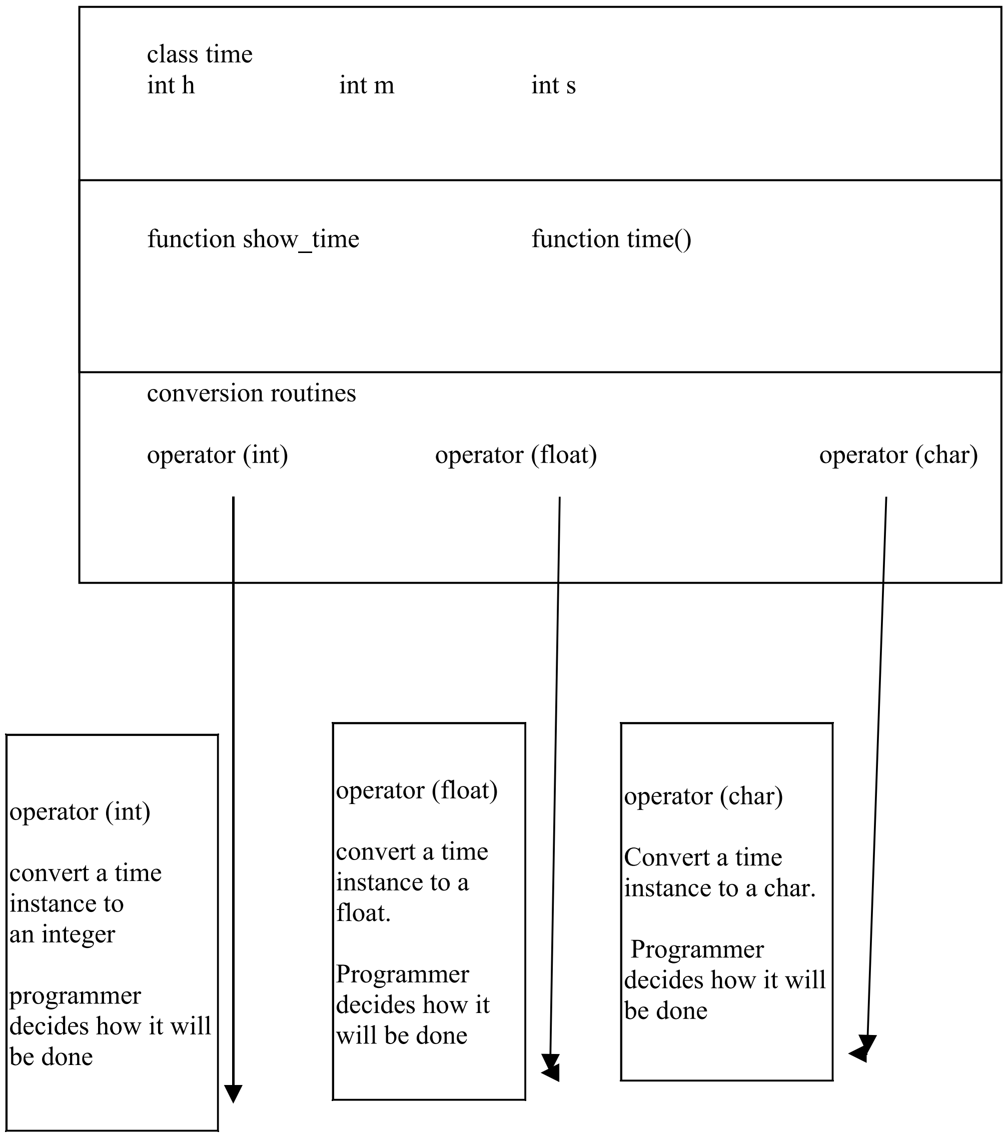
```
// castops.cpp

#include <iostream.h>

// this program improves castfunc.cpp by using the C++ casting syntax
// it removes from the programmer the onerous task of learning the names
// of the conversion routines
// instead it provides the programmer with the standard casting operators
// that they are used to using to cast other things to ints, floats chars

// If we are going to allow conversions from one type to another,
// We may as well make the syntax convenient and straight forward

// C provides the casting syntax   i = (int) f + (int) g;
// C++ extends the casting syntax to user defined types, structs and classes

class time
{
        private:
                int h,m,s;                      // hours, minutes, seconds

        public:
                time(int,int,int);              // three argument constructor
                void show_time();               // show what time it is

        // int is the return type of the operation
        // operator is a keyword that identifies which operator we are taking over
        // int is the operator we are taking over
        // (time) is the argument that will be on the immediate right
        // of the operator we are taking over if it is a binary operator
        operator int   ();                      / how to turn a time into an int
        operator float ();              // how to turn a time into a float
        operator char  ();                      // how to turn a time into a char
};

void time::show_time()
{
        cout << "h is " << h << endl;
        cout << "m is " << m << endl;
        cout << "s is " << s << endl << endl;
}
```

Note that the return type is not specified
If you are going to provide a casting operator for a class to a type
The return type will match the name of the operator
In this case the operator is operator int
Therefore the return type will be int

```
return      class           function    argument
type        membership      name        list




time::operator int ( )
{
        cout << "The (int) operator was invoked \n";
        return ( this->h );          // return the integer hour
        // you could have done whatever you wanted here
}


float time::operator float( )
{
        cout << "The float operator was invoked \n";

        // I have created some silly conversion from time to float
        // you could probably come up with one that made sense
        // whoever writes the cast controls what happens
        return ( (float) (this->h + this->m + this->s) );
}

char time::operator char ( )
{
        cout << "The char operator was invoked \n";
        return ( 'a' );
}

// three argument constructor
time::time(int a, int b, int c) : h(a), m(b), s(c)
{
}
```

```
int main()
{
time t(1,2,3);

	int i = 0;
	float f = 0.0;
	char c = 'C';

	cout << "Before casting i is " << i << endl;
	cout << "Before casting to int t is ";
	t.show_time();
	i = (int) t;		// now we don't call the subroutine like time_to_int, we just cast it
				// exactly the same amount of work will be done, in fact we end up calling
				// a subroutine that is suspiciously similar to time_to_int,
				// the point here isn't the efficiency of the language but the ease of use
				// of the language and the intuitiveness, the ability to use without learning,
				// of features of the language.  Since C programmers already know casting
				// syntax, when using a user-defined class, they should be able to cast.
	cout << "After casting i is " << i << endl;
	cout << "After casting to int t is ";
	t.show_time();

	cout << "Before casting f is " << f << endl;
	cout << "Before casting to float t is ";
	t.show_time();
	f = (float) t;
	cout << "After casting f is " << f << endl;
	cout << "After casting to float t is ";
	t.show_time();

	cout << "Before casting c is " << c << endl;
	cout << "Before casting to char t is ";
	t.show_time();
	c = (char) t;
	cout << "After casting c is " << c << endl;
	cout << "After casting to char t is ";
	t.show_time();
}
```

```
Output From Running Program

Before casting i is 0
Before casting to int t is h is 1
m is 2
s is 3

The (int) operator was invoked
After casting i is 1
After casting to int t is h is 1
m is 2
s is 3

Before casting f is 0
Before casting to float t is h is 1
m is 2
s is 3

The float operator was invoked
After casting f is 6
After casting to float t is h is 1
m is 2
s is 3

Before casting c is C
Before casting to char t is h is 1
m is 2
s is 3

The char operator was invoked
After casting c is a
After casting to char t is h is 1
m is 2
s is 3
```


class time
int h
int m
int s
function show_time
function time()
conversion routines
operator (int)
operator (float)
operator (char)
operator (int)
convert a time instance to an integer
programmer decides how it will be done
operator (float)
convert a time instance to a float.
Programmer decides how it will be done
operator (char)
Convert a time instance to a char.
Programmer decides how it will be done

```
// castcons.cpp

// So you say that's great, we can now turn a user defined type into any pre defined type
// Well then, how do we cast a pre-defined type to a user-defined type?
// Answer, you don't cast, you construct!
// We have looked at constructors before, this program will look at them again, and will show how
// One way that a programmer might hasve decided to build a time from many different combinations
// and variations of pre defined types
#include <iostream.h>

// this program shows how a time could be built from
// one int
// two ints
// three ints
// one float
// two floats
// three floats
// one char
// two chars
// three chars

// these are all constructor functions

class time
{
	private:
		int h,m,s;			// hours, minutes, seconds

	public:
		void show_time();		// show what time it is

		time(int);			// build a time from one int
		time(int,int);			// build a time from two ints
		time(int,int,int);		// build a time from three ints
		time(float);			// build a time from one float
		time(float,float);		// build a time from two floats
		time(float,float,float);	// build a time from three floats
		time(char);			// build a time from one char
		time(char,char);		// build a time from two chars
		time(char,char,char);		// build a time from three chars
};

void time::show_time()
{
	cout << "h is " << h << endl;
	cout << "m is " << m << endl;
	cout << "s is " << s << endl << endl;
}

```

```
// one int argument constructor
time::time(int a) : h(a), m(0), s(0)
{
        cout << "One int arg constructor called \n";
}
// two int argument constructor
time::time(int a, int b) : h(a), m(b), s(0)
{
        cout << "Two int arg constructor called \n";
}
// three int argument constructor
time::time(int a, int b, int c) : h(a), m(b), s(c)
{
        cout << "Three int arg constructor called \n";
}

// one float argument constructor
time::time(float a) : h( (int) a), m(0), s(0)
{
        cout << "One float arg constructor called \n";
}
// two float argument constructor
time::time(float a, float b) : h( (int) a), m( (int) b), s(0)
{
        cout << "Two float arg constructor called \n";
}
// three float argument constructor
time::time(float a, float b, float c) : h( (int) a), m( (int) b), s( (int)c)
{
        cout << "Three float arg constructor called \n";
}

// one char argument constructor
time::time(char a) : h( (int) a), m(0), s(0)
{
        cout << "One Char arg constructor called \n";
}
// two char argument constructor
time::time(char a, char b) : h((int) a), m( (int) b), s(0)
{
        cout << "Two Char arg constructor called \n";
}
// three char argument constructor
time::time(char a, char b, char c) : h((int) a), m((int) b), s((int) c)
{
        cout << "Three Char arg constructor called \n";
}
```

```
int main()
{
	int i = 1;		int j = 2;		int k = 3;
	char a = 'a';		char b = 'b';		char c = 'c';
	float f = 10.1;	float g = 20.2;	float h = 30.3;

	// build examples using all nine types of constructors
	time d1(i);
	cout << "d1 is ";
	d1.show_time();

	time d2(i,j);
	cout << "d2 is ";
	d2.show_time();

	time d3(i,j,k);
	cout << "d3 is ";
	d3.show_time();
	cout << endl;

	time d4(f);
	cout << "d4 is ";
	d4.show_time();

	time d5(f,g);
	cout << "d5 is ";
	d5.show_time();

	time d6(f,g,h);
	cout << "d6 is ";
	d6.show_time();
	cout << endl;

	time d7(a);
	cout << "d7 is ";
	d7.show_time();

	time d8(a,b);
	cout << "d8 is ";
	d8.show_time();

	time d9(a,b,c);
	cout << "d9 is ";
	d9.show_time();
}
```

Output From Running Program

```
One int arg constructor called
d1 is h is 1
m is 0
s is 0

Two int arg constructor called
d2 is h is 1
m is 2
s is 0

Three int arg constructor called
d3 is h is 1
m is 2
s is 3


One float arg constructor called
d4 is h is 10
m is 0
s is 0

Two float arg constructor called
d5 is h is 10
m is 20
s is 0

Three float arg constructor called
d6 is h is 10
m is 20
s is 30


One Char arg constructor called
d7 is h is 97
m is 0
s is 0

Two Char arg constructor called
d8 is h is 97
m is 98
s is 0

Three Char arg constructor called
d9 is h is 97
m is 98
s is 99
```

```
// castcopy.cpp

#include <iostream.h>

// this program shows how a time could be built from another time
// basically what we are doing is constructing one time from another tim
// this is sometimes called the copy constructor
// it is implemented using the operator =

class time
{
	private:
		int h,m,s;			// hours, minutes, seconds

	public:
		void show_time();		// show what time it is

		time();				// no arg time constructor
		time(int,int,int);		// three arg time constructor
		time operator = (time);	// "copy operator",   or one arg where arg is time, constructor
};

void time::show_time()
{
	cout << "h is " << h << endl;
	cout << "m is " << m << endl;
	cout << "s is " << s << endl << endl;
}

// no int argument constructor
time::time()
{
	cout << "No arg constructor called \n";
	h = 12;
	m = 34;
	s = 56;
}

// three int argument constructor
time::time(int a, int b, int c) : h(a), m(b), s(c)
{
	cout << "Three int arg constructor called \n";
}
```

```
This subroutine will be called when someone uses syntax like
time t2 = t1;                    where t1 was already defined somewhere else
                                 t2 will be created using the copy constructor

return     class          function        argument
type       membership     name            list

time time::operator = (time t1)
{
        cout << "The copy operator was invoked\n";
        this->h = t1.h;
        this->m = t1.m;
        this->s = t1.s;
        return *this;
}

int main()
{
        cout << "AAA\n";
        time t1;
        t1.show_time();

        cout << "BBB" << endl;
        time t2(1,2,3);
        t2.show_time();

        cout << "CCC " << endl;
        time t3 = t1;                   // which constructor will be called?
        t3.show_time();

        cout << "DDD " << endl;
        t3 = t2;
        t3.show_time();
}
```

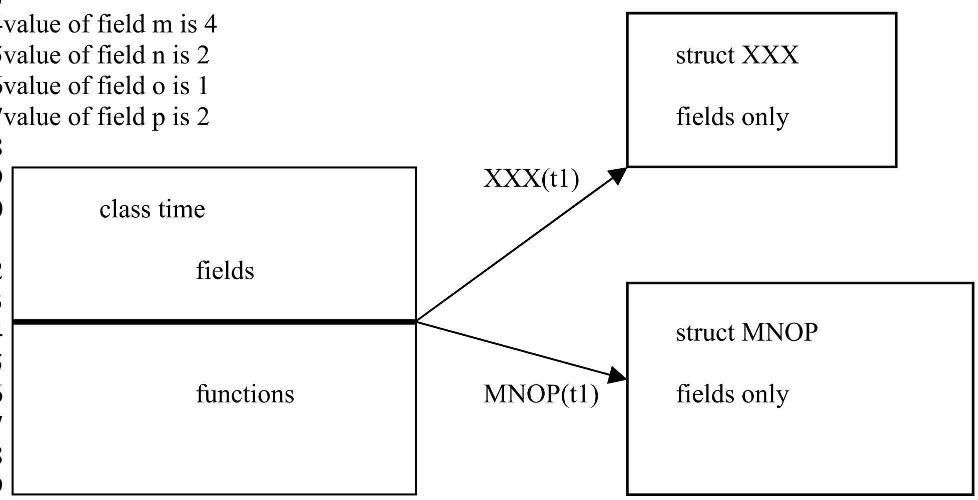
```
Output From Running Program

AAA
No arg constructor called
h is 12
m is 34
s is 56

BBB
Three int arg constructor called
h is 1
m is 2
s is 3

CCC
h is 12
m is 34
s is 56

DDD
The copy operator was invoked
h is 1
m is 2
s is 3
```

```
// cls2stct.cpp
#include <iostream.h>
// this program shows how a time class instance could be turned into a time structure
// In going from a class object to a struct instance, we typically
// will be going from a larger item to a smaller item because the struct
// doesn't carry around the entry points to the functions like the class
// does.  However, the struct being converted to could have more fields
// than the class object, and thus could be larger.  The point of these
// couple sentences is that we can make no assumptions about whether the
// thing we are casting to is smaller or larger than the thing we are
// casting from.

// In this program I cast a time to an XXX which has one field, two less than a time.
// I also cast a time to an MNOP which has four fields, one more than a time.

struct XXX
{
        int x;
};

void print_XXX(XXX x1)
{
        cout << "value of field x is " << x1.x << endl;
        cout << endl;
}

struct MNOP
{
        int m,n,o;
        float p;
};

void print_MNOP(MNOP mnop1)
{
        cout << "value of field m is " << mnop1.m << endl;
        cout << "value of field n is " << mnop1.n << endl;
        cout << "value of field o is " << mnop1.o << endl;
        cout << "value of field p is " << mnop1.p << endl;
        cout << endl;
}

class time
{
        private:
                int h,m,s;                      // hours, minutes, seconds

        public:
                void show_time();               // show what time it is
                time(int,int,int);
                operator XXX ();                // turn a time into an XXX;
                operator MNOP ();               // turn a time into an MNOP
};
```

```
void time::show_time()
{
	cout << "h is " << h << endl;
	cout << "m is " << m << endl;
	cout << "s is " << s << endl << endl;
}

// three int argument constructor
time::time(int a, int b, int c) : h(a), m(b), s(c)
{
	cout << "Three int arg constructor called \n";
}

return    class          function       argument
type      membership     name           list

time::operator XXX ()
{
	// I made up the cast so I get to decide how to do it
	// you could decide how to do it differently if you wrote the code
	// that's the point, you get to decide how to convert one to the other
	XXX temp;
	temp.x = this->h + this->m + this->s;
	return temp;
}

return    class          function       argument
type      membership     name           list

time::operator MNOP ()
{
	// I made up the cast so I get to decide how to do it
	// you could decide how to do it differently if you wrote the code
	// that's the point, you get to decide how to convert one to the other
	MNOP temp;
	temp.m = this->h;
	temp.n = this->m;
	temp.o = this->s;
	temp.p = ( (float) this->h )  / ( (float) this->m );
	return temp;
}
```

```
int main()
{
	// create a time
	time t1(4,2,1);
	t1.show_time();


	// this line will cast the time instance t1 into an XXX that can then be used in the
	// XXX one argument copy constructor
	// we did not write an XXX one argument copy constructor
	// furthermore, XXX is a struct type not a class
	// so where did the one arg copy constructor come from?
	// C and C++ both provide default constructors for data types that do a memberwise
	// copy from one instance of the type to another instance of that type
	// remember in an earlier program we learned that = was the only operator defined for C structs
	// well here is the consequence of having that = available
	XXX x1 = (XXX) t1;
	print_XXX(x1);

	// this line will cast the time instance t1 into an MNOP that can then be used in the
	// MNOP one argument copy constructor
	MNOP m1 = (MNOP) t1;
	print_MNOP(m1);
}
```

```
Output From Running Program
Three int arg constructor called
h is 4
m is 2
s is 1

value of field x is 7

value of field m is 4
value of field n is 2
value of field o is 1
value of field p is 2
```

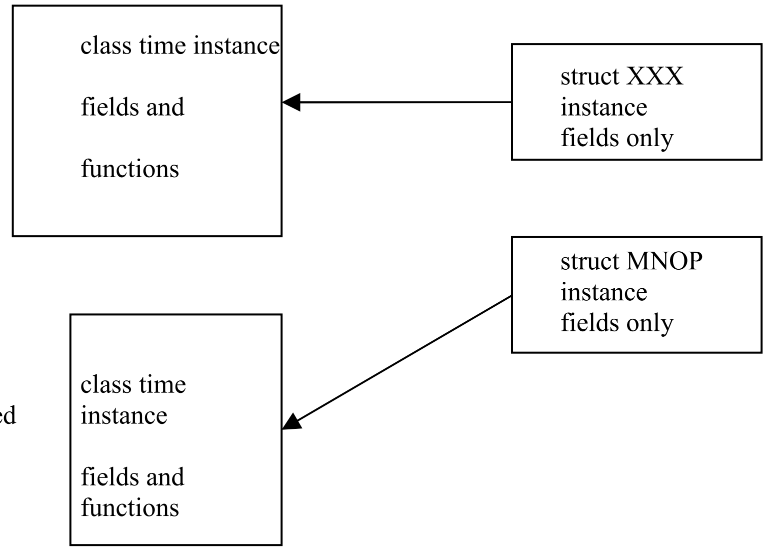


```
// stct2cls.cpp

#include <iostream.h>

// to convert a class to a struct, we need to build a constructor for
// the class, we do not get to use the casting syntax because C structs
// don't allow the inclusion of member functions and C++ classes do

// In the next programs, cls2cls1.cpp and cls2cls2.cpp, we show casting
// from a class to a class
// Casting from a class to a class can be different than casting from a struct to class
// C++ allows for special syntax and relationships between classes
// One of the relationships is called INHERITANCE
// We will cover inheritance in another section
// So the next two programs, after this one, deal with casting from class to class
// where there are and aren't inheritance relationships to INTRODUCE what the differences are
// between embedding and inheritance

struct XXX
{
	int x;
};

void print_XXX(XXX x1)
{
	cout << "value of field x is " << x1.x << endl;
	cout << endl;
}

struct MNOP
{
	int m,n,o;
	float p;
};

void print_MNOP(MNOP mnop1)
{
	cout << "value of field m is " << mnop1.m << endl;
	cout << "value of field n is " << mnop1.n << endl;
	cout << "value of field o is " << mnop1.o << endl;
	cout << "value of field p is " << mnop1.p << endl;
	cout << endl;
}
```

```
class time
{
	private:
		int h,m,s;			// hours, minutes, seconds

	public:
		void show_time();		// show what time it is
		time(int,int,int);
		time( XXX );	// build a time from an XXX, this isn't casting, it is  constructing
		time ( MNOP );	// build a time from an MNOP, this isn't casting, it is constructing
		operator XXX  ();// turn a time into an XXX;
		operator MNOP ();		// turn a time into an MNOP
};

void time::show_time()
{
	cout << "h is " << h << endl;
	cout << "m is " << m << endl;
	cout << "s is " << s << endl << endl;
}

// three int argument constructor
time::time(int a, int b, int c) : h(a), m(b), s(c)
{
	cout << "Three int arg constructor called \n";
}

time::operator XXX ()
{
	// I made up the cast so I get to decide how to do it
	// you could decide how to do it differently if you wrote the code
	// that's the point, you get to decide how to convert one to the other
	XXX temp;
	temp.x = this->h + this->m + this->s;
	return temp;
}

time::operator MNOP ()
{
	// I made up the cast so I get to decide how to do it
	// you could decide how to do it differently if you wrote the code
	// that's the point, you get to decide how to convert one to the other
	MNOP temp;
	temp.m = this->h;
	temp.n = this->m;
	temp.o = this->s;
	temp.p = ( (float) this->h )  / ( (float) this->m );
	return temp;
}

```

```
// This is the function that will construct "cast" an XXX into a time
return     class              function           argument
type       membership         name               list





time::time(XXX x1)
{
	cout << "The time(XXX) constructor was called \n";
	h = x1.x;		// WHY did I decide to do it this way?
	m = x1.x * 2;			// who knows, who cares, that's the point
	s = x1.x * 4;			// you are in control, you can write the constructor
}

// note that this function also has the name time, however it has a different argument list
// remember polymorphism?  As long as the functions have different argument lists they can have
// the same name.  In both cases we are casting to a time using the casting syntax so this is a place
// where having polymorphism available is an absolute necessity, Thank you Bjarne.
time::time(MNOP m1)
{
	cout << "The time(MNOP) constructor was called \n";
	h = m1.m;
	m = m1.n + m1.o;
	s = (int) m1.p + 10;
}

int main()
{
	// create a time
	time t1(4,2,1);
	t1.show_time();

	// create an XXX from a time
	XXX x1 = (XXX) t1;
	print_XXX(x1);

	// create an MNOP from a time
	MNOP m1 = (MNOP) t1;
	print_MNOP(m1);

	XXX x2 = { 40 };
	MNOP m2 = { 100,200,300,4.5 };

	// create a time from an XXX
	time t2(x2);
	t2.show_time();

	// create a time from an MNOP
	time t3(m2);
	t3.show_time();
}
```

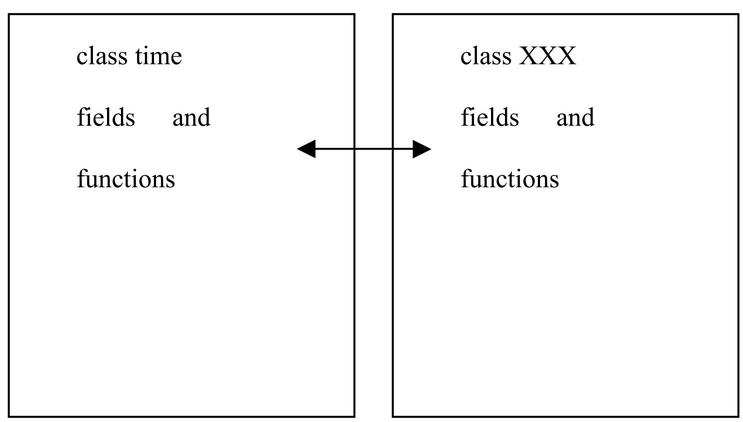
```
Output From Running Program

Three int arg constructor called
h is 4
m is 2
s is 1

value of field x is 7

value of field m is 4
value of field n is 2
value of field o is 1
value of field p is 2

The time(XXX) constructor was called
h is 40
m is 80
s is 160

The time(MNOP) constructor was called
h is 100
m is 500
s is 14
```



```
// cls2cls1.cpp

#include <iostream.h>

// this program converts one class to another class and back from the other class to the first class

// this presents a bit of a chicken and an egg problem
// For one class to know how to convert to the other class the other class
// has to already exist and yet for the other class to know how to convert
// to the one class the one class has to already exist.
// Well, not exactly.
// C++ allows you enter the NAME of a class into the name space of the
// compiler without actually implementing the class
// Thus you can register the casting functions in the class definitions
// and then, after both classes have been defined, implement the cast functions later

class XXX;          // Enter the names of the two classes into the name space of the compiler
class time;

class XXX
{
        private:
                int x;
        public:
                operator time();            // turn an XXX into a time
                XXX(int);                   // one int arg constructor
                XXX();                      // no arg constructor
                void show_XXX();
};

XXX::XXX() : x(10)
{
        cout << "NO arg XXX constructor called \n";
}
XXX::XXX(int xin) : x(xin)
{
        cout << "One int XXX constructor called \n";
}

void XXX::show_XXX()
{
        cout << "value of field x is " << x << endl;
        cout << endl;
}
```

```
class time
{
	private:
		int h,m,s;			// hours, minutes, seconds

	public:
		void show_time();		// show what time it is
		time(int,int,int);
		operator XXX ();		// turn a time into an XXX;
};

void time::show_time()
{
	cout << "h is " << h << endl;
	cout << "m is " << m << endl;
	cout << "s is " << s << endl << endl;
}

// three int argument constructor
time::time(int a, int b, int c) : h(a), m(b), s(c)
{
	cout << "Three int arg constructor called \n";
}
// now that we have both classes implemented
// we can write the casting routines

return	class		function	argument
type	membership	name		list




time::operator XXX ()
{
	// I made up the cast so I get to decide how to do it
	// you could decide how to do it differently if you wrote the code
	// that's the point, you get to decide how to convert one to the other
	XXX temp(this->h + this->m + this->s);
	return temp;
}

return	class		function	argument
type	membership	name		list




XXX::operator time()
{
	time temp(this->x, this->x * 2, this->x * 4);
	return temp;
}
```

```
int main()
{
            // create a time
            time t1(4,2,1);
            t1.show_time();
            cout << endl;

            // make an XXX out of it
            XXX x1 = (XXX) t1;              // this casts the time instance t1 into an XXX
            x1.show_XXX();
            t1.show_time();                 // t1 is not affected
            cout << endl;

            // create an XXX
            XXX x2(12);
            x2.show_XXX();
            cout << endl;

            // make a time out of it
            time t2 = (time) x2;            // this casts the XXX instance x2 into a time
            t2.show_time();
            x2.show_XXX();                  // x2 is not affected
            cout << endl;
}

Output From Running Program
Three int arg constructor called
h is 4
m is 2
s is 1


One int XXX constructor called
value of field x is 7

h is 4
m is 2
s is 1


One int XXX constructor called
value of field x is 12


Three int arg constructor called
h is 12
m is 24
s is 48

value of field x is 12
```



```
// cls2cls2.cpp

#include <iostream.h>

// this program converts one class to another class and back from the other
// class to the first class where an inheritance relationship exists

class XXX;
class time;

class XXX
{
	private:
		int x;
	public:
		void setx(int);	// access routine for x
		int getx();
		XXX(int);		// one int arg constructor
		void show_XXX();
};

XXX::XXX(int xin) : x(xin)
{
	cout << "One int XXX constructor called \n";
}

void XXX::setx(int i)
{
	x = i;
	return;
}

int XXX::getx()
{
	return x;
}

void XXX::show_XXX()
{
	cout << "value of field x is " << x << endl;
	cout << endl;
}
```

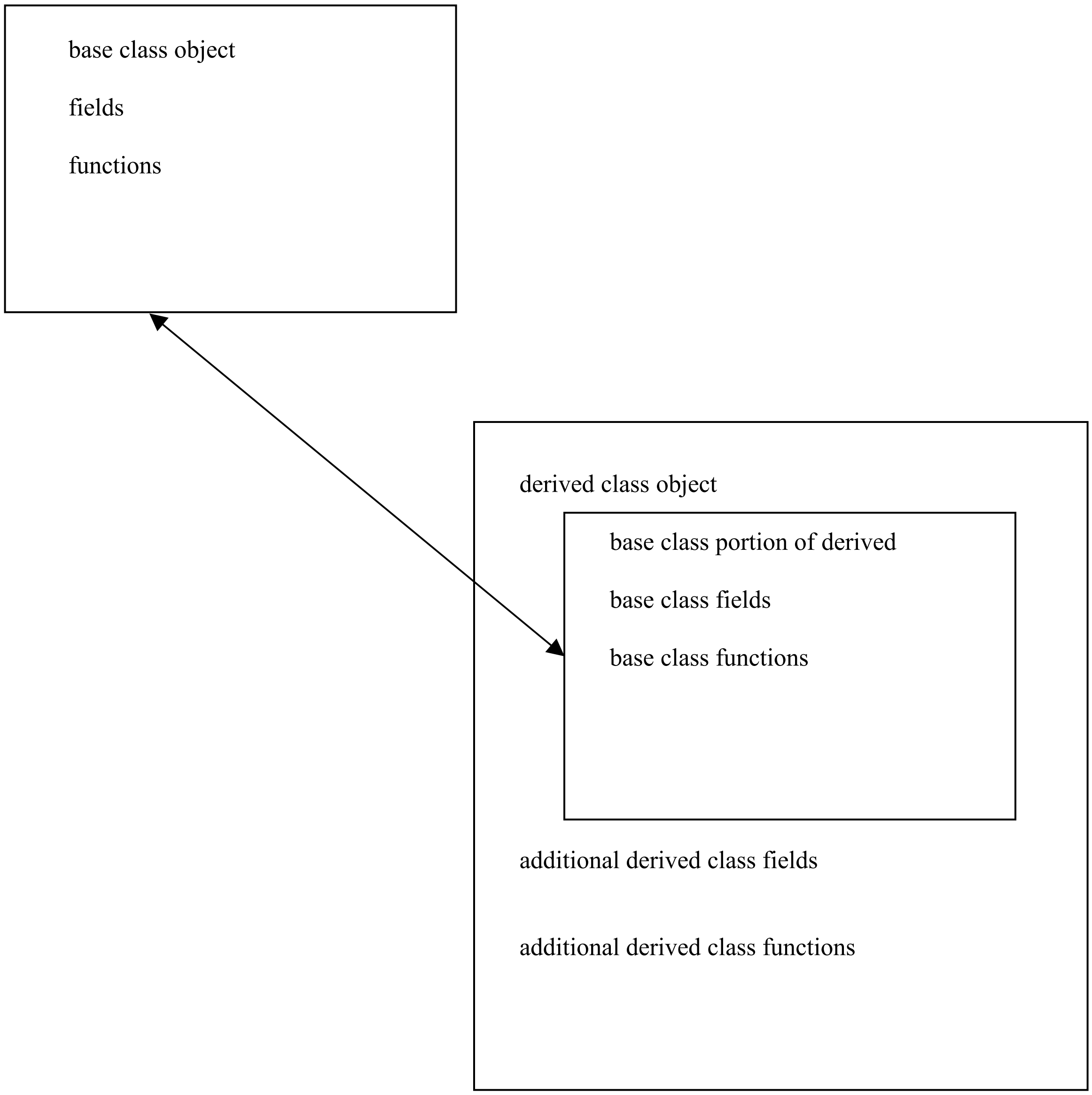
This class inherits from the XXX class
What that means is that a time IS an XXX   AND the extra stuff declared in here

We will do much more work with inheritance in later sections
I have included this program here to illustrate the casting differences when inheritance is involved
Perhaps you want to skip this program now and come back to it after you fully understand inheritance

```
class           inheritance            base
name            type                   class name







class time : public XXX
{
        private:
                int h,m,s;              // hours, minutes, seconds

        public:
                void show_time();       // show what time it is
                time(int,int,int);      // how to build a time from three ints
                time(XXX);              // how to build a time from an XXX

        // we don't have to write this routine anymore when inheritance is involved
        // C++ already has a method, that you should just accept, for converting a derived instance
        // into a base instance.  It hacks off all the extra parts of the derived portion and just leaves
        // the base class portion of the derived object.  This is known as the Manassis approach.
        //operator XXX ();                // turn a time into an XXX;
        time operator = (time);         // copy one time to another time
};

time time::operator = (time t1)
{
        this->h = t1.h;
        this->m = t1.m;
        this->s = t1.s;
        this->setx(-99);
        return *this;
}

void time::show_time()
{
        cout << "the x part of the base portion is " <<
        this->getx() << endl;
        cout << "h is " << h << endl;
        cout << "m is " << m << endl;
        cout << "s is " << s << endl << endl;
}
```

```
// three int argument constructor
time::time(int a, int b, int c) : h(a), m(b), s(c), XXX(4567)
{	cout << "Three int arg constructor called \n";	}

#if 0
WE DON'T HAVE TO WRITE THIS ROUTINE, C++ ALREADY KNOWS HOW TO DO THIS
XXX time::operator XXX ()
{
	// I made up the cast so I get to decide how to do it
	// you could decide how to do it differently if you wrote the code
	// that's the point, you get to decide how to convert one to the other
	XXX temp(this->getx() );
	return temp;
}
#endif

// We still write the XXX to time class cast "Constructor" in this case.
// we can build a derived object from a base object by adding on the extra fields and entry points
time::time(XXX inx) : XXX(0)
{
	// leave x with whatever value it had
	// we are basically building a new XXX AND a time
	this->setx(inx.getx() );
	h = -1;   m = -2;   s = -3;
}

int main()
{
	XXX x1(12);	// create an XXX
	x1.show_XXX();
	cout << endl;

	time t1(4,2,1);	// create a time
	t1.show_time();
	cout << endl;

	// create an XXX out of a time
	cout << endl << "t1 and x1 before assignment " << endl;
	t1.show_time();
	x1.show_XXX();
	// x1 = (XXX) t1;		// don't need the cast syntax
	x1 = t1;			// C++ just knows how to do this, because C++ has the = operator
					//defined between any two instances of the same type, and since a
					// time IS an XXX, it copies the like fields and drops the unlike
	cout << endl << "t1 and x1 after assignment " << endl;
	t1.show_time();
	x1.show_XXX();

	time t2(12,34,56);
	t2.show_time();
	t2 = t1;
	t2.show_time();
}
```

```
Output From Running Program

One int XXX constructor called
value of field x is 12


One int XXX constructor called
Three int arg constructor called
the x part of the base portion is 4567
h is 4
m is 2
s is 1



t1 and x1 before assignment
the x part of the base portion is 4567
h is 4
m is 2
s is 1

value of field x is 12


t1 and x1 after assignment
the x part of the base portion is 4567
h is 4
m is 2
s is 1

value of field x is 4567

One int XXX constructor called
Three int arg constructor called
the x part of the base portion is 4567
h is 12
m is 34
s is 56

the x part of the base portion is -99
h is 4
m is 2
s is 1
```


base class object
fields
functions
derived class object
base class portion of derived
base class fields
base class functions
additional derived class fields
additional derived class functions

Section 6

This section is about inheritance.
C++ is all about classes.
Once you know about classes, then you can get sophisticated and learn about inheritance.

Object oriented analysis and design requires classes.
Good object oriented analysis and design requires inheritance.
It is a thorough understanding of what inheritance is and what it can do for you that takes you from the ranks of the novice C++ programmer to the good C++ programmer.

If you want to use C++ as more than just a better C compiler, then focus your study on classes and inheritance.

This section explains what inheritance is not. Many people EMBED things and think they're doing inheritance. They aren't.

Inheritance implements the "IS A" relationship.
Embedding implements the "Has A" relationship.

For example,
- a car has an engine.
- a car is not an engine.
- therefore a car does not inherit from an engine.

- a car is a vehicle.
- therefore a car inherits from a vehicle.
- a base class of vehicle could be established where the common fields and functions for all vehicles could be isolated. This would greatly reduce the number of places that the code for vehicle would need to be maintained.
- the deriving classes, cars and trucks etc... would all be considered vehicles AND the extra fields and functions that make them different.

- Deriving classes are also able to redefine a field or function found in a base class.
- Without this last point, inheritance would be too restrictive.

This section shows you **EVERY** rule and regulation and possibility of inheritance.

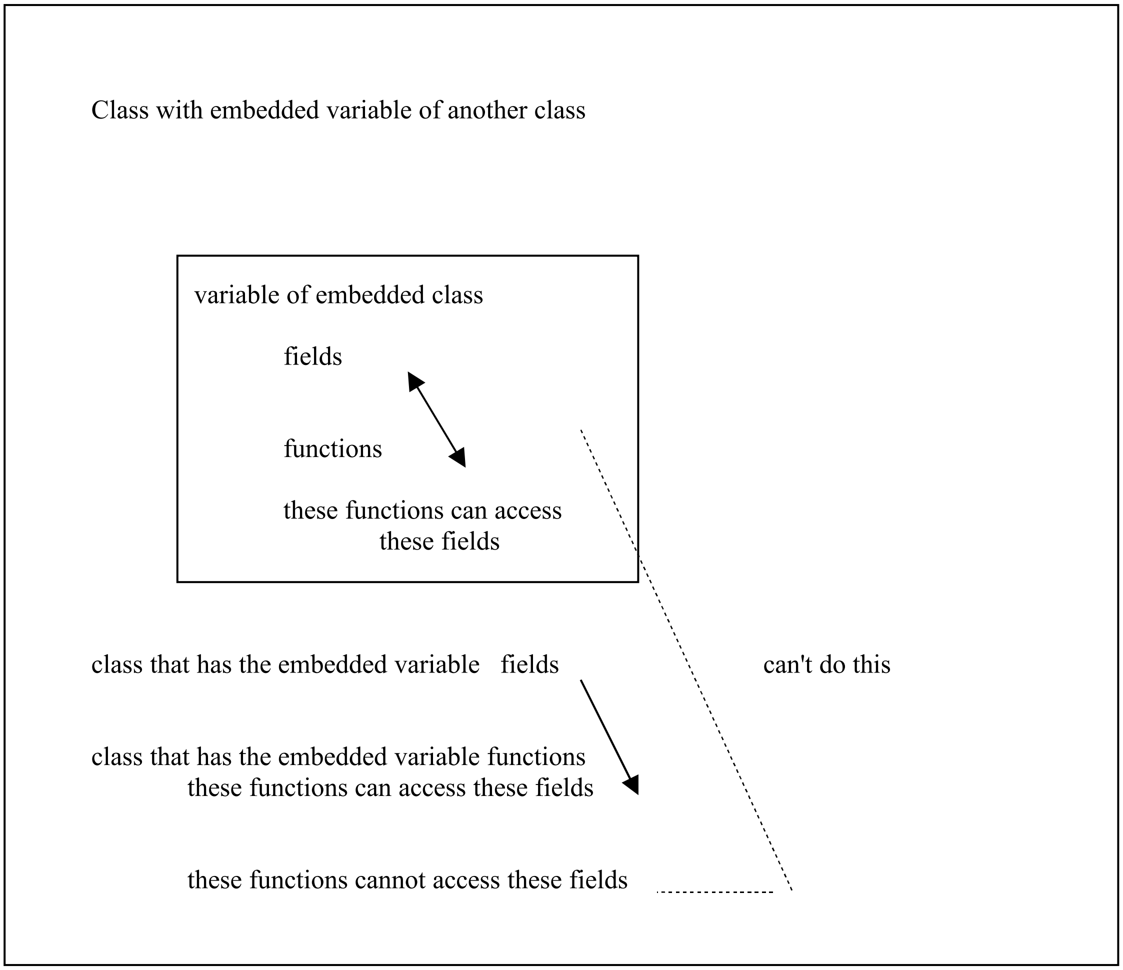
embed.cpp	one class embedded in another, not an inheritance relationship
base_a.cpp	a class that can stand by itself and be used as a base class
tstbasea.cpp	program to exercise base_a
puba.cpp	a class that derives using public inheritance from base class coded in base_a.cpp
pubinht.cpp	program to exercise class puba
priva.cpp	a class that derives using private inheritance from base class coded in base_a.cpp
privinht.cpp	program to exercise derived class priva
polyinh1.cpp	polymorphism in an ineritance relationship

```
// embed.cpp
// this program illustrates the fact that one class can be embedded inside
// of another.  This is NOT inheritance.  We will see inheritance very soon

// Each of the functions does something simple and prints out that it was
// called.  What each function does in this example is not important
// What functions can and can't be called from where and by whom IS important
#include <iostream.h>

class a
{
	private:
		int a;
	public:
		a(void);		// no arg constructor
		void seta(int);
		void printa(void);
};

a::a() : a(23)
{
	cout << "No arg constructor for a was called \n";
}

void a::seta(int x )
{
	cout << "a::seta function was called \n";
	a = x;
}

void a::printa(void)
{
	cout << "a::printa function called, a is " << a << endl;
}
```

```
This class is going to have an instance of an A embedded in it
Question?       What functions of a can the b instance call?
                There are two places to examine.
                From within the member code of a b instance
                From user code, i.e. main, that is not part of the b member functions

class b
{
        private:
        int     b;      // b is an integer
        a       a1;     // a1 is of type a   B HAS AN INSTANCE OF AN A EMBEDDED IN IT
                        // a1 is of a user defined type, not a pre-defined type
        public:
                b(void);                // no arg constructor
                void setb(int);         // function to set b
                void printb(void);
                void setb_and_a1(int,int);
};

// this is the no arg constructor for b
// before the first executable statement in the constructor can be executed
// ALL of the memory for the instance must be allocated
// The system will cause the no arg constructor for type a to be called because a1 is of type a
b::b() : b(-97)
{
        // before the first line of the constructor is called, the memory for the a that is
        // part of this b instance, is allocated and initialized using the appropriate constructor
        // in this case we gave no information about what constructor to use for the a portion
        // therefore, the no argument constructor for the a part was called
        cout << "No arg constructor for b was called \n";
}

// function to set the integer portion of b
void b::setb(int x )
{
        cout << "b::setb function was called \n";
        b = x;  // You can do this because b is private data of the class b
}

// function to set BOTH fields
void b::setb_and_a1(int x, int y)
{
        b = x;                  // you can do this
        //a = y;                // you cannot do this because a is private data of a1
        a1.seta(y);             // you are forced to do this, call a PUBLIC member function of an a instance
}
```

```
void b::printb(void)
{
	// you can do this
	cout << "b::printb function called, b is " << b << endl;

	// you can't do this because a1 is private data of the a class and this is not a
	// member function of the a class
	// cout << "b::printb function called, a1 is " << a1 << endl;

	// you are forced to do this, call the PUBLIC member function that knows how to print an a
	a1.printa();
}

main()
{
	// declare a variable of type a
	a	abc;
	abc.printa();	// a is a class in and of itself, it's variables can be accessed by the functions
	abc.seta(88);	// assigned to do that job
	abc.printa();

	// declare a variable of type b, a variable of type b has a variable of type a embedded in it
	b	xyz;
	xyz.printb();
	// xyz.printa();	// you can't do this, printa is a member function of class a, not of class b
			// if class b had INHERITED from class a, then you COULD have made
			// this call, we will see this in the inheritance section

	xyz.setb(34);	// you can do this
	// xyz.seta(345);	// you can't do this because seta is a member function of class a, not class b

	xyz.setb_and_a1(-23,-654);		// we need a special function to access
					// the a part of the b class variable
	xyz.printb();
}

Output From Running Program

No arg constructor for a was called
a::printa function called, a is 23
a::seta function was called
a::printa function called, a is 88

No arg constructor for a was called		This happened on line	b xyz;
No arg constructor for b was called
b::printb function called, b is -97
a::printa function called, a is 23
b::setb function was called
a::seta function was called
b::printb function called, b is -23
a::printa function called, a is -654
```

Embedding Diagram



```
// base_a.cpp
#ifndef BASEACPP
#define BASEACPP
the first time this file is inclued into a compile list, the COMPILER variable BASEACPP is not "defined",
therefore #ifndef is TRUE therefore we compile the source contained within the #ifndef #endif block.  In
subsequent inclusions for the same compile list for the same load module, the compiler variable
BASEACPP is "defined", because the #define BASEACPP line was encountered.  Therefore the #ifndef
BASEACPP is FALSE and the source code is not included again.

// this file lays out a class that is destined to be used as a BASE class
// Just because something will be used as a base class doesn't mean that it can't be used
// all by itself as a regular class.  There is no special syntax for classes that will be base classes
// There exist in and of and by themselves as regular classes UNTIL some other class issues special
// syntax that identifies that they wish to derive from some other class.  At that point they become a
// base class.  However, even after they have been identified by an inheritor as its base class, they can
// still function as regular classes.  Being a base class places no restrictions on a class

// Any class may have three portions:
//          private:
//          protected:
//          public:

// Each of these sections may have data and functions

// We start our derivation of inheritance by examining what these three
// sections mean with respect to the visibility and accessebility of the
// variables and subroutines declared in each area

// There are two places where the visibility needs to be examined
//          in code that is part of the subroutines that are part of the class
//          in code that is not part of the subroutines that are part of the class
//          (i.e.  in main )

#include <iostream.h>

class a
{
        private:
                int     a_priv_var;             // a private variable
                void    a_private_f1();         // a private function
        protected:
                int     a_prot_var;             // a protected variable
                void    a_protected_f1(); // a protected function
        public:
                int a_pub_var;                  // a public variable
                void    a_public_f1();          // a public function

                a();                            // no arg constructor
                void a_print();                 // another public function
                void a_call_em();       // another public function
};
```

```
// no argument constructor
a::a()
{
	cout << "The a no arg constructor has been invoked \n";
	a_priv_var = 1;
	a_prot_var = 2;
	a_pub_var  = 3;
}

// this is a private member function of the a class
// from inside a private member function you can access
// private, protected, public data
void a::a_private_f1()
{
	cout << "In a_private_f1 "
	<<	a_priv_var	// we have access to private data
	<< " "
	<<	a_prot_var	// we have access to protected data
	<< " "
	<<	a_pub_var	// we have access to public data
	<< " "
	<<	endl;
}

// this is a protected member function of the a class
// from inside a protected member function you can access
// private, protected, public data
void a::a_protected_f1()
{
	cout << "In a_protected_f1 "
	<<	a_priv_var	// we have access to private data
	<< " "
	<<	a_prot_var	// we have access to protected data
	<< " "
	<<	a_pub_var	// we have access to public data
	<< " "
	<<	endl;
}

// this is a public member function of the a class
// from inside a public member function you can access
// public, protected and private data
void a::a_public_f1()
{
	cout << "In a_public_f1 "
	<<	a_priv_var	// we have access to private data
	<< " "
	<<	a_prot_var	// we have access to protected data
	<< " "
	<<	a_pub_var	// we have access to public data
	<< " "
	<<	endl;
}
```

```
// this is a public member function that can access all three types of data
void a::a_print()
{
	cout << "In a_print "
	<<	a_priv_var		// we have access to private data
	<< " "
	<<	a_prot_var		// we have access to protected data
	<< " "
	<<	a_pub_var		// we have access to public data
	<< " "
	<<	endl;
}

// this is a public member function
// from a public member function you may access the
// public, protected and private member functions
void a::a_call_em()
{
	this->a_private_f1();
	this->a_protected_f1();
	this->a_public_f1();
}

// this class definition will be used in several C++ programs
// therefore it is left by itself in this file with no main
// immediately associated with it
// it will be compiled with other sources to form a complete program
// this #endif is the end of the #ifndef #endif block used to "protect" this file from multiple inclusions
#endif
```

class a data	class a functions
private data	private functions
protected data	protected functions
public data	public functions

private function
may access:
private, protected, public data
private, protected, public functions

protected function
may access:
private, protected, public data
private, protected, public functions

public function
may access:
private, protected, public data
private, protected, public functions

```
// tstbasea.cpp
// this program exercises the base class a as a stand alone class
// it will be used as the base class for other objects
// it is purely a teaching class object
// it has public, private and protected data and functions
// just because it is going to be a base class doesn't mean it can't
// function as a regular class all by itself.

// the first thing you have to be completely clear about is who can
// access the public, private and protected elements of the class

// there are two places we have to examine each of these three areas from:
//		from within the code of a member function
//		from outside the scope of any member function (i.e. in main)

// The basic access rules are:
//		Any member function can access any public, private or protected
//			data element of its own class
//		Any member function can access any public, private or protected
//			member function of its own class

//		No non member function code can access any private or protected
//			data element of any class
//		No non member function code can access any private or protected
//			member function of any class

//		Any non member function code can access any public data of any class
//		Any non member function code can access any public function of any class
```

```
#include "base_a.cpp"
#include <iostream.h>

main()
{
	// You can create an instance of a base class object
	a  obj1;
	obj1.a_print();

	// You can call the public member functions
	obj1.a_print();
	obj1.a_public_f1();

	// You cannot call the protected member functions from outside
	// a member function of the class
	// obj1.a_protected_f1();

	// You cannot call the private member functions from outside
	// a member function of the class
	// obj1.a_private_f1();

	// You can call a public member function that calls the
	// public, protected and private member functions
	// this is the only way you  are going to get access to the protected and private data of the class
	obj1.a_call_em();

	// examine the accessability of the variables
	// obj1.a_priv_var = 1;	// this variable is not visible from non member functions
	// obj1.a_prot_var = 1;	// this variable is not visible from non member functions
	obj1.a_pub_var  = 1;	// this variable, a public variable, is visible
}

Output From Running Program

The a no arg constructor has been invoked
In a_print 1 2 3
In a_print 1 2 3
In a_public_f1 1 2 3
In a_private_f1 1 2 3
In a_protected_f1 1 2 3
In a_public_f1 1 2 3
```

class a data	class a functions
private data	private functions
protected data	protected functions
public data	public functions

private function
may access:
private, protected, public data
private, protected, public functions

protected function
may access:
private, protected, public data
private, protected, public functions

public function
may access:
private, protected, public data
private, protected, public functions

main()

may access:
- public data of any class
- public functions of any class

may not access:
- protected data of any class
- protected functions of any class
- private data of any class
- private functions of any class

```
// puba.cpp

#ifndef PUBACPP
#define PUBACPP

// this file introduces the class puba
// it INHERITS from the class a
// it inherits using the public inheritance mechanism
// there are two types of inheritance, public and private

// Inheritance means that an object of type puba is an object of type a AND
// some extra stuff


        base class portion
        base class fields               base class functions

-------------------------------------------------------------------
        derived class portion
        derived class fields            derived class functions


// there are visibility issues in a derived class object
#include <iostream.h>
#include "base_a.cpp"
// the class name is puba
// it inherits from a
// using the public mechanism
class puba : public a
{
        private:
                int puba_priv_var;              // private data field
                void puba_private_f1();         // private member function
        protected:
                int puba_prot_var;              // protected data field
                void puba_protected_f1();       // protected member function
        public:
                int puba_pub_var;               // public data field
                void puba_public_f1();          // public member function
        puba();                                 // constructor
                void puba_print();              // public member function
                void puba_call_em();            // public member function
};
```

```
// Notice that we are specifying the way in which we want the three variables of the derived portion
// to be initialized.  However, we have not said how we want the variables of the base class portion
// of the object to be initialized.  Because of this, the default, no arg constructor for the base class
// will be invoked

puba::puba() : puba_priv_var(10), puba_prot_var(20), puba_pub_var(30)
{
	// before the first line in the derived class constructor is executed
	// ALL the memory for the entire object has to be allocated
	// since an object of type puba IS an object of type a AND some other
	// stuff, the constructor for the a portion of puba object will be
	// invoked before the first executable statement in this constructor
	// is executed

	// you will notice the print out from the a class constructor before
	// the print out from this constructor
	cout << "In puba no arg constructor \n";
}

// this is a private member function of the derived class
// it has access to the public, protected and private data of the puba class
// it has access to the public and protected variables of the a class
// it DOES NOT have access to the private data of the a class
void puba::puba_private_f1()
{
	cout << "In puba_private_f1 " << endl;
	// examine accessability to the puba portion variables
	cout	<< "puba_priv_var is: "
	<< puba_priv_var
	<< " puba_prot_var is: "
	<< puba_prot_var
	<< " puba_pub_var  is: "
	<< puba_pub_var
	<< endl;

	// remember what happened when we EMBEDDED an instance of one class in another class?
	// none of the data was visible to the class that had the instance embedded in it.
	// Now that we are doing INHERITANCE, the public and protected portions will be visible
	// examine accessability to variables of base class
	cout	// << a_priv_var  // this variable is not visible, it is private data of the base class
	<< a_prot_var			// this variable is visible, it is protected data of the base class
	<< " "
	<< a_pub_var			// this variable is visible, it is public data of the base class
	<< endl;
}
```

```
// this is a protected member function of the derived class
// it has access to the public, protected and private data of the puba class
// it has access to the public and protected variables of the a class
// it DOES NOT have access to the private data of the a class
void puba::puba_protected_f1()
{
	cout << "In puba_protected_f1 " << endl;
	// examine accessability to variables of the puba portion
	cout	<< "puba_priv_var is: "
	<< puba_priv_var
	<< " puba_prot_var is: "
	<< puba_prot_var
	<< " puba_pub_var  is: "
	<< puba_pub_var
	<< endl;

	// examine accessability to variables of base class
	cout	// << a_priv_var	// this variable is not visible
	<< a_prot_var
	<< " "
	<< a_pub_var
	<< endl;


// this is a public member function of the derived class
// it has access to the public, protected and private data of the puba class
// it has access to the public and protected variables of the a class
// it DOES NOT have access to the private data of the a class
void puba::puba_public_f1()
{
	cout << "In puba_public_f1 " << endl;
	// examine accessability of puba portion of the class variables
	cout	<< "puba_priv_var is: "
	<< puba_priv_var
	<< " puba_prot_var is: "
	<< puba_prot_var
	<< " puba_pub_var  is: "
	<< puba_pub_var
	<< endl;

	// examine accessability to variables of base class
	cout	// << a_priv_var	// this variable is not visible
	<< a_prot_var
	<< " "
	<< a_pub_var
	<< endl;
}
```

```
// this public member function has access to all three types of data that
// are part of the puba portion of a puba object
// it has access to the public and protected data of the base class portion
// of the puba object
// it DOES NOT have access to the private data of the base class portion
// of the puba object
void puba::puba_print()
{
// from inside a member function, public, protected or private
// you can access the public, protected and private data elements
        cout    << "puba_priv_var is: "
        << puba_priv_var
        << " puba_prot_var is: "
        << puba_prot_var
        << " puba_pub_var  is: "
        << puba_pub_var
        << endl;

        cout    // << a_priv_var        // this variable is not visible
        << a_prot_var
        << " "
        << a_pub_var
        << endl;
}

// this is a public member function
// it can call the private, protected and public member functions of the
//      puba portion of the puba class
// it can call the protected and public member functions of the a portion
//      of the puba class
void puba::puba_call_em()
{
        // from inside a member function, you can call any other
        // member function of  its own class
        this->puba_private_f1();
        this->puba_protected_f1();
        this->puba_public_f1();

        // examine accessability of functions in base class
        // from inside a member function that derives from another class, you may access the protected
        // and public functions of the base class.  From inside a member function that derives from
        another
        // class you MAY NOT access the private functions of the base class
        // this->a_private_f1();        // this function is not visible
        this->a_protected_f1();
        this->a_public_f1();
}
#endif
```

Derived Class Object

Derived Class Data	Derived Class Functions
private	private
protected	protected
public	public

Base Class Portion of Derived Class Object

Derived Class Data	Derived Class Functions
private	private
protected	protected
public	public

USING PUBLIC INHERITANCE

derived class functions may access:

derived class portion, private, protected, public data
derived class portion, private, protected, public functions

base class portion, protected and public data
base class portion, protected and public functions

main()

may access public functions and data of derived class portion
may access public functions and data of base class portion directly
may access public functions and data of base class via instance of derived class

may not access protected or private portions of derived class
may not access protected or private portions of base class

```
// pubinht.cpp

// this program exercises the class puba  which is derived from
// the base class a
// it uses public inheritance
// because public inheritance is used, certain rules apply to what code
// has access to the public, private and protected data and member functions
// of the base class
// the access rules for the derived class are also spelled out in this program

// the base class has three areas to examine, public, protected, private
// the derived class has three areas to examine, public, protected, private

// there are three places we have to examine each of these three areas from:
//      from within the code of a member function of the base class
//      from within the code of a member function of the deriving class
//      from outside the scope of any member function (i.e. in main)

// The basic access rules are:
//      Any member function can access any public, private or protected
//              data element of its own class
//      Any member function can access any public, private or protected
//              member function of its own class

//      No non member function code can access any private or protected
//              data element of any class
//      No non member function code can access any private or protected
//              member function of any class

//      Any non member function code can access any public data
//      Any non member function code can access any public function
```

```
#include <iostream.h>
#include "base_a.cpp"
#include "puba.cpp"

main()
{
        // You can create an instance of a base class object
        a  obj1;
        obj1.a_print();

        // You can call the public member functions of the base class object
        obj1.a_print();
        obj1.a_public_f1();

        // You cannot call the protected member functions from outside
        // a member function of the class
        // obj1.a_protected_f1();

        // You cannot call the private member functions from outside
        // a member function of the class
        // obj1.a_private_f1();

        // You can call a public member function that calls the
        // public, protected and private member functions
        obj1.a_call_em();

        puba    obj2;               // you can create an object of derived class
        obj2.puba_print();          // we can call the public print function of the derived class
        obj2.a_print();             // we can call the public print function of
                                    // the class from which it is derived

        obj2.puba_public_f1();      // can call the public function
                                    // of the puba portion of the class
        // obj2.puba_protected_f1();// protected function not visible
        // obj2.puba_private_f1();  // private   function not visible

        obj2.a_public_f1();                 // can call the public function of the base class portion of an
                                            // object of type puba
        // obj2.a_protected_f1();           // protected function not visible
        // obj2.a_private_f1();             // private   function not visible

        obj2.puba_call_em();                // can call public function of the derived portion of the class
```

```
    // examine accessability of member variables of base object
    // via an instance of the base object
    // obj1.a_priv_var = 100;   // private variable is not visible
    // obj1.a_prot_var = 100;   // protected variable is not visible
    obj1.a_pub_var  = 100;      // public data is visible

    // examine accessability of member variables of derived class portion
    // of puba object via an instance of the derived object
    // obj2.puba_priv_var = 200;          // private variable is not visible
    // obj2.puba_prot_var = 200;          // protected variable is not visible
    obj2.puba_pub_var  = 200;             // public data is visible

    // examine the accessability of member variables of base portion of
    // puba object via an instance of a derived object
    // obj2.puba_priv_var = 300;          // variable is not visible
    // obj2.puba_prot_var = 300;          // variable is not visible
    obj2.puba_pub_var  = 300;
}
```

```
Output From Running Program

The a no arg constructor has been invoked
In a_print 1 2 3
In a_print 1 2 3
In a_public_f1 1 2 3
In a_private_f1 1 2 3
In a_protected_f1 1 2 3
In a_public_f1 1 2 3

The a no arg constructor has been invoked
In puba no arg constructor
puba_priv_var is: 10 puba_prot_var is: 20 puba_pub_var  is: 30
2 3

In a_print 1 2 3
In puba_public_f1
puba_priv_var is: 10 puba_prot_var is: 20 puba_pub_var  is: 30
2 3

In a_public_f1 1 2 3
In puba_private_f1
puba_priv_var is: 10 puba_prot_var is: 20 puba_pub_var  is: 30
2 3

In puba_protected_f1
puba_priv_var is: 10 puba_prot_var is: 20 puba_pub_var  is: 30
2 3

In puba_public_f1
puba_priv_var is: 10 puba_prot_var is: 20 puba_pub_var  is: 30
2 3

In a_protected_f1 1 2 3
In a_public_f1 1 2 3
```

```
// priva.cpp

#ifndef PRIVACPP
#define PRIVACPP

// this file introduces the class priva
// it INHERITS from the class a
// it inherits using the private inheritance mechanism
// there are two types of inheritance, public and private

// Inheritance means that an object of type priva is an object of type a AND some extra stuff


        base class portion
        base class fields               base class functions

        ---------------------------------------------------------------
        derived class portion
        derived class fields            derived class functions



// there are visibility issues in a derived class object
#include <iostream.h>
#include "base_a.cpp"
// the class name is priva
// it inherits from a
// using the private mechanism
class priva : private a
{
        private:
                int priva_priv_var;             // private data field
                void priva_private_f1();        // private member function
        protected:
                int priva_prot_var;             // protected data field
                void priva_protected_f1();      // protected member function
        public:
                int priva_pub_var;              // public data field
                void priva_public_f1();         // public member function
                priva();                        // constructor
                void priva_print();             // public member function
                void priva_call_em();           // public member function
};
```

```
//this is the constructor for the derived class object
//it should specify how it wants the construction of its base class portion to occur
//it doesn't make this specification therefore the default no arg constructor is called

priva::priva() : priva_priv_var(10), priva_prot_var(20), priva_pub_var(30)
{
        // before the first line in the derived class constructor is executed
        // all the memory for the entire object has to be allocated
        // since an object of type priva IS an object of type a AND some other
        // stuff, the constructor for the a portion of priva object will be
        // invoked before the first executable statement in this constructor
        // is executed

        // you will notice the print out from the a class constructor before
        // the print out from this constructor
        cout << "In priva no arg constructor \n";
}

// this is a private member function of the derived class
// it has access to the public, protected and private data of the priva class
// it has access to the public and protected variables of the a class
// it DOES NOT have access to the private data of the a class
void priva::priva_private_f1()
{
        cout << "In priva_private_f1  " << endl;
        // examine accessability to the priva portion variables
        cout    << "priva_priv_var is: "  // member functions may access private data of their own class
        << priva_priv_var
        << " priva_prot_var is: "
        << priva_prot_var           // member functions may access protected data of their own class
        << " priva_pub_var  is: "
        << priva_pub_var            // member functions may access public data of their own class
        << endl;

        // examine accessability to variables of base class
        cout    // << a_priv_var      // this variable is not visible, member functions of the derived
                                      // class may not access private data of the base class
        << a_prot_var                 // member functions of the derived class may access protected
                                      // data of the base class
        << " "
        << a_pub_var                  // member functions of the derived class may access public
                                      // data of the base class
        << endl;
}
```

```
// this is a protected member function of the derived class
// it has access to the public, protected and private data of the priva class
// it has access to the public and protected variables of the base class a
// it DOES NOT have access to the private data of the base class a
void priva::priva_protected_f1()
{
	cout << "In priva_protected_f1 " << endl;
	// examine accessability to variables of the priva portion
	cout	<< "priva_priv_var is: "
	<< priva_priv_var
	<< " priva_prot_var is: "
	<< priva_prot_var
	<< " priva_pub_var  is: "
	<< priva_pub_var
	<< endl;

	// examine accessability to variables of base class
	cout	// << a_priv_var		// this variable is not visible
	<< a_prot_var
	<< " "
	<< a_pub_var
	<< endl;
}

// this is a public member function of the derived class
// it has access to the public, protected and private data of the priva class
// it has access to the public and protected variables of the base class a
// it DOES NOT have access to the private data of the base class a
void priva::priva_public_f1()
{
	cout << "In priva_public_f1 " << endl;
	// examine accessability of priva portion of the class variables
	cout	<< "priva_priv_var is: "
	<< priva_priv_var
	<< " priva_prot_var is: "
	<< priva_prot_var
	<< " priva_pub_var  is: "
	<< priva_pub_var
	<< endl;

	// examine accessability to variables of base class
	cout	// << a_priv_var	// this variable is not visible
	<< a_prot_var
	<< " "
	<< a_pub_var
	<< endl;
}
```

```
// this public member function has access to all three types of data that
// are part of the priva portion of a priva object
// it has access to the public and protected data of the base class portion
// of the priva object
// it DOES NOT have access to the private data of the base class portion
// of the priva object
void priva::priva_print()
{
        // from inside a member function, public, protected or private
        // you can access the public, protected and private data elements
        cout    << "priva_priv_var is: "
        << priva_priv_var
        << " priva_prot_var is: "
        << priva_prot_var
        << " priva_pub_var  is: "
        << priva_pub_var
        << endl;

        cout    // << a_priv_var      // this variable is not visible
        << a_prot_var
        << " "
        << a_pub_var
        << endl;
}

// this is a public member function
// it can call the private, protected and public member functions of the
//      priva portion of the priva class
// it can call the protected and public member functions of the a portion, the base class portion
//      of the priva class
// it cannot call the private member functions of the a portion, the base class portion  of the
//          priva class
void priva::priva_call_em()
{
        // from inside a member function, you can call any other member function of this class
        this->priva_private_f1();
        this->priva_protected_f1();
        this->priva_public_f1();

        // examine accessability of functions in base class
        // this->a_private_f1();                // this function is not visible
        this->a_protected_f1();                 // this function is visible
        this->a_public_f1();                    // this function is visible
}
#endif
```

// privinht.cpp

```
// this program exercises the class priva derived from the base class a
// it uses private inheritance
// because private  inheritance is used, certain rules apply to what code
// has access to the public, private and protected data and member functions
// of the base class
// the access rules for the derived class are also spelled out in this program

USING PRIVATE INHERITANCE
Member functions of the deriving class can access public and protected data and functions of the BASE
class.  Member functions of the deriving class cannot access private data or functions of the base class.
Non member function code is the code that USERS of the class would write.  Objects of the derived type,
in non-member function code (i.e. main() ) MAY NOT access public functions of the base class.  THIS is
what separates Public from Private inheritance.

// the base class has three areas to examine, public, protected, private
// the derived class has three areas to examine, public, protected, private

// there are three places we have to examine each of these three areas from:
//      from within the code of a member function of the base class
//      from within the code of a member function of the deriving class
//      from outside the scope of any member function (i.e. in main)

// The basic access rules are:
//      Any member function can access any public, private or protected
//              data element of its own class
//      Any member function can access any public, private or protected
//              member function of its own class

//      No non member function code can access any private or protected
//              data element of any class
//      No non member function code can access any private or protected
//              member function of any class

//      Any non member function code can access any public data of any class directly
//      Any non member function code can access any public function of any class directly

//          No non member function code can access the public data of any class that serves as
//          the base class for a class that derived PRIVATELY from it by using an instance of the
//          derived class.  However, that public data can be accessed directly through an instance of the
//          Base class itself.
```

```
#include <iostream.h>
#include "base_a.cpp"
#include "priva.cpp"

main()
{
        // You can create an instance of a base class object
        a  obj1;
        obj1.a_print();

        // You can call the public member functions of the base class object
        obj1.a_print();
        obj1.a_public_f1();

        // You cannot call the protected member functions from outside
        // a member function of the class
        // obj1.a_protected_f1();

        // You cannot call the private member functions from outside
        // a member function of the class
        // obj1.a_private_f1();

        // You can call a public member function that calls the
        // public, protected and private member functions
        obj1.a_call_em();

        priva      obj2;                  // you can create an object of derived class
        obj2.priva_print();               // we can call the public print function of the derived class
        // obj2.a_print();                // we CANNOT call the public print function of the
                                          // base class via a derived class object
                                          //  BECAUSE WE USED PRIVATE INHERITANCE

        obj2.priva_public_f1();           // can call the public function of the priva portion of the class
        // obj2.priva_protected_f1();     // protected function not visible
        // obj2.priva_private_f1();       // private   function not visible

        // obj2.a_public_f1();            // CANNOT call the public function
                                          // of the base class portion of an  object of type priva
                                          // via an instance of a priva object
                                          // we could call it directly via an instance of an a class object
        // obj2.a_protected_f1();         // protected function not visible
        // obj2.a_private_f1();           // private   function not visible

        obj2.priva_call_em();             // can call the public function of the
                                          // derived portion of the class
```

```
	// examine accessability of member variables of base object
	// via an instance of the base object
	// obj1.a_priv_var = 100;		// private variable is not visible
	// obj1.a_prot_var = 100;		// protected variable is not visible
	obj1.a_pub_var  = 100;		// public data is visible

	// examine accessability of member variables of derived class portion
	// of priva object via an instance of the derived object
	// obj2.priva_priv_var = 200;	// private variable is not visible
	// obj2.priva_prot_var = 200;	// protected variable is not visible
	obj2.priva_pub_var  = 200;	// public data is visible

	// examine the accessability of member variables of base portion of
	// priva object via an instance of a derived object
	// obj2.priva_priv_var = 300;	// variable is not visible
	// obj2.priva_prot_var = 300;	// variable is not visible
	obj2.priva_pub_var  = 300;
}
```

Output From Running Program

```
The a no arg constructor has been invoked
In a_print 1 2 3
In a_print 1 2 3
In a_public_f1 1 2 3
In a_private_f1 1 2 3
In a_protected_f1 1 2 3
In a_public_f1 1 2 3

The a no arg constructor has been invoked
In priva no arg constructor
priva_priv_var is: 10 priva_prot_var is: 20 priva_pub_var  is: 30
2 3

In priva_public_f1
priva_priv_var is: 10 priva_prot_var is: 20 priva_pub_var  is: 30
2 3

In priva_private_f1
priva_priv_var is: 10 priva_prot_var is: 20 priva_pub_var  is: 30
2 3

In priva_protected_f1
priva_priv_var is: 10 priva_prot_var is: 20 priva_pub_var  is: 30
2 3

In priva_public_f1
priva_priv_var is: 10 priva_prot_var is: 20 priva_pub_var  is: 30
2 3

In a_protected_f1 1 2 3
In a_public_f1 1 2 3
```

So, in a nutshell, private inheritance is more restrictive than public inheritance to the end user of the class because it hides the public portion of the base class from the user of the derived class when attempting to access the base class via an instance of the derived class.

Private Inheritance Diagram

Derived Class Object

Derived Class Data	Derived Class Functions
private	private
protected	protected
public	public

Base Class Portion of Derived Class Object

Derived Class Data	Derived Class Functions
private	private
protected	protected
public	public

USING PRIVATE INHERITANCE

derived class functions may access:

derived class portion, private, protected, public data
derived classportion, private, protected, public functions

base class portion, protected and public data
base class portion, protected and public functions

THIS IS IDENTICAL TO PUBLIC INHERITANCE
from the point of view of the member functions

main()

may access public functions and data of derived class portion
may access public functions and data of base class portion directly
may not access public functions and data of base class via instance of derived class

may not access protected or private portions of derived class
may not access protected or private portions of base class

```
// polyinh1.cpp
// this program illustrates that a deriving class can redefine a function
// that is defined in the base class, this is due to a feature of the compiler known as complete
// function name resolution.   Although a function call may look as simple as f1()
// to the compiler it looks like    class_membership::f1()
#include <iostream.h>

class b
{
	private:
		int var1;
	public:
		b();				// no arg constructor
		void set_var1(int);		// will not be overloaded
		int get_var1(void);		// will not be overloaded

		// this function is going to be redefined in the class
		// that inherits from class b, the question then will be;
		// in code that calls manipulate_var1, which function will
		// be called, the base class function or the derived class
		// function??
		void manipulate_var1(int);	// do something to var1
```

Notice that nothing special or different is done to the function manipulate_var1. Nothing different is done to a function that is going to be overloaded. No designer of a function could anticipate which of their functions are going to be overloaded. You may have a pretty good idea, but you can't be sure. When you are sure, then you will have access to another feature of inheritance called virtuality, which is discussed in later programs. But for now, let's see how to deal with functions that are overloaded by the designer of the deriving classes.

```
};

b::b() : var1(10)
{
	cout << "In b no arg constructor \n";
}

void b::set_var1(int ivar1)
{	// would do error checking here
	var1 = ivar1;
}

int b::get_var1()
{
	return(var1);
}
// this function is called when the invoking object is of type class b
void b::manipulate_var1(int ix)
{
	// could do error checking here
	cout << "In base class manipulate_var1, multiplying "
	<< var1 << " by " << ix << endl;
	var1 *= ix;
}
```

```
// the class pubb is an object of type b AND the additional fields varz and the additional functions
// pubb, set_var2, get_var2, and manipulate_var1
// // notice that manipulate_var1 has the same name and argument list as manipulate_var1 in the base
// class portion.  The question then is how can the compiler or run time system decide which version
// of manipulate_var1 is to be called.  Up until now we have said functions can have the same name as
// long as they have a different argument list.  Here we have the same name and the same argument list
// the Compiler needs to decide whether you mean      b::manipulate_var1(int)     or
//                                                    pubb::manipulate_var1(int)
// the compiler decides based on the data type of the invoking object
// if manipulate_var1() is called via            instance_of_base.manipulate_var1()
// then    b::manipulate_var1 is called
// if manipulate_var1() is called via            instance_of_derived.manipulate_var1()
// then    pubb::manipulate_var1 is called

class pubb : public b
{
        private:
                int var2;
        public:
                pubb();                         // no arg constructor
        void manipulate_var1(int);              // OVERLOADED FUNCTION
                                                // SAME FUNCTION EXISTS IN BASE CLASS
        void set_var2(int);                     // new function unique to pubb
        int get_var2(void);
};

pubb::pubb() : var2(5)
{
        cout << "In pubb no arg constructor \n";
}

void pubb::set_var2(int ivar2)
{
        var2 = ivar2;
}

int pubb::get_var2()
{
        return(var2);
}

// this is the overloaded function
// it is called when the invoking object is of type pubb
void pubb::manipulate_var1(int ix)
{
        cout << "In derived class manipulate_var1, adding"
        << ix << " to " << this->get_var1() << endl
        << "multiplying "
        << var2 << " by " << 2 << endl;

        this->set_var1( this->get_var1() + ix);
        var2 *= 2;
}
```

```
main()
{
	// declare an object of the base type
	b obj1;

	// make calls to the base class functions
	cout << "Initial value of obj1.var1 is " << obj1.get_var1() << endl;
	obj1.set_var1(53);
	cout << "New     value of obj1.var1 is " << obj1.get_var1() << endl;
	obj1.manipulate_var1(2);		// base class function called
	cout << "New     value of obj1.var1 is " << obj1.get_var1() << endl;

	// declare an object of the derived type
	pubb    obj2;

	// make calls to the derived class functions
	cout << "\nInitial value of obj2.var1 is " << obj2.get_var1() << endl;
	cout << "Initial value of obj2.var2 is " << obj2.get_var2() << endl;

	cout << "Calling obj2.set_var1(7) and obj2.set_var2(17)\n";
	obj2.set_var1(7);
	obj2.set_var2(17);
	cout << "New     value of obj2.var1 is " << obj2.get_var1() << endl;
	cout << "New     value of obj2.var2 is " << obj2.get_var2() << endl;

	// Which manipulate_var1 will be called?  Derived or Base?
	cout << "\nCalling manipulate_var1 via obj2.manipulate_var1(2) \n";
	obj2.manipulate_var1(2);		// derived class function is called
	cout << "New     value of obj2.var1 is " << obj2.get_var1() << endl;
	cout << "New     value of obj2.var2 is " << obj2.get_var2() << endl;

	// force call to the base class functions
	cout << "\nCalling manipulate_var1 via obj2.b::manipulate_var1(2) \n";
	obj2.b::manipulate_var1(3);		// base class function called because of name forcing
	cout << "New values for obj2 are " <<
	obj2.get_var1() << "  " <<
	obj2.get_var2() << endl;
}
```

Output From Running Program

In b no arg constructor
Initial value of obj1.var1 is 10
New   value of obj1.var1 is 53
In base class manipulate_var1, multiplying 53 by 2
New   value of obj1.var1 is 106
In b no arg constructor
In pubb no arg constructor

Initial value of obj2.var1 is 10
Initial value of obj2.var2 is 5
Calling obj2.set_var1(7) and obj2.set_var2(17)
New   value of obj2.var1 is 7
New   value of obj2.var2 is 17

Calling manipulate_var1 via obj2.manipulate_var1(2)
In derived class manipulate_var1, adding2 to 7
multiplying 17 by 2
New   value of obj2.var1 is 9
New   value of obj2.var2 is 34

Calling manipulate_var1 via obj2.b::manipulate_var1(2)
In base class manipulate_var1, multiplying 9 by 3
New values for obj2 are 27 34

Polymorphism of Functions In Inheritance Relationship Diagram

main()

derived class instance d
base class instance b
function f1 occurs in both d and b

b.f1() call base class function

d.f1() call derived class function

d.b::f1() call base class function

b.d::f1() call derived class function

function call resolver

decide which function to call by name
if multiple functions with same name, use argument list to distinguish
if name and argument list are the same, look for inheritance relationship
if inheritance relationsihp exists:
decide which function to call based on data type of invoking object
otherwise generate "ambiguous function call" compiler error

base::f1()

can be called directly

can not be called indirectly via an
instance of a derived class object

derived::f1()

can be called directly

Section7.doc

Inheritance leads to some very powerful possibilities in the C++ programming language. Virtual functions allow us to utilize late binding which allows us to implement polymophic data structures. A polymorphic data structure is a data structure that can hold objects of different types.

For example, how would you like to have an array that could hold cars and trucks and any other type of vehicle you ever dreamed of. Cars and trucks are different data types, however, a polymorphic data structure, through the magic of late binding, can make a truly powerful data structure.

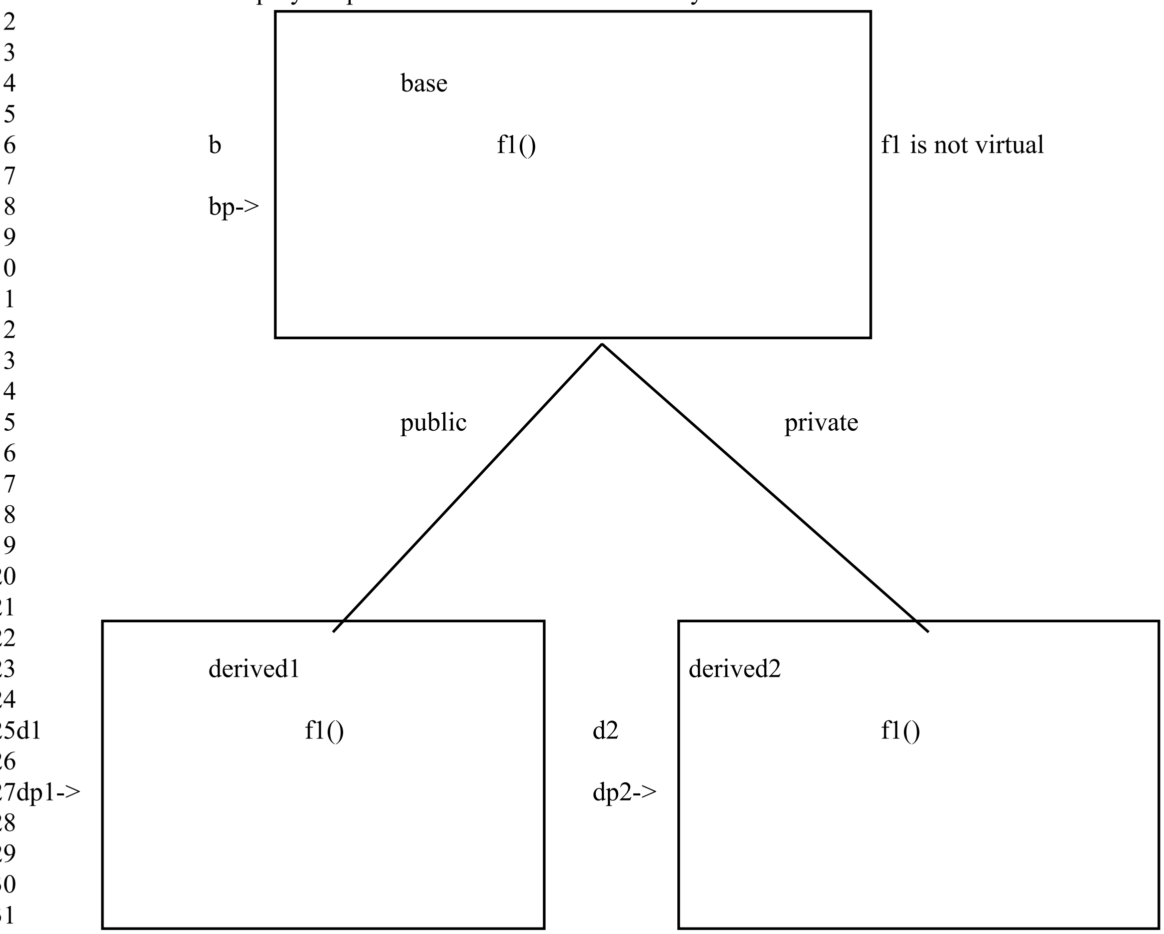
I call late binding and virtual functions and polymorphic data structures the general purpose car wash.
When you go the car wash, is there a car wash for small cars, a car wash for mid sized cars, a car wash for small trucks, a car wash for full sized cars, a car wash for jeeps, and so on???
NO!
There is a car wash for four wheeled vehicles.
How can the car wash designers get away with this?
It's because of that orange rubber thing that grabs your left front tire.
The car wash knows that all VEHICLES are going to have a left front tire that is between this width and that width and as long as they recognize that fact who cares what the rest of the car looks like.

We are used to polymorphism in our every day lives, it's time we started getting polymorphism into our programming languages.

Late Binding
Virtual Functions
Deep Hierarchies
Multiple Chains of Virtuality.

p2bp2d.cpp	polymorphism, pointers and late binding
p2bp2d1.cpp	polymorphism, pointers and late binding
p2bp2d2.cpp	virtual functions
p2bp2d3.cpp	3 deep chain of virtuality and late binding
name.h name.cpp	class definition for a human name
drv_name.cpp	program to exercise name class
address.h address.cpp	class definition for a human address
drv_addr.cpp	program to exercise address class
polyarray.cpp	program that demonstrates polymorphic arrays and virtual functions in action.

Examine the issues of polymorphism in an inheritance hierarchy. f1 is in base and derived classes



Code	Function That Is Called
b.f1();	base f1
bp = &b;	
bp->f1();	base f1
d1.f1();	derived f1
dp1 = &d1;	
dp1->f1()	derived f1
d2.f1();	derived f1
dp2 = &d2;	
dp2->f1();	derived f1
bp = &d1;	
bp->f1();	base class f1
bp = &d2;	
bp->f1();	base class f1

```
// pointer_to_base__pointer_to_derived.C
// p2bp2d.cpp
#if 0

When you have a base class and 1 or more classes derived from that base
class, in either a flat hierarchy or a deep hierarchy, you can implement very
powerful and extremely useful data structures built on two features of C++:
        Variables of type pointer to base class may hold the address of
                either a base class object or any object of a class derived
                from that base class
        If there is a polymorphic function shared between the base class and
                the derived class, then which function is called can be
                determined either at compile time OR at run time.
        When it is determined is based on whether the polymorphic
                function has been declared VIRTUAL in the base class or not
                AND based on whether the polymorphic function is called
                using an object of a certain type or a pointer to a certain
                type.  There are many possibilities, the subsequent programs
                hopefully illustrate all the confusing yet infinitely valuable possibilities.

In this program there is a base class and two derived classes.
One class derives publicly, the other derives privately.
There is a function f1 in the base class
There is a function f1 in each of the derived classes

If we call f1 using an instance of an object then we know which f1
will be called. The function call will be determined at compile time by the
data type of the bound object.

But what if we aren't calling f1 using a bound object?

What if we are calling f1 using a pointer mechanism?
Will which function to call be determined by using the data type of the pointer
or the data type of the object being pointed to?
This program and the two which follow demonstrate the answer.

```

```
Which function is called is determined first by the data type of the bound
object.  If the type of the bound object is known and is not a pointer, then
static binding is employed.

If the bound object is a pointer there are two situations:
	1. the pointer holds the address of an object of the "proper" type
		where proper is either base or derived
		i.e. the pointer is of type pointer to base and the address
		stored is the address of a base class object or the pointer is
		of type pointer to derived and the address stored is the
		address of a derived class object

	2. the pointer holds the address of an object of a "derived" type
		i.e. the pointer is of type pointer to base but the address
		stored is the address of a derived class object

In this case, if the base class function is NOT VIRTUAL
	then the base class function is called, regardless of the data type of
	the object being pointed to because of the data type of the pointer.

	if the base class function is VIRTUAL
	then the base class or derived class function is called based on the
	data type of the object being pointed to, not the data type of the
	pointer.  This is called DYNAMIC BINDING.

This program does not use virtual functions
it shows how non virtual functions will always revert to the base class function
if a base class pointer is used because of the data type of the pointer
#endif

#include <iostream.h>
class base
{
	private:
		int x;
	public:
		void f1() { cout << "In base::f1\n";}
};

class derived1 : public base
{
	private:
		int y;
	public:
		void f1() { cout << "In derived1::f1\n"; }
};

class derived2 : private base
{
	private:
		int y;
	public:
		void f1() { cout << "In derived2::f1\n"; }
};
```

```
main()
{
	// these allocations and assignments make sense
	// we are putting addresses of objects that are of the type
	// that the pointer is supposed to point to
	base b;			// base class object
	base * bp = & b;		// pointer to base with address of base object

	// call the function f1 that is in the base class
	// the compiler knows which one to call based on the data type of the
	// invoking object b
	cout << "Calling b.f1() ";
	b.f1();
	// in this call the compiler knows which function to call based on
	// the data type of the pointer, pointer to base
	cout << "Calling bp->f1() with address of base object b  ";
	bp->f1();
	cout << endl;

	derived1 d1;
	derived1 * dp1 = & d1;	// derived, pointer to derived

	// call the function f1 that is in the derived class
	// the compiler knows which one to call based on the data type of the
	// invoking object
	cout << "Calling d1.f1() ";
	d1.f1();
	// in this call the compiler knows which function to call based on
	// the data type of the pointer, pointer to derived
	cout << "Calling dp1->f1() with address of derived object d1  ";
	dp1->f1();
	cout << endl;


	derived2 d2;
	derived2 * dp2 = & d2;	// derived, pointer to derived

	// call the function f1 that is in the base class
	// the compiler knows which one to call based on the data type of the
	// invoking object
	cout << "Calling d2.f1() ";
	d2.f1();
	// in this call the compiler knows which function to call based on
	// the data type of the pointer, pointer to derived
	cout << "Calling dp2->f1() with address of derived object d2  ";
	dp2->f1();
	cout << endl;
```

```
        // assign a derived address to a pointer to base class object
        bp = &d1;
        cout << "Calling bp->f1() with address of derived object d1  ";
        bp->f1();           // which function will get called, base or derived?
        // the base class function is called because of the data type of the
        // pointer, IF THE FUNCTION F1 WERE VIRTUAL, the derived function
        // would have been called

        // bp = &d2;        // cannot assign address of privately derived object
                            // to a pointer to base because of the private
                            // derivation
                            // In a later program we will attempt to resolve
                            // this problem
        // bp->f1();        // which function will get called, base or derived?

        // now try to make an arary of pointers to base and populate it with
        // addresses of both base and derived objects
        // this works.  It is called a POLYMORPHIC data structure
        // the next program illustrates an outstanding usage of this structure
        base * barray[4];

        barray[0] = bp;
        barray[1] = dp1;

        // this line won't work because of the private derivation
        // barray[2] = dp2;

        cout << "Calling barray[0]->f1() ";
        barray[0]->f1();
        cout << endl;
        cout << "Calling barray[1]->f1() ";
        barray[1]->f1();
        cout << endl << endl;
}
```

Output From Running Program

Calling b.f1() In base::f1
Calling bp->f1() with address of base object b In base::f1

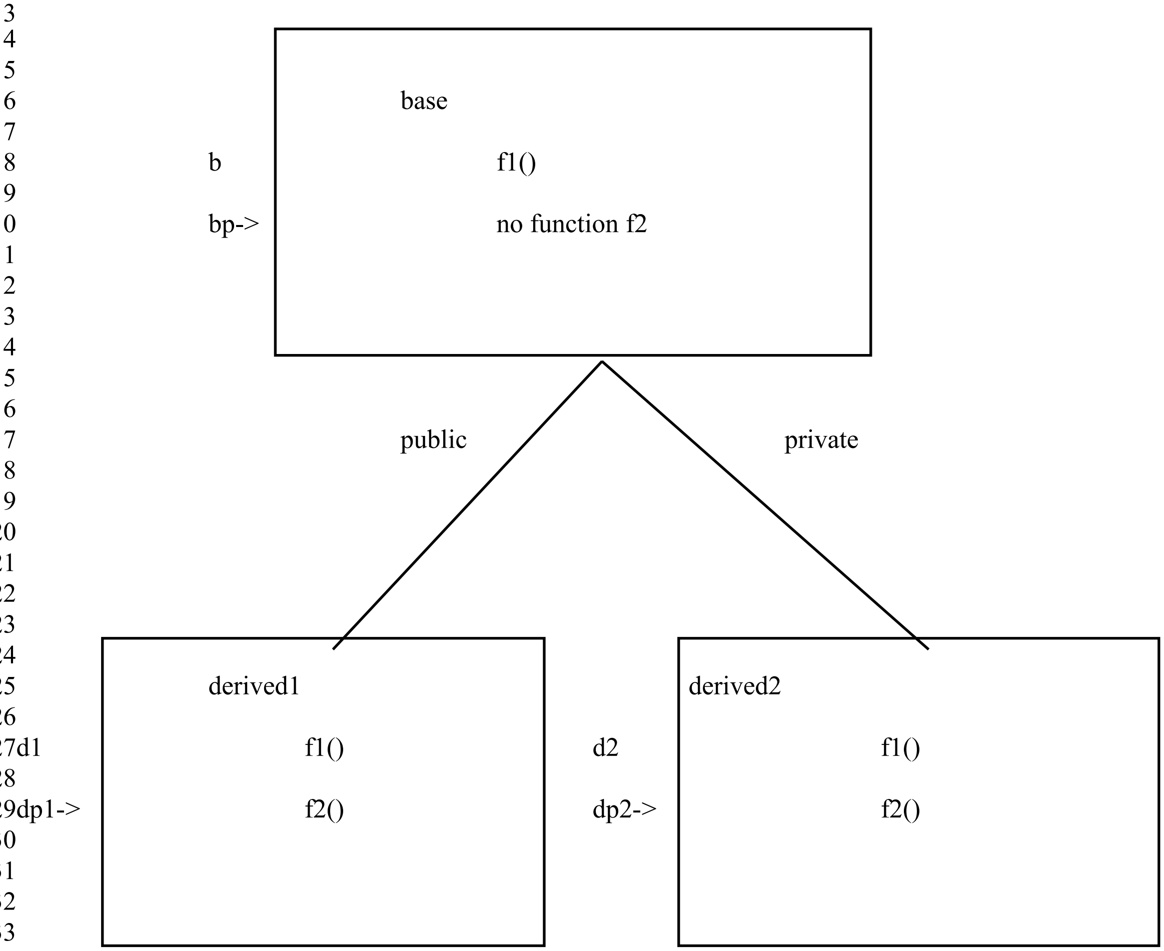
Calling d1.f1() In derived1::f1
Calling dp1->f1() with address of derived object d1 In derived1::f1

Calling d2.f1() In derived2::f1
Calling dp2->f1() with address of derived object d2 In derived2::f1

Calling bp->f1() with address of derived object d1 In base::f1
Calling barray[0]->f1() In base::f1

Calling barray[1]->f1() In base::f1

Examine "bottom of hierarchy only" functions. f2 is only in the deriving classes, not the base class



Code	Function That Is Called	Code	Function That Is Called
b.f1();	base f1	bp = &d1;	
bp = &b;		bp->f1();	base class f1
bp->f1();	base f1	bp->f2();	derived class f2
d1.f1();	derived f1	bp = &d2;	
d1.f2();	derived f2	bp->f1();	base class f1
dp1 = &d1;		bp->f2();	derived class f2
dp1->f1();	derived f1		
dp1->f2();	derived f2		
d2.f1();	derived f1		
d2.f2();	derived f2		
dp2 = &d2;			
dp2->f1();	derived f1		
dp2->f2();	derived f2		

```
// pointer_to_base__pointer_to_derived1.C
// p2bp2d1.cpp


// this program extends pointer_to_base__pointer_to_derived.C by adding a
// function f2 to the derived classes.  There is no function f2 in the base
// class.  The question then  becomes, can f2 be called using the pointer
// methods that were employed to call f1 in the previous program?

// The answer is yes.
// Since f2 is not polymorphic on a base class function, the call will resolve
// to the derived class object
// Furthermore, calls to f2 cannot be made using a pointer to base class object
// even if it holds the address of a derived class object because static
// binding is used because the functions are still not declared VIRTUAL

#include <iostream.h>

class base
{
	private:
		int x;
	public:
		void f1() { cout << "In base::f1\n";}
};

class derived1 : public base
{
	private:
		int y;
	public:
		void f1() { cout << "In derived1::f1\n"; }
		void f2() { cout << "In derived1::f2\n"; }
};

class derived2 : private base
{
	private:
		int y;
	public:
		void f1() { cout << "In derived2::f1\n"; }
		void f2() { cout << "In derived2::f2\n"; }
};
```

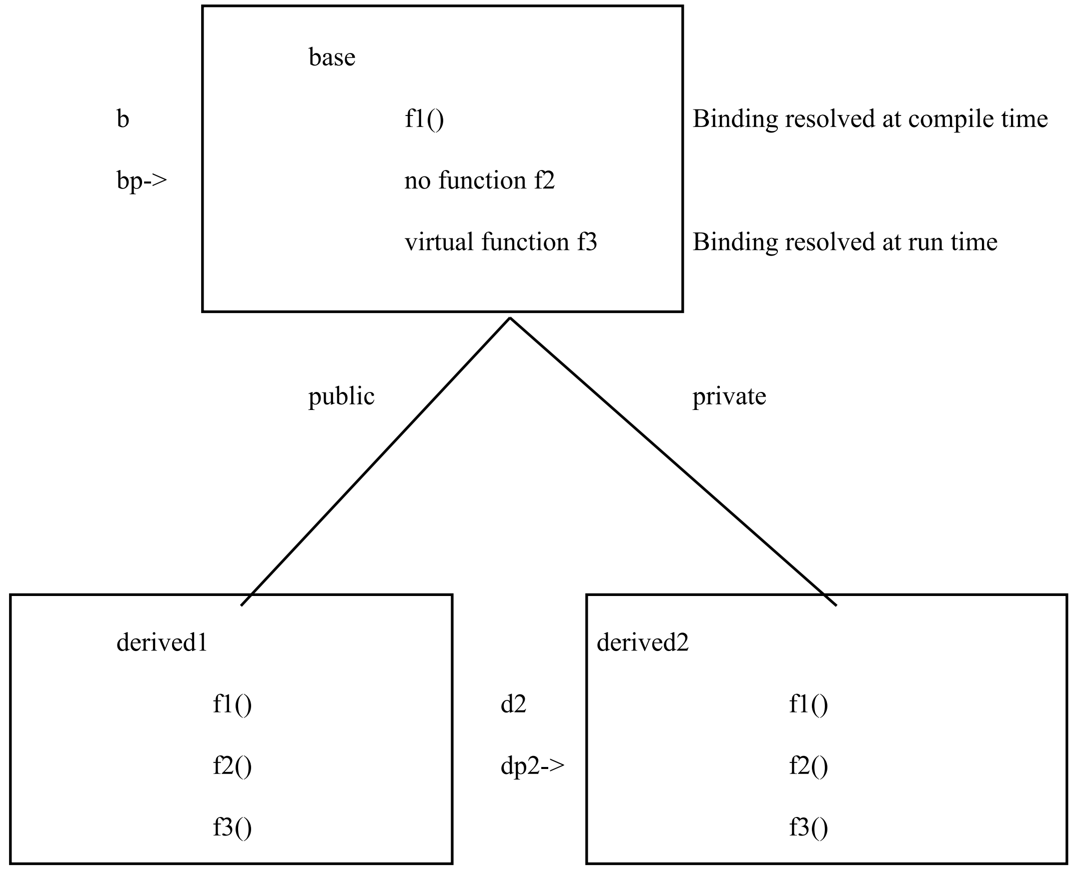
```
main()
{
        base b;                         // base class object
        base * bp = & b;                // pointer to base with address of base object

        cout << "Calling b.f1() ";
        b.f1();
        // b.f2();                      // there is no f2 function in the base class
        cout << "Calling bp->f1() with address of base object b  ";
        bp->f1();
        // bp->f2();                    // there is no f2 function in base class
        cout << endl;


        derived1 d1;
        derived1 * dp1 = & d1;          // derived, pointer to derived

        cout << "Calling d1.f1() ";
        d1.f1();                        // the derived f1 is called
        cout << "Calling d1.f2() ";
        d1.f2();                        // the derived f2 is called
        cout << "Calling dp1->f1() with address of derived object d1  ";
        dp1->f1();                      // the derived f1 is called because dp1 is of type
                                        // pointer to derived
        cout << "Calling dp1->f2() with address of derived object d1  ";
        dp1->f2();                      // the derived f2 is called because dp1 is of type
                                        // pointer to derived
        cout << endl;


        derived2 d2;
        derived2 * dp2 = & d2;          // derived, pointer to derived

        // call the function f1 that is in the base class
        // the compiler knows which one to call based on the data type of the
        // invoking object or the data type of the pointer
        cout << "Calling d2.f1() ";
        d2.f1();                        // derived f1 called because d2 is of type derived
        cout << "Calling d2.f2() ";
        d2.f2();                        // derived f2 is called because d2 is of type derived
        cout << "Calling dp2->f1() with address of derived object d2  ";
        dp2->f1();                      // derived f1 called, dp2 of type pointer to derived
        cout << "Calling dp2->f2() with address of derived object d2  ";
        dp2->f2();                      // derived f2 called, dp2 of type pointer to derived
        cout << endl;
```

```
	// assign the address of a derived object to a variable of type pointer
	// to base
	bp = &d1;
	cout << "Calling bp->f1() with address of derived object d1  ";
	bp->f1();			// which function will get called, base or derived?
					// the base function is called because f1 is not
					// VIRTUAL and bp is of type pointer to base

	// bp->f2();			// there is no base class function f2 and bp is of
					// type pointer to base

	// bp = &d2;			// can't do this assignment because private derivation
	// bp->f1();			// can't do this call
	// bp->f2();			// can't do this call

	// now try to make an arary of pointers to base and populate it with
	// addresses of both base and derived objects
	// this works.  It is called a POLYMORPHIC data structure
	// the next program illustrates an outstanding usage of this structure
	base * barray[4];

	barray[0] = bp;
	barray[1] = dp1;

	// this line won't work because of the private derivation
	// barray[2] = dp2;

	cout << "Calling barray[0]->f1() ";
	barray[0]->f1();
	cout << endl;
	cout << "Calling barray[1]->f1() ";		// will the base or derived be called?
	barray[1]->f1();

	// cout << "Calling barray[1]->f2() ";	// will the derived be accessible?
	// barray[1]->f2();				// NO
	cout << endl << endl;
}
```

Output From Running Program

```
Calling b.f1() In base::f1
Calling bp->f1() with address of base object b  In base::f1

Calling d1.f1() In derived1::f1
Calling d1.f2() In derived1::f2
Calling dp1->f1() with address of derived object d1  In derived1::f1
Calling dp1->f2() with address of derived object d1  In derived1::f2

Calling d2.f1() In derived2::f1
Calling d2.f2() In derived2::f2
Calling dp2->f1() with address of derived object d2  In derived2::f1
Calling dp2->f2() with address of derived object d2  In derived2::f2

Calling bp->f1() with address of derived object d1  In base::f1
Calling barray[0]->f1() In base::f1

Calling barray[1]->f1() In base::f1
```

Examine Virtual Functions. f3 is a virtual function, it is in the base class and the deriving classes.



Code	Function That Is Called	Code	Function That Is Called
b.f1();	base f1	bp = &d1;	
b.f2() CANNOT MAKE THIS CALL		bp->f1();	base class f1
b.f3();	base f3	bp->f2();	derived class f2
bp = &b;		bp->f3();	derived class f3
bp->f1();	base f1		
bp->f2(); CANNOT MAKE THIS CALL			
bp->f3()	base f3		
d1.f1();	derived f1	d2.f1();	derived f1
d1.f2();	derived f2	d2.f2();	derived f2
d1.f3();	derived f3	d2.f3();	derived f3
dp1 = &d1;		dp2 = &d2;	
dp1->f1();	derived f1	dp2->f1();	derived f1
dp1->f2();	derived f2	dp2->f2();	derived f2
dp1->f3();	derived f3	dp2->f3();	derived f3
d2.f1();	derived f1	bp = &d2;	
d2.f2();	derived f2	bp->f1();	base class f1
d2.f3();	derived f3	bp->f2();	derived class f2
dp2 = &d2;		bp->f3();	derived class f3
dp2->f1();	derived f1		
dp2->f2();	derived f2		
dp2->f3();	derived f3		

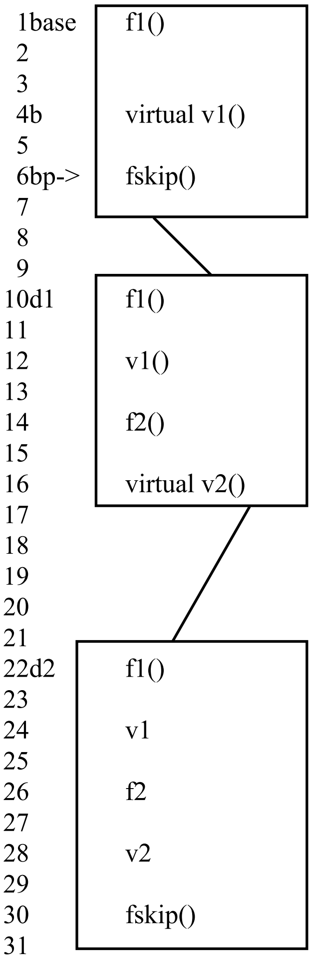
```
// pointer_to_base__pointer_to_derived2.C
// p2bp2d2.cpp

// this program extends pointer_to_base__pointer_to_derived1.C by adding a
// function f3 to the derived classes.  There is a function f3 in the base
//class and it has been declared to be VIRTUAL.  The question then
// becomes, can f3 be called using the pointer methods that were employed to
// call f1 in the previous program and if so, which function f3 will be called,
// the base or derived function class function?

// VIRTUAL functions cause DYNAMIC binding to used to determine which function
// to call.  That means, that the function call will not be resolved at compile
// time based on the data type of the invoking object or the data type of
// the pointer used to do the invocation but rather at run time
// based on the data type of the invoking object or the data type of the
// address stored in a pointer REGARDLESS OF THE DATA TYPE OF THE POINTER

#include <iostream.h>

class base
{
	private:
		int x;
	public:
		void f1() { cout << "In base::f1\n";}
		virtual void f3() { cout << "In base::f3\n"; }
};

class derived1 : public base
{
	private:
		int y;
	public:
		void f1() { cout << "In derived1::f1\n"; }
		void f2() { cout << "In derived1::f2\n"; }
		void f3() { cout << "In derived1::f3\n"; }
};

class derived2 : private base
{
	private:
		int y;
	public:
		void f1() { cout << "In derived2::f1\n"; }
		void f2() { cout << "In derived2::f2\n"; }
		void f3() { cout << "In derived2::f3\n"; }
};
```

```
main()
{
	base b;
	base * bp = & b;		// base, pointer to base

	cout << "Calling b.f1() ";
	b.f1();
	// b.f2();			// there is no f2 in base class objects
	cout << "Calling b.f3() ";
	b.f3();
	cout << "Calling bp->f1() with address of base object b  ";
	bp->f1();
	// bp->f2();			// there is no f2 in base class objects
	cout << "Calling bp->f3() with address of base object b  ";
	bp->f3();
	cout << endl;

	derived1 d1;
	derived1 * dp1 = & d1;	// derived, pointer to derived

	cout << "Calling d1.f1() ";
	d1.f1();			// derived class object called, d1 determines
	cout << "Calling d1.f2() ";
	d1.f2();			// derived class object called, d1 determines
	cout << "Calling d1.f3() ";
	d1.f3();			// derived class object called, d1 determines
	cout << "Calling dp1->f1() with address of derived object d1  ";
	dp1->f1();			// derived class object called, dp1 determines
	cout << "Calling dp1->f2() with address of derived object d1  ";
	dp1->f2();			// derived class object called, dp1 determines
	cout << "Calling dp1->f3() with address of derived object d1  ";
	dp1->f3();			// derived class object called, dp1 determines
	cout << endl;


	derived2 d2;
	derived2 * dp2 = & d2;	// derived, pointer to derived

	// call the function f1 that is in the base class
	// the compiler knows which one to call based on the data type of the
	// invoking object or the data type of the pointer
	cout << "Calling d2.f1() ";
	d2.f1();			// derived class object called, d2 determines
	cout << "Calling d2.f2() ";
	d2.f2();			// derived class object called, d2 determines
	cout << "Calling d2.f3() ";
	d2.f3();			// derived class object called, d2 determines
	cout << "Calling dp2->f1() with address of derived object d2  ";
	dp2->f1();			// derived class object called, dp2 determines
	cout << "Calling dp2->f2() with address of derived object d2  ";
	dp2->f2();			// derived class object called, dp2 determines
	cout << "Calling dp2->f3() with address of derived object d2  ";
	dp2->f3();			// derived class object called, dp2 determines
	cout << endl;
```

```
        // assign the address of a derived object to a variable of type pointer
        // to base
        bp = &d1;
        cout << "Calling bp->f1() with address of derived object d1  ";
        bp->f1();                   // base called, bp determines
        cout << "Calling bp->f3() ";
        bp->f3();                   // DERIVED called, address stored in bp determines
                                    // because f3 is VIRTUAL

        base * barray[4];
        barray[0] = bp;
        barray[1] = dp1;

        cout << "Calling barray[0]->f1() ";
        barray[0]->f1();
        cout << endl;

        cout << "Calling barray[1]->f1();          // base called.  why?
        barray[1]->f1();
        // cout << "Calling barray[1]->f2() ";     // can't make this call
        //barray[1]->f2();                         // can't make this call
        cout << "Calling barray[1]->f3() ";        // DERIVED CALLED.  WHY?
        barray[1]->f3();
        cout << endl << endl;
}
```

Output From Running Program

```
Calling b.f1() In base::f1
Calling b.f3() In base::f3
Calling bp->f1() with address of base object b  In base::f1
Calling bp->f3() with address of base object b  In base::f3

Calling d1.f1() In derived1::f1
Calling d1.f2() In derived1::f2
Calling d1.f3() In derived1::f3
Calling dp1->f1() with address of derived object d1  In derived1::f1
Calling dp1->f2() with address of derived object d1  In derived1::f2
Calling dp1->f3() with address of derived object d1  In derived1::f3

Calling d2.f1() In derived2::f1
Calling d2.f2() In derived2::f2
Calling d2.f3() In derived2::f3
Calling dp2->f1() with address of derived object d2  In derived2::f1
Calling dp2->f2() with address of derived object d2  In derived2::f2
Calling dp2->f3() with address of derived object d2  In derived2::f3

Calling bp->f1() with address of derived object d1  In base::f1
Calling bp->f3() In derived1::f3
Calling barray[0]->f1() In base::f1

Calling barray[1]->f1() In base::f1
Calling barray[1]->f3() In derived1::f3
```



b.f1()	base class f1	bp = &b	
b.v1()	base class v1	bp->f1(	base f
b.fskip()	base class fskip	bp->v1()base v1	
b.fskip();	base class fskip	bp->fskip()	base fskip

derived1

d1.f1()	derived class f1	dp1 = &d1	
d1.v1()	derived class v1	dp1->f1()	derived f1
d1.f2()	derived class f2	dp1->v1()	derived v1
d1.v2()	derived class v2	dp1->f2()	derived f2
d1.fskip()	base class fskip	dp1->v2()	derived v2
		dp1->fskip()	base fskip

derived2

d2.f1()	derived class f1	dp2 = &d2;	
d2.v1()	derived class v1	dp2->f1()	derived f1
d2.f2()	derived class f2	dp2->v1()	derived v1
d2.v2()	derived class v2	dp2->f2()	derived f2
d2.fskip()	derived class fskip	dp2->v2()	derived v2
		dp2->fskip()	derived fskip

NOW THESE ARE THE INTERESTING CALLS

bp = &d1; base classs pointer with address of derived object
bp->f1() base class f1
bp->v1()derived class v1
bp->fskip() base class fskip
bp->f2() CANNOT MAKE THIS CALL
bp->v2()CANNOT MAKE THIS CALL

bp = &d2; base class pointer with address of object that inherited from object that inherited from the base class
bp->f1(); base class f1
bp->v2()derived 2 v2
bp->fskip() base class fskip
bp->f2() CANNOT MAKE THIS CALL
bp->v2()CANNOT MAKE THIS CALL

dp1 = &d2; pointer to class derived from base with address of object derived from derived class
dp1->f1() derived 1 f1
dp1->v1() derived 2 v1
dp1->fskip() base class fskip
dp1->f2() derived 1 f2
dp1->v2() derived 2 v2

```
// pointer_to_base__pointer_to_derived3.C
// p2bp2d3.cpp

// this program illustrates virtual functions in a deep hierarchy
#include <iostream.h>

class base
{
	private:
		int x;
	public:
		void f1() { cout << "In base::f1\n";}
		virtual void v1() { cout << "In base::v1\n"; }

		void fskip() { cout << "In base::fskip\n"; }
};

class derived1 : public base
{
	private:
		int y;
	public:
		void f1() { cout << "In derived1::f1\n"; }
		void v1() { cout << "In derived1::v1\n"; }

		// there is no f2 in base class
		// there is no v2 in base class

		// these two functions form a "terminus of function chains"
		// in the case of v2, a new "terminus of virtuality"
		void f2() { cout << "In derived1::f2\n"; }
		virtual void v2() { cout << "In derived1::v2\n"; }

		// notice that this class does not redefine fskip
};

// this class derives from derived1, not directly from base
class derived2 : public derived1
{
	private:
		int y;
	public:
		void f1() { cout << "In derived2::f1\n"; }
		void v1() { cout << "In derived2::v1\n"; }

		void f2() { cout << "In derived2::f2\n"; }
		void v2() { cout << "In derived2::v2\n"; }

		void fskip() { cout << "In derived2::fskip\n"; }
};
```

```
main()
{
	base b;
	base * bp = & b;		// base, pointer to base

	// these calls illustrate the straight forward calls using a known object
	cout << "\nCalling b.f1() ";
	b.f1();
	cout << "Calling b.v1() ";
	b.v1();
	cout << "Calling b.fskip() ";
	b.fskip();

	// and a pointer of type pointer to base holding the address
	// of an object of type pointer to base
	cout << "\nHave assigned address of b to bp \n";
	cout << "Calling bp->f1() ";
	bp->f1();
	cout << "Calling bp->v1() ";
	bp->v1();
	cout << "Calling bp->fskip() ";
	bp->fskip();

	// these calls also illustrate straight forward calls using a known object of type derived 1
	// and a pointer of type pointer to derived1 holding the address of a derived 1 class object
	derived1 d1;
	derived1 * dp1 = & d1;

	cout << "\n\nCalling d1.f1() ";
	d1.f1();			// derived f1 called, d1 determines
	cout << "Calling d1.v1() ";
	d1.v1();			// derived v1 called, d1 determines
	cout << "Calling d1.fskip() ";
	d1.fskip();			// base called, no fskip in derived1 class
					// therefore d1 determines through inheritance
					// relationship with base
	cout << "Calling d1.f2() ";
	d1.f2();			// derived f2 called, d1 determines
	cout << "Calling d1.v2() ";
	d1.v2();			// derived v2 called, d1 determines

	cout << "\nHave assigned &d1 to dp1 \n";
	cout << "Calling dp1->f1() ";
	dp1->f1();			// derived f1 called, dp1 determines
	cout << "Calling dp1->v1() ";
	dp1->v1();			// derived v1 called, dp1 determines
	cout << "Calling dp1->fskip() ";
	dp1->fskip();			// base called, no fskip in derived1 class
					// therefore dp1 determines through inheritance
					// relationship with base
	cout << "Calling dp1->f2() ";
	dp1->f2();			// derived f2 called, dp1 determines
	cout << "Calling dp1->v2() ";
	dp1->v2();			// derived v2 called, dp1 determines
```

```
// these calls also illustrate straight forward calls using a known object of type derived 2
// and a pointer of type pointer to derived2 holding the address of a derived 2 class object
derived2 d2;
derived2 * dp2 = & d2;
cout << "\n\nCalling d2.f1() ";
d2.f1();                    // derived f1 called, d2 determines
cout << "Calling d2.v1() ";
d2.v1();                    // derived v1 called, d2 determines
cout << "Calling d2.fskip() ";
d2.fskip();                 // derived called, d2 determines
cout << "Calling d2.f2()";
d2.f2();                    // derived f2 called, d2 determines
cout << "Calling d2.v2() ";
d2.v2();                    // derived v2 called, d2 determines

cout << "\nHave assigned &d2 to dp2\n";
cout << "Calling dp2->f1() ";
dp2->f1();                  // derived f1 called, dp2 determines
cout << "Calling dp2->v1() ";
dp2->v1();                  // derived f1 called, dp2 determines
cout << "Calling dp2->fskip() ";
dp2->fskip();               // derived called
cout << "Calling dp2->f2() ";
dp2->f2();                  // derived f2 called, dp2 determines
cout << "Calling dp2->v2() ";
dp2->v2();                  // derived v2 called, dp2 determines

// now assign the address of a derived1 object to bp
cout << "\n\nHave assigned &d1 to bp " << endl;
bp = & d1;
cout << "Calling bp->f1() ";
bp->f1();                   // base called, bp determines
cout << "Calling bp->v1() ";
bp->v1();                   // derived called, &d1 determines
cout << "Calling bp->fskip() ";
bp->fskip();                // base called, no fskip in derived1 class
//cout << "Calling bp->f2() ";
// bp->f2();                CANNOT MAKE THIS CALL
//cout << "Calling bp->v2() ";
// bp->v2();                CANNOT MAKE THIS CALL
cout << endl << endl;
// now assign the address of a derived2 object to bp
cout << "\n\nHAVE ASSIGNED &d2 to bp " << endl;
bp = & d2;
cout << "Calling bp->f1() ";
bp->f1();                   // base called, bp determines
cout << "Calling bp->v1() ";
bp->v1();                   // derived called, &d1 determines
cout << "Calling bp->fskip() ";
bp->fskip();                // base called, bp determines
//cout << "Calling bp->f2() ";
//bp->f2();                 CANNOT MAKE THIS CALL
//cout << "Calling bp->v2() ";
//bp->v2();                 CANNOT MAKE THIS CALL
```

```
        // now assign the address of a derived2 object to dp1
        dp1 = & d2;
        cout <<  "\n\nHave Assigned &d2 to dp1 " << endl;
        cout << "Calling dp1->f1() ";
        dp1->f1();              //          derived1::f1 is called
        cout << "Calling dp1->v1() ";
        dp1->v1();              //          derived2::v1 is called
        cout << "Calling dp1->fskip() ";
        dp1->fskip();           //          base::fskip is called
        cout << "Calling dp1->f2() ";
        dp1->f2();              //          derived1::f2 is called
        cout << "Calling dp1->v2() ";
        dp1->v2();              //          derived2::v2 is called
}
```

```
Output From Running Program

Calling b.f1() In base::f1
Calling b.v1() In base::v1
Calling b.fskip() In base::fskip

Have assigned address of b to bp
Calling bp->f1() In base::f1
Calling bp->v1() In base::v1
Calling bp->fskip() In base::fskip

Calling d1.f1() In derived1::f1
Calling d1.v1() In derived1::v1
Calling d1.fskip() In base::fskip
Calling d1.f2() In derived1::f2
Calling d1.v2() In derived1::v2

Have assigned &d1 to dp1
Calling dp1->f1() In derived1::f1
Calling dp1->v1() In derived1::v1
Calling dp1->fskip() In base::fskip
Calling dp1->f2() In derived1::f2
Calling dp1->v2() In derived1::v2

Calling d2.f1() In derived2::f1
Calling d2.v1() In derived2::v1
Calling d2.fskip() In derived2::fskip
Calling d2.f2()In derived2::f2
Calling d2.v2() In derived2::v2

Have assigned &d2 to dp2
Calling dp2->f1() In derived2::f1
Calling dp2->v1() In derived2::v1
Calling dp2->fskip() In derived2::fskip
Calling dp2->f2() In derived2::f2
Calling dp2->v2() In derived2::v2

Have assigned &d1 to bp
Calling bp->f1() In base::f1
Calling bp->v1() In derived1::v1
Calling bp->fskip() In base::fskip

HAVE ASSIGNED &d2 to bp
Calling bp->f1() In base::f1
Calling bp->v1() In derived2::v1
Calling bp->fskip() In base::fskip

Have Assigned &d2 to dp1
Calling dp1->f1() In derived1::f1
Calling dp1->v1() In derived2::v1
Calling dp1->fskip() In base::fskip
Calling dp1->f2() In derived1::f2
Calling dp1->v2() In derived2::v2
```

```
// name.h
// introduce the name class
// This is a class that I came up with to model a human name
// It is far from complete but it gives you the idea
// This class and an associated class, address, are going to be
// EMBEDDED in another class

// our class descriptions will be in files labelled            .h
// the implementations of a class will be in files labelled    .cpp
// programs to test a class will be in files labelled          ___.cpp

// this mechanism prevents this .h file from being included more than once
// this is important to do if you are going to embed classes or inherit
#ifndef NAME_PROTECTOR
#define NAME_PROTECTOR

#include <iostream.h>
#include <string.h>

#define         MAX_NAME_LEN        50

class name
{
        private:
                char n[MAX_NAME_LEN];           // the characters to store
                int  len;                       // length of stored name
                int initialized;                // has the name been set yet?
        public:
                void print_name();              // function to print a name
                void set_name(char *);          // function to change a name
                void get_name_from_user();

                name(char *);                   // one argument constructor
                name();                         // no argument constructor

                int is_it_initialized();        // returns 0 if name has not been set
                int what_is_len();              // return value of len field
                char * what_is_name();          // return the value of n field
};
#endif
```

```
// name.cpp
// implement the name class described in name.h
#include <iostream.h>
#include <stdlib.h>
#include "name.h"

void name::print_name()
{
	cout << n << endl;		// print out the name field
}

// receive a name, error check on length, prompt if necessary
void name::set_name(char * s)
{
	len = strlen(s);			// get length of new name
	while ( len > MAX_NAME_LEN )
	{
		cout << "Name is too long, try again" << endl;
		cin >> s;
		len = strlen(s);
	}
	strcpy(&n[0],s);			// copy new name into new area of memory
	initialized = 1;			// set the initialized field
	return;
}
```

```
void name::get_name_from_user()
{
	char temp [MAX_NAME_LEN * 2];
	char * tptr = &temp[0];
	char inchar;

	cout << "What is your name?\n";

	// account for any carriage return that may have been left over
	cin.get(inchar);
	if ( inchar == '\n' )
	{
		// go ahead and get next char, looking for non cr
		cin.get(inchar);
	}

	// continue to read characters until next new line is encountered
	while ( inchar != '\n' )
	{
		*tptr++ = inchar;
		cin.get(inchar);
	}
	*tptr = NULL;		// replace the newline with a null
	len = strlen(temp);	// get length of new name
	while ( len > MAX_NAME_LEN )
	{
		cout << "Name is too long, try again" << endl;
		cout << "What is your name?\n";

		tptr = &temp[0];
		cin.get(inchar);

		while ( inchar != '\n' )
		{
			*tptr++ = inchar;
			cin.get(inchar);
		}

		*tptr = '\0';
		len = strlen(temp);
	}
	strcpy(&n[0],temp);	// copy new name into new area of memory
	initialized = 1;	// set the initialized field
	return;
}
```

```
name::name(char * s)
{
#ifdef TRACE
        cout << "name char * constructor called \n";
#endif
        len = strlen(s);                // get length of new name
        while ( len > MAX_NAME_LEN )
        {
                cout << "Name is too long, try again" << endl;
                cin >> s;
                len = strlen(s);
        }

        strcpy(&n[0],s);                // copy new name into new area of memory
        initialized = 1;                // set the initialized field
        return;
}

// this is the no argument constructor
name::name()
{
#ifdef TRACE
        cout << "name no arg constructor called \n";
#endif

        len = strlen("Uninitialized Name");             // get length of new name
        strcpy(&n[0],"Uninitialized Name");             // copy name into memory
        initialized = 0;                                // set the initialized field to not set
        return;
}

// these are convenience functions which provide access to private data
int name::what_is_len()
{
        return(len);
}

char * name::what_is_name()
{
        return(n);
}

int name::is_it_initialized()
{
        return(initialized);
}
```

```
// drv_name.cpp
// to compile this program  use
// tcc drv_name.cpp name.cpp

// this program will exercise the name class
// it tries to call every function that is implemented in the name class
// A program like this is invaluable in the documentation of your class

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

main()
{
        // Cause the no argument constructor to be invoked
        name n1;
        cout << "Calling print_name for uninitialized n1 \n";
        n1.print_name();
        cout << "return value from what_is_len is " << n1.what_is_len() << endl;
        cout << "return value from initialized  is " << n1.is_it_initialized() << endl;

        // exercise the set_name function
        n1.set_name("John Timothy Kalnay");
        cout << "Repeating call sequence after setting name to jtk\n";
        n1.print_name();
        cout << "return value from what_is_len is " << n1.what_is_len()  << endl;
        cout << "return value from initialized  is " << n1.is_it_initialized() << endl;

        // cause the one arg constructor to be called
        name n2("Dr. Susann Marie Brady Kalnay");
        cout << "Repeating call sequence after creating n2 with name in it\n";
        n2.print_name();
        cout << "return value from what_is_len is " << n2.what_is_len() << endl;
        cout << "return value from initialized  is " << n2.is_it_initialized() << endl;

        name n3;
        n3.get_name_from_user();
        n3.print_name();
        cout << "return value from what_is_len is " << n3.what_is_len() << endl;
        cout << "return value from initialized  is " << n3.is_it_initialized()<< endl;
}
```

```
Output From Running Program

Calling print_name for uninitialized n1
Uninitialized Name
return value from what_is_len is 18
return value from initialized  is 0
Repeating call sequence after setting name to jtk
John Timothy Kalnay
return value from what_is_len is 19
return value from initialized  is 1
Repeating call sequence after creating n2 with name in it
Dr. Susann Marie Brady Kalnay
return value from what_is_len is 29
return value from initialized  is 1
```

```
// address.h
// introduce the address class
// ask yourself if this differs from the name class?  Could name and address both be instances of a
// string -like class?
#ifndef ADDRESS_PROTECTOR
#define ADDRESS_PROTECTOR

#include <iostream.h>
#include <string.h>

#define  MAX_ADDRESS_LEN    50

class address
{
        private:
                char a[MAX_ADDRESS_LEN];
                int  len;                          // length of stored address
                int initialized;                   // has an addr been stored in there?
        public:
                void print_address();              // function to print an address
                void set_address(char *);          // function to change an address
                void get_address_from_user();      // do what name says

                address(char  *);                  // one argument constructor
                address();                         // no argument constructor

                int is_it_initialized();           // returns 0 if address has been set
                int what_is_len();                 // return value of len field

                char * what_is_address();          // return the a field
};
#endif
```

```
// address.cpp
// implement the address class
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
#include "address.h"

void address::print_address()
{
	cout << a << endl;
}

void address::set_address(char * s)
{
	len = strlen(s);			// get length of new address
	if ( len > MAX_ADDRESS_LEN )
	{
		cout << "address is too long \n";
		exit(-1);		// this is drastic, could do an error checking loop  instead
	}
	strcpy(a,s);			// copy new address into new area of memory
	initialized = 1;			// set the initialized field
	return;
}

address::address(char * s)  {
#ifdef TRACE
	cout << "address char * constructor called \n";
#endif
	len = strlen(s);			// get length of new address
	if ( len > MAX_ADDRESS_LEN ) {
		cout << "address is too long \n";
		exit(-1);		// this is drastic, could error check instead
	}
	strcpy(a,s);			// copy new address into new area of memory
	initialized = 1;			// set the initialized field
	return;
}

address::address()		{
#ifdef TRACE
	cout << "address no arg constructor called \n";
#endif
	len = strlen("Uninitialized address");		// get length of new address
	if ( len > MAX_ADDRESS_LEN )
	{
		cout << "address is too long \n";
		exit(-1);
	}
	strcpy(a,"Uninitialized address");			// copy into new area of memory
	initialized = 0;					// set the initialized field to not set
	return;
}
```

```
void address::get_address_from_user()
{
	char temp[MAX_ADDRESS_LEN * 2];
	char * aptr = & temp[0];
	char inchar;

	cout << "What is your address? \n";
	cin.get(inchar);
	if ( inchar == '\n' )
	{	// then we found a leading \n, dump it
		cin.get(inchar);
	}
	while ( inchar != '\n' )
	{
		*aptr++ = inchar;
		cin.get(inchar);
	}
	*aptr = NULL;

	len = strlen(temp);		// get length of new address
	while ( len > MAX_ADDRESS_LEN )
	{
	cout << "address is too long\n";
	cout << "What is your shorter address? \n";

	aptr = &temp[0];
	cin.get(inchar);

	while ( inchar != '\n' )
	{
		*aptr++ = inchar;
		cin.get(inchar);
	}
	*aptr = NULL;

		len = strlen(temp);
	}
	strcpy(a,temp);			// copy into new area of memory
	initialized = 1;		// set the initialized field to not set
	return;
}

int address::what_is_len()
{
	return(len);
}
char * address::what_is_address()
{
	return(a);
}
int address::is_it_initialized()
{
	return(initialized);
}
```

```
// drv_addr.cpp
// program to exercise the address class
#include <iostream.h>
#include "address.h"

main()
{
	address a1;
	cout << "Calling print_address for uninitialized n1 \n";
	a1.print_address();
	cout << "return value from what_is_len is " << a1.what_is_len() << endl;
	cout << "return value from initialized  is " << a1.is_it_initialized() << endl;
	cout << "The address returned from what_is_address is " <<
		a1.what_is_address() << endl;

	a1.set_address("600 Pine Hollow Road");
	cout << "Repeating call sequence after setting address to home\n";
	a1.print_address();
	cout << "return value from what_is_len is " << a1.what_is_len() << endl;
	cout << "return value from initialized  is " << a1.is_it_initialized() << endl;
	cout << "The address returned from what_is_address is " <<
		a1.what_is_address() << endl;


	address a2("108 Bridge Street East");
	cout << "Repeating call sequence after creating a2 with addr in it\n";
		a2.print_address();
	cout << "return value from what_is_len is " << a2.what_is_len()  << endl;
	cout << "return value from initialized  is " << a2.is_it_initialized() << endl;
	cout << "The address returned from what_is_address is " <<
		a2.what_is_address() << endl;

	address a3;
	a3.get_address_from_user();
	cout << "return value from what_is_len is " << a3.what_is_len()  << endl;
	cout << "return value from initialized  is " << a3.is_it_initialized() << endl;
	cout << "The address returned from what_is_address is " <<
		a3.what_is_address() << endl;
}
```

```
Output From Running Program

Calling print_address for uninitialized n1
Uninitialized address
return value from what_is_len is 21
return value from initialized  is 0
Repeating call sequence after setting address to home
600 Pine Hollow Road

return value from what_is_len is 20
return value from initialized  is 1

Repeating call sequence after creating a2 with addr in it
108 Bridge Street East
return value from what_is_len is 22
return value from initialized  is 1
```

```
// polyarry.cpp
// this program demonstrates the usefullness of virtual functions and of base classes
// we are going to have objects of type army, navy, airforce and marines which all derive from a base
// class called common_class.  The common_class collects all the behaviour that is common between
// the classes and through the use of virtual functions makes the writing of the derived classes much
// easier.
#include <iostream.h>
#include "name.h"
#include "address.h"
class common_class
{
	private:
		name	n;		// this is embedded, it is not derived
		address	a;		// this also is embedded, it is not derived
	public:
		common_class(); // no arg constructor
		void q1();		// question common to services
		void q2();		// question common to services
		virtual void q3();	// question specific to service
		virtual void print_vals();	// will allow each class derived from this class to use their
						// own function of the same name and call this function too
};

// no argument constructor
common_class::common_class() : n("no name yet"), a("no address yet")
{	// invocation line initialization was employed, therefore there is no body to this constructor
}

void common_class::print_vals()
{
	n.print_name();		// access the public member function of the embedded class
	a.print_address();	// access the public member function of the embedded class
}
void common_class::q1()
{
	// this question is used to get the recruits name
	n.get_name_from_user();		// public member function of embedded class
}

void common_class::q2()
{
	// this question is used to get the recruits address
	a.get_address_from_user();	// public member function of embedded class
}

void common_class::q3()
{
	// this function only does something if the derived class
	// does not re-implement the function
	cout << "I know you haven't decided which service you want \n"
		<< "to go into, so let me give you some information \n"
		<< "on each!" << endl;
}
```

```
// An army class IS a common_class and the additional fields and functions listed here
// because q3 and print_vals were declared virtual in the base class, and because they are redefined
// here, the binding of which function will be called can be delayed until run time if pointers are used
// instead of instance names.  The value of this is in being able to call the function q3 for each of the
// deriving classes without having to have the instance name of the deriving class.  The array below
// demonstrates this.
class army : public common_class
{
	private:
		int max_miles_run;		// 0 to 100
	public:
		void q3();
		void print_vals();
};

void army::print_vals()
{
	cout << "ARMY RECRUIT \n";
	common_class::print_vals();		// use the base class routine
	cout << "Can run " << max_miles_run << " At a time \n\n";
}

void army::q3()
{
	// get the number of miles the recruit can run
	cout << "How many miles can you run?";
	cin >> max_miles_run;			// ERROR CHECKING COULD BE DONE HERE
}

// a navy class is a common_class and the additional fields and functions defined here
class navy : public common_class {
	private:
		char can_you_swim;		// n no, y yes
	public:
		void q3();
		void print_vals();
};

void navy::q3() {
	// question used to ask if the recruit can swim or not
	cout << "Can you swim?   n => no    y => yes \n";
	cin >> can_you_swim;
	if ( can_you_swim == '\n' )
		cin >> can_you_swim;
}
void navy::print_vals() {
	cout << "NAVY RECRUIT \n";
	common_class::print_vals();
	if ( can_you_swim == 'n' )
		cout << "Cannot Swim \n";
	else
		cout << "Can Swim \n";
	cout << endl;
}
```

```
// an air force class is a common_class and the extra fields and functions defined here
class air_force : public common_class
{
	private:
		char afraid_of_heights;	// n no, y yes
	public:
		void q3();
		void print_vals();
};

void air_force::q3()
{
	// question used to ask if the recruit is afraid of heights
	cout << "Are you afraid of heights?   n => no   y => yes \n";
	cin >> afraid_of_heights;
	if ( afraid_of_heights == '\n' )
		cin >> afraid_of_heights;
}

void air_force::print_vals()
{
	cout << "AIR FORCE RECRUIT \n";
	common_class::print_vals();
	if ( afraid_of_heights == 'n' )
		cout << "Is not afraid of heights\n";
	else
		cout << "Is afraid of heights \n";
	cout << endl;
}

class marines : public common_class
{
	private:
	public:
		void q3();
		void print_vals();
};

void marines::print_vals()
{
	cout << "MARINE RECRUIT \n";
	common_class::print_vals();
	cout << endl;
}

void marines::q3()
{
	return;	// no additional work done
}
```

```
main()
{
	// At the armed forces in-processing  center all of the freshly shaved recruits came to the head
	// of the line, The common information that all the services needed  from us, name,
	// address, social security number etc. were gathered by one sergeant.  This sergeant
	// didn't know and didn't care what service each recruit was going into.  The sergeants
	// job was the same regardless of what service we were going in.  Next, the special
	// information for each individual service was gathered at special counters,
	// i.e. one counter for the navy, one for the air force, one for the army
	// After our special information was acquired, we were all the same again for the next test
	// so the system had to have a way to treat us all as RECRUITS again, not as army, or navy
	// recruits.  This is very powerful in data processing, being able to process a data structure
	// at one time from one point of view, taking into account how it is SIMILAR to all other things
	// and then at the next moment being able to process that same data structure from another
	// point of view that takes into account how it is DIFFERENT from other things.
	// Pointers to base class objects holding addresses of base class objects and addresses
	// of derived class objects are extremely useful in modelling this situation.

	common_class * array[100];		// assume there's less than 100 people
						// waiting to join the armed forces today

	// we are going to build an array that
	// represents a line of recruits waiting at the induction center
	// for the armed forces
	// the sergeant is going to ask ALL RECRUITS some questions
	// and then each recruit, based on their service, is going
	// to be asked specific questions

	int i;
	int num_recruits;

	cout << "How many people are waiting? ";
	cin >> num_recruits;

	cout << "Each recruit is asked what service \n"
		<< "They would like to join \n"
		<< "\t0\tArmy\n"
		<< "\t1\tNavy\n"
		<< "\t2\tAir Force\n"
		<< "\t3\tMarines\n"
		<< "\t4\tDoesn't Know\n\n";
```

```
	for ( i = 0; i < num_recruits; i++ )
	{
		int service;

		cout << "\n\nWhich service does recruit " << i
			<< " wish to join?" << endl;
		cin >> service;

		// the statement array[i] = new  army/navy/air_force/marines is an extremely powerful
		// statement.  Even though we are creating instances of four different classes, we are
		// storing the pointer to the instances in ONE DATA STRUCTURE which is of
		// type pointer to base class that they are derived from. Instead of needing four
		// data structures, one for each service, we are able to have one data structure.  This
		// can be extremely useful in reducing the complexity of algorithms.
		switch (service )
		{
			case 0:
				cout << "Recruit " << i
					<< " is going in the army\n";
				array[i]  = new army;
				break;
			case 1:
				cout << "Recruit " << i
					<< " is going in the navy\n";
				array[i]  = new navy;
				break;

			case 2:
				cout << "Recruit " << i
					<< " is going in the air force\n";
				array[i]  = new air_force;
				break;

			case 3:
				cout << "Recruit " << i
					<< " is going in the marines\n";
				array[i]  = new marines;
				break;

			default:
				// later on, when they decide what service
				// they are going in, we will have to be
				// able to cast from base to derived
				cout << "Recruit " << i
					<< " doesn't know where they're"
					<< " going \n";
				array[i]  = new common_class;
				break;
		} // end switch

```

```
		// We are now going to exercise another useful feature of inheritance and virtual
		// functions.  The questions q1 and q2 are defined in the base class only.  The binding
		// of these calls is unambiguous.  We are able to call the q1 of the base class for
		// members of each of the four deriving classes via a pointer to the base class
		// NOTICE that we do not have to switch on which service the recruit is in to be
		// able to call these functions.
		array[i]->q1();
		array[i]->q2();

		// ask the specific questions
		// the question q3 is specific to each of the services, and yet we can call it via a pointer
		// to the base class.  We are able to call four DIFFERENT functions, but with one line
		// of syntactically exact code.  Remember that q3 is a virtual function.  We are invoking
		// it via a pointer to a base class.  If the value in the array is an instance of the base class
		// then the base class function will be called.  If the value in the array is an instance of
		// a derived class, and the derived class has a polymophic version of the virtual function,
		// then the derived class function will be called.
		// NOTICE that we do not have to switch on which service the recruit is in to be able
		// to ask this service specific question.  Due to how the virtual function call will be
		// resolved, if the service has overridden this call, the service specific call will be made
		// if the service has not overridden this call, the common class call will be made
		// if the recruit is an undecided class recruit, then the base class version will be called.
		array[i]->q3();
	}

	// go through the array and print out the common fields
	// name and address
	// and the specific values found in the objects
	cout << "\n\nLIST OF RECRUITS\n";
	for ( i = 0; i < num_recruits; i++ )
	{
		array[i]->print_vals();
	}
}
```

```
Input Data File Used To Generate Output
12
0
army guy 1
100 army way
10
1
navy guy 1
123 navy street
y
1
navy guy 2
222 navy blvd
n
2
air force recruit 1
99 air force court
n
2
air force recruit 2
100 air force way
y
3
marine guy 1
123 marine street
3
marine guy 2
222 marine blvd
4
undecided recruit 1
99 wishy washy way
2
air force recruit 3
100 air force way
y
3
marine guy 3
999 marine street
3
marine guy 4
2354222 marine blvd
4
undecided recruit 2
100 don't know where I live
```

```
Output From Running Program
How many people are waiting? Each recruit is asked what service
They would like to join
        0       Army
        1       Navy
        2       Air Force
        3       Marines
        4       Doesn't Know

Which service does recruit 0 wish to join?
Recruit 0 is going in the army
What is your name?
What is your address?
How many miles can you run?

Which service does recruit 1 wish to join?
Recruit 1 is going in the navy
What is your name?
What is your address?
Can you swim?   n => no    y => yes

Which service does recruit 2 wish to join?
Recruit 2 is going in the navy
What is your name?
What is your address?
Can you swim?   n => no    y => yes

Which service does recruit 3 wish to join?
Recruit 3 is going in the air force
What is your name?
What is your address?
Are you afraid of heights?   n => no   y => yes

Which service does recruit 4 wish to join?
Recruit 4 is going in the air force
What is your name?
What is your address?
Are you afraid of heights?   n => no   y => yes

Which service does recruit 5 wish to join?
Recruit 5 is going in the marines
What is your name?
What is your address?

Which service does recruit 6 wish to join?
Recruit 6 is going in the marines
What is your name?
What is your address?
```

```
Which service does recruit 7 wish to join?
Recruit 7 doesn't know where they're going
What is your name?
What is your address?
I know you haven't decided which service you want
to go into, so let me give you some information
on each!


Which service does recruit 8 wish to join?
Recruit 8 is going in the air force
What is your name?
What is your address?
Are you afraid of heights?   n => no   y => yes


Which service does recruit 9 wish to join?
Recruit 9 is going in the marines
What is your name?
What is your address?


Which service does recruit 10 wish to join?
Recruit 10 is going in the marines
What is your name?
What is your address?


Which service does recruit 11 wish to join?
Recruit 11 doesn't know where they're going
What is your name?
What is your address?
I know you haven't decided which service you want
to go into, so let me give you some information
on each!


LIST OF RECRUITS
ARMY RECRUIT
army guy 1
100 army way
Can run 10 At a time

NAVY RECRUIT
navy guy 1
123 navy street
Can Swim

NAVY RECRUIT
navy guy 2
222 navy blvd
Cannot Swim

AIR FORCE RECRUIT
air force recruit 1
```

```
99 air force court
Is not afraid of heights

AIR FORCE RECRUIT
air force recruit 2
100 air force way
Is afraid of heights

MARINE RECRUIT
marine guy 1
123 marine street

MARINE RECRUIT
marine guy 2
222 marine blvd

undecided recruit 1
99 wishy washy way
AIR FORCE RECRUIT
air force recruit 3
100 air force way
Is afraid of heights

MARINE RECRUIT
marine guy 3
999 marine street

MARINE RECRUIT
marine guy 4
2354222 marine blvd

undecided recruit 2
100 don't know where I live

```

Section 8

Dynamic Memory Allocation

C++ replaces malloc, calloc and free with new and delete.
C++ also provides constructors that allow code we have written to be called when an instance of an object is created. This provides us with a great opportunity to gaurantee the initial states of our objects. This chapter goes over every last little detail there is involved with dynamic memory allocation.

new1.cpp	this program introduces the very basics of new as applied to pre defined types
new2.cpp	using new with a user defined structure
new3.cpp	using new to create arrays of pre defined types
new4.cpp	using new to create arrays of user defined structures
new5.cpp	using new with user defined classes
new6.cpp	new and arrays of user defined classes
new7.cpp	new with a class embedded inside another class how is the memory allocation for the embedded class handled
new8.cpp	new and inheritance
new9.cpp	new and arrays of inherited objects
new10.cpp	overloading the new operator for a user defined class

```
// new1.cpp
// new replaces malloc and calloc from C
// malloc can still be used, calloc can still be used
// delete replaces free from C

#include <iostream.h>
#include <stdlib.h>
main()
{
	int *  iarray  = new int   [10];
	float * farray = new float [10];
	char *  carray = new char  [10];

	cout << "Size of iarray is " << sizeof(iarray) << endl;
	cout << "Size of farray is " << sizeof(farray) << endl;
	cout << "Size of carray is " << sizeof(carray) << endl;

	cout << "Size of *iarray is " << sizeof(*iarray) << endl;
	cout << "Size of *farray is " << sizeof(*farray) << endl;
	cout << "Size of *carray is " << sizeof(*carray) << endl;

	int i;
	for ( i = 0; i < 10; i++ )
	{
		iarray[i] = (int)   i;
		farray[i] = (float) i;
		carray[i] = 'a' + i;
	}

	char temp;
	for ( i = 0; i < 10; i++ )
	{
		cout << "Address of iarray sub " << i << " is "
			<< &iarray[i] << " Value stored there is "
			<< iarray[i] << endl;
		cout << "Value stored at farray sub " << i << " is "
			<< farray[i] << endl;
		temp = carray[i];
		cout << "Value stored at carray sub " << i << " is "
			<< temp << endl;
	}
}
```

```
Output From Running Program
Size of iarray is 2
Size of farray is 2
Size of carray is 2
Size of *iarray is 2
Size of *farray is 4
Size of *carray is 1
Address of iarray sub 0 is 0x1f9f1460 Value stored there is 0
Value stored at farray sub 0 is 0
Value stored at carray sub 0 is a
Address of iarray sub 1 is 0x1f9f1462 Value stored there is 1
Value stored at farray sub 1 is 1
Value stored at carray sub 1 is b
Address of iarray sub 2 is 0x1f9f1464 Value stored there is 2
Value stored at farray sub 2 is 2
Value stored at carray sub 2 is c
Address of iarray sub 3 is 0x1f9f1466 Value stored there is 3
Value stored at farray sub 3 is 3
Value stored at carray sub 3 is d
Address of iarray sub 4 is 0x1f9f1468 Value stored there is 4
Value stored at farray sub 4 is 4
Value stored at carray sub 4 is e
Address of iarray sub 5 is 0x1f9f146a Value stored there is 5
Value stored at farray sub 5 is 5
Value stored at carray sub 5 is f
Address of iarray sub 6 is 0x1f9f146c Value stored there is 6
Value stored at farray sub 6 is 6
Value stored at carray sub 6 is g
Address of iarray sub 7 is 0x1f9f146e Value stored there is 7
Value stored at farray sub 7 is 7
Value stored at carray sub 7 is h
Address of iarray sub 8 is 0x1f9f1470 Value stored there is 8
Value stored at farray sub 8 is 8
Value stored at carray sub 8 is i
Address of iarray sub 9 is 0x1f9f1472 Value stored there is 9
Value stored at farray sub 9 is 9
Value stored at carray sub 9 is j
```

```
//new2.cpp
// new also works with structures
// new will return a pointer to an instance of the desired structure
// type that you request.
// If new returns NULL as the address of the new variable,
//          then the dynamic memory allocation failed
#include <iostream.h>

struct XY
{
        int x;
        float y;
};

main()
{
        struct XY * X = new struct XY;

        cout << "new returned " << X << endl;
        cout << "Size of X is " << sizeof(X) << endl;
        cout << "Size of *X is " << sizeof(*X) << endl;
}
```

Output From Running Program

```
new returned 0x18140fe6
Size of X is 2
Size of *X is 6
```

```
// new3.cpp
// new is not limited to creating single instances of variables
// new can be used to create arrays
#include <iostream.h>
main()
{
	int * iarray = new int [10];
	float * farray = new float [10];
	char * carray = new char [10];

	cout << "Size of iarray is " << sizeof(iarray) << endl;
	cout << "Size of farray is " << sizeof(farray) << endl;
	cout << "Size of carray is " << sizeof(carray) << endl;

	cout << "Size of *iarray is " << sizeof(*iarray) << endl;
	cout << "Size of *farray is " << sizeof(*farray) << endl;
	cout << "Size of *carray is " << sizeof(*carray) << endl;

	int i;
	for ( i = 0; i < 10; i++ )
	{
		iarray[i] = (int) i;
		farray[i] = (float) i;
		carray[i] = 'a' + i;
	}

	char temp;
	for ( i = 0; i < 10; i++ )
	{
		cout << "Address of iarray sub " << i << " is"
		<< &iarray[i] << " Value stored there is "
		<< iarray[i] << endl;
		cout << "Address of farray sub " << i << " is"
		<< &farray[i] << " Value stored there is "
		<< farray[i] << endl;

		temp = carray[i];
		cout << "Value stored at carray sub " << i << " is "
			<< temp << endl;
	}
}
```

```
Output From Running Program
Size of iarray is 2
Size of farray is 2
Size of carray is 2

Size of *iarray is 2
Size of *farray is 4
Size of *carray is 1

Address of iarray sub 0 is0x1c5f1470 Value stored there is 0
Address of farray sub 0 is0x1c5f1488 Value stored there is 0
Value stored at carray sub 0 is a

Address of iarray sub 1 is0x1c5f1472 Value stored there is 1
Address of farray sub 1 is0x1c5f148c Value stored there is 1
Value stored at carray sub 1 is b

Address of iarray sub 2 is0x1c5f1474 Value stored there is 2
Address of farray sub 2 is0x1c5f1490 Value stored there is 2
Value stored at carray sub 2 is c

Address of iarray sub 3 is0x1c5f1476 Value stored there is 3
Address of farray sub 3 is0x1c5f1494 Value stored there is 3
Value stored at carray sub 3 is d

Address of iarray sub 4 is0x1c5f1478 Value stored there is 4
Address of farray sub 4 is0x1c5f1498 Value stored there is 4
Value stored at carray sub 4 is e

Address of iarray sub 5 is0x1c5f147a Value stored there is 5
Address of farray sub 5 is0x1c5f149c Value stored there is 5
Value stored at carray sub 5 is f

Address of iarray sub 6 is0x1c5f147c Value stored there is 6
Address of farray sub 6 is0x1c5f14a0 Value stored there is 6
Value stored at carray sub 6 is g

Address of iarray sub 7 is0x1c5f147e Value stored there is 7
Address of farray sub 7 is0x1c5f14a4 Value stored there is 7
Value stored at carray sub 7 is h

Address of iarray sub 8 is0x1c5f1480 Value stored there is 8
Address of farray sub 8 is0x1c5f14a8 Value stored there is 8
Value stored at carray sub 8 is i

Address of iarray sub 9 is0x1c5f1482 Value stored there is 9
Address of farray sub 9 is0x1c5f14ac Value stored there is 9
Value stored at carray sub 9 is j
```

```
//new4.cpp
// new also works with arrays of structures
// new will return a pointer to an instance of the desired structure
// type that you request.
// If new returns NULL as the address of the new variable,
//          then the dynamic memory allocation failed
#include <iostream.h>

struct XY
{
        int x;
        float y;
};

main()
{
        int i;
        struct XY * Xarray = new struct XY [10];

        cout << "new returned " << Xarray << endl;
        cout << "Size of X is " << sizeof(Xarray) << endl;
        cout << "Size of *X is " << sizeof(*Xarray) << endl;

        for ( i = 0; i < 10; i++ )
        {
                cout << "Address of Xarray sub " << i << " is "
                << &Xarray[i] << endl;
        }
        cout << endl;
}

Output From Running Program
new returned 0x181a1002
Size of X is 2
Size of *X is 6

Address of Xarray sub 0 is 0x181a1002
Address of Xarray sub 1 is 0x181a1008
Address of Xarray sub 2 is 0x181a100e
Address of Xarray sub 3 is 0x181a1014
Address of Xarray sub 4 is 0x181a101a
Address of Xarray sub 5 is 0x181a1020
Address of Xarray sub 6 is 0x181a1026
Address of Xarray sub 7 is 0x181a102c
Address of Xarray sub 8 is 0x181a1032
Address of Xarray sub 9 is 0x181a1038
```

```
// new5.cpp
// new also works with classes
// new will return a pointer to an instance of the desired class type
// that you request
// if new returns NULL as the address of the new variable,
//          then the dynamic memory allocation failed.

#include <iostream.h>

class XY
{
        private:
                int x;
                float y;
        public:
                XY();
                void show_x_and_y();
};

// just slam some values in x and y
// put in a cout to show when this thing is called
// this will be very instructive when we are dealing with arrays
XY::XY()
{
        cout << "XY no arg constructor called " << endl;
        x = 10;
        y = 11.1;
}

void XY::show_x_and_y()
{
        cout << "x is " << x << " y is " << y << endl;
}

main()
{
        XY * X = new XY;

        cout << "Size of X is " << sizeof(X) << endl;
        cout << "Size of *X is " << sizeof(*X) << endl;

        // remember that the thing returned by new is a POINTER
        X->show_x_and_y();
}
```

Output From Running Program

```
XY no arg constructor called
Size of X is 2
Size of *X is 6
x is 10 y is 11.1
```

```
// new6.cpp
// new also works with arrays of classes
// new will return a pointer to an instance of the desired class type
// that you request
// if new returns NULL as the address of the new variable,
//          then the dynamic memory allocation failed.
#include <iostream.h>
class XY
{
        private:
                int x;
                float y;
                static int xval;        // use for differing init values
                static float yval;
        public:
                XY();
                void show_x_and_y();
};

int XY::xval = 0;
float XY::yval = 0.0;

// put in a cout to show when this thing is called
// this will be very instructive when we are dealing with arrays
XY::XY()
{
        cout << "XY no arg constructor called " << endl;
        x = xval;
        xval = xval + 1;
        y = yval;
        yval = yval + 1.0;
}

void XY::show_x_and_y()
{
        cout << "x is " << x << " y is " << y << endl;
}

main()
{
        int i;
        XY * X = new XY[5];

        cout << "Size of X is " << sizeof(X) << endl;
        cout << "Size of *X is " << sizeof(*X) << endl;

        for ( i = 0; i < 5; i++ )
        {
                // remember that the thing returned by new is a POINTER
                X[i].show_x_and_y();
        }
}
```

```
Output From Running Program
XY no arg constructor called
XY no arg constructor called
XY no arg constructor called
XY no arg constructor called
XY no arg constructor called
Size of X is 2
Size of *X is 6
x is 0 y is 0
x is 1 y is 1
x is 2 y is 2
x is 3 y is 3
x is 4 y is 4
```

```
// new7.cpp
// new also works with classes where other classes are embedded
// new will return a pointer to an instance of the desired class type
// that you request
// if new returns NULL as the address of the new variable,
//	then the dynamic memory allocation failed.

#include <iostream.h>

class XY
{
	private:
		int x;
		float y;
		static int xval;		// use for differing init values
		static float yval;
	public:
		XY();
		void show_x_and_y();
		void setxy(int,float);
};

int XY::xval = 0;
float XY::yval = 0.0;

// put in a cout to show when this thing is called
// this will be very instructive when we are dealing with arrays
XY::XY()
{
	cout << "XY no arg constructor called " << endl;
	x = xval;
	xval = xval + 1;
	y = yval;
	yval = yval + 1.0;
}

void XY::show_x_and_y()
{
	cout << "x is " << x << " y is " << y << endl;
}

void XY::setxy(int i, float f)
{
	x = i;
	y = f;
}
```

```
// create a new class into which an instance of an XY class object will
// be embedded.  This will be interesting to watch in what order the
// constructors are called.
class AB
{
        private:
                int a;
                int b;
                XY x1;   // an XY is embedded in AB
        public:
                AB();
                void show_ab();
};

AB::AB()
{
        cout << "AB no arg constructor called " << endl;
        a = -1;
        b = -2;
        // at this point the no arg constructor was already called for x1
        // we are CHANGING, not INITIALIZING the values for x and y of x1
        x1.setxy(4,5.0);
}

void AB::show_ab()
{
        cout << "a is " << a << " b is " << b << endl;
        x1.show_x_and_y();          // remember to use the public member function
                                    // x and y are private data of an XY that
                                    // is embedded, not inherited from.
        cout << endl;
}

main()
{
        int i;
        XY * X = new XY;            // first create an instance of an XY  It still works as we'd expect

        cout << "Size of X is " << sizeof(X) << endl;
        cout << "Size of *X is " << sizeof(*X) << endl;
        X->show_x_and_y();
        cout << endl << endl;

        //now create an insance of an AB
        // an XY will need to be created to complete the construction
        // of the AB.
        // Whose constructor will be called first?  The AB or the XY?
        AB * A = new AB;

        cout << "Size of A is " << sizeof(A) << endl;
        cout << "Size of *A is " << sizeof(*A) << endl;
        A->show_ab();
}
```

```
XY no arg constructor called
Size of X is 2
Size of *X is 6
x is 0 y is 0


// the AB constructor is called, but before the first executable statement
// of the AB constructor can be performed, ALL the memory for an instance
// of an AB must be acquired, therefore the no argument version of the
// XY constructor is invoked and that's why these prints occur in the
// order that they do!
XY no arg constructor called
AB no arg constructor called
Size of A is 2
Size of *A is 10
a is -1 b is -2
x is 4 y is 5

```

```
// new8.cpp
// new also works with classes where inheritance is involved
// new will return a pointer to an instance of the desired class type
// that you request
// if new returns NULL as the address of the new variable,
//          then the dynamic memory allocation failed.

#include <iostream.h>

class XY
{
        private:
                int x;
                float y;
                static int xval;        // use for differing init values
                static float yval;
        public:
                XY();
                void show_x_and_y();
                void setxy(int,float);
};

int XY::xval = 0;
float XY::yval = 0.0;

// put in a cout to show when this thing is called
// this will be very instructive when we are dealing with arrays
XY::XY()
{
        cout << "XY no arg constructor called " << endl;
        x = xval;
        xval = xval + 1;
        y = yval;
        yval = yval + 1.0;
}

void XY::show_x_and_y()
{
        cout << "x is " << x << " y is " << y << endl;
}

void XY::setxy(int i, float f)
{
        x = i;
        y = f;
}
```

```
// the MNO class will inherit from the XY class
// again, all the memory for an MNO must be acquired before the first
// executable statement of the MNO constructor can be run
// therefore we will continue to be interested in the order in which
// the XY constructors are called when creating MNO objects
class MNO : public XY
{
	private:
		int m,n,o;
	public:
		MNO();
		void show_mno();
};

MNO::MNO()
{
	cout << "MNO no arg constructor was called " << endl;
	m = -1;		n = -2;		o = -3;
	// at this point we will change the values for x and y
	// they have already been set in the XY constructor
	cout << "Initial values of x and y are \n";
	show_x_and_y();
	setxy(24,35.0);
	cout << "Post setxy values of x and y are \n";
	show_x_and_y();
}
void MNO::show_mno()
{
	cout << "m is " << m << endl;
	cout << "n is " << n << endl;
	cout << "o is " << o << endl;
	show_x_and_y();
	cout << endl;
}

main()
{
	int i;
	XY * X = new XY;	// first create an instance of an XY It still works as we'd expect
	cout << "Size of X is " << sizeof(X) << endl;
	cout << "Size of *X is " << sizeof(*X) << endl;
	X->show_x_and_y();
	cout << endl << endl;

	//now create an insance of an MNO
	// an XY will need to be created to complete the construction  of the MNO.
	// Whose constructor will be called first?  The MNO or the XY?
	MNO * M = new MNO;
	cout << endl << endl;
	cout << "Size of M is " << sizeof(M) << endl;
	cout << "Size of *M is " << sizeof(*M) << endl;
	M->show_mno();
}
```

```
Output From Running Program
XY no arg constructor called
Size of X is 2
Size of *X is 6
x is 0 y is 0


XY no arg constructor called
MNO no arg constructor was called
Initial values of x and y are
x is 1 y is 1
Post setxy values of x and y are
x is 24 y is 35


Size of M is 2
Size of *M is 12
m is -1
n is -2
o is -3
x is 24 y is 35
```

```
// new9.cpp
// new also works with arrays of classes where inheritance is involved
// new will return a pointer to an instance of the desired class type
// that you request
// if new returns NULL as the address of the new variable,
//          then the dynamic memory allocation failed.

#include <iostream.h>

class XY
{
        private:
                int x;
                float y;
                static int xval;     // use for differing init values
                static float yval;
        public:
                XY();
                void show_x_and_y();
                void setxy(int,float);
};

int XY::xval = 0;
float XY::yval = 0.0;

// put in a cout to show when this thing is called
// this will be very instructive when we are dealing with arrays
XY::XY()
{
        cout << "XY no arg constructor called " << endl;
        x = xval;
        xval = xval + 1;
        y = yval;
        yval = yval + 1.0;
}

void XY::show_x_and_y()
{
        cout << "x is " << x << " y is " << y << endl;
}

void XY::setxy(int i, float f)
{
        x = i;
        y = f;
}
```

```
// the MNO class will inherit from the XY class
// again, all the memory for an MNO must be acquired before the first
// executable statement of the MNO constructor can be run
// therefore we will continue to be interested in the order in which
// the XY constructors are called when creating MNO objects

class MNO : public XY
{
	private:
		int m,n,o;
	public:
	MNO();
	void show_mno();
};

MNO::MNO()
{
	cout << "MNO no arg constructor was called " << endl;
	m = -1;
	n = -2;
	o = -3;
	// at this point we will change the values for x and y
	// they have already been set in the XY constructor
	cout << "Initial values of x and y are \n";
	show_x_and_y();
	setxy(24,35.0);
	cout << "Post setxy values of x and y are \n";
	show_x_and_y();
}

void MNO::show_mno()
{
	cout << "m is " << m << endl;
	cout << "n is " << n << endl;
	cout << "o is " << o << endl;
	show_x_and_y();
	cout << endl;
}
```

```
main()
{
	int i;
	XY * X = new XY;		// first create an instance of an XY It still works as we'd expect

	cout << "Size of X is " << sizeof(X) << endl;
	cout << "Size of *X is " << sizeof(*X) << endl;
	X->show_x_and_y();
	cout << endl << endl;

	//now create an insance of an MNO
	// an XY will need to be created to complete the construction
	// of the MNO.
	// Whose constructor will be called first?  The MNO or the XY?
	MNO * M = new MNO;
	cout << endl << endl;

	cout << "Size of M is " << sizeof(M) << endl;
	cout << "Size of *M is " << sizeof(*M) << endl;
	M->show_mno();

	// now create an array of MNOs
	MNO * Marray = new MNO[5];
	cout << endl << endl;

	cout << "Size of Marray is " << sizeof(Marray) << endl;
	cout << "Size of *Marray is " << sizeof(*Marray) << endl;

	for ( i = 0; i < 5; i++ )
		Marray[i].show_mno();
}
```

```
Output From Running Program
XY no arg constructor called
Size of X is 2
Size of *X is 6
x is 0 y is 0
XY no arg constructor called
MNO no arg constructor was called
Initial values of x and y are
x is 1 y is 1
Post setxy values of x and y are
x is 24 y is 35
Size of M is 2
Size of *M is 12
m is -1
n is -2
o is -3
x is 24 y is 35
XY no arg constructor called
MNO no arg constructor was called
Initial values of x and y are
x is 2 y is 2
Post setxy values of x and y are
x is 24 y is 35
XY no arg constructor called
MNO no arg constructor was called
Initial values of x and y are
x is 3 y is 3
Post setxy values of x and y are
x is 24 y is 35
XY no arg constructor called
MNO no arg constructor was called
Initial values of x and y are
x is 4 y is 4
Post setxy values of x and y are
x is 24 y is 35
XY no arg constructor called
MNO no arg constructor was called
Initial values of x and y are
x is 5 y is 5
Post setxy values of x and y are
x is 24 y is 35
XY no arg constructor called
MNO no arg constructor was called
Initial values of x and y are
x is 6 y is 6
Post setxy values of x and y are
x is 24 y is 35
```

```
Size of Marray is 2
Size of *Marray is 12
m is -1
n is -2
o is -3
x is 24 y is 35

m is -1
n is -2
o is -3
x is 24 y is 35

m is -1
n is -2
o is -3
x is 24 y is 35

m is -1
n is -2
o is -3
x is 24 y is 35

m is -1
n is -2
o is -3
x is 24 y is 35

```

```
// new10.cpp
// new is an operator
// operators may be overloaded
// therefore new may be overloaded
// WARNING, THIS PROGRAM WORKS DIFFERENTLY AND NEEDS DIFFERENT #INCLUDES
// ON DIFFERENT OPERATING SYSTEMS.  THIS VERSION IS FOR DOS PCs
#include <iostream.h>
#include <stddef.h>
#include <stdlib.h>

class XY
{
	private:
		int x;
		float y;
	public:
		XY();
		XY(int);
		void show_x_and_y();
		void setxy(int,float);
		void * operator new(size_t);
		void * operator new(size_t,int);
		void * operator new(size_t,int,int);
};

// put in a cout to show when this thing is called
// this will be very instructive when we are dealing with arrays
XY::XY()
{
	cout << "XY no arg constructor called " << endl;
	x = 10;
	y = 11.1;
}

XY::XY(int i )
{
	cout << "XY one arg constructor called " << endl;
	x = i;
	y = 999.999;
}

void XY::show_x_and_y()
{
	cout << "x is " << x << " y is " << y << endl;
}

void XY::setxy(int i, float f)
{
	x = i;
	y = f;
}
```

```
// this is a new and interesting thing to do
// we are taking over the new function for the XY class
// taking over new for this class in NO WAY affects new for any other class
void * XY::operator new (size_t size_of_item)
{
        // for the listed number of items, invoke the constructor
        cout << "The new(size_t) operator for XY class was invoked\n";
        cout << "size_of_item was " << size_of_item << endl;

        // we are going to allocate enough memory here
        cout << "Calling malloc.... ";
        XY * ptr = ( XY*) malloc (size_of_item);
        cout << "Back from malloc\n";

        // we are going to pass a pointer to the memory to the constructor
        // the next statement executed after this one will be the first
        // executable statement of the constructor
        return ( ptr );
}

// now we are going to provide a polymorphic version of the new operator
// remember this affects only new for the XY class, not any other class
// the new function will be called to get the memory and then the
// constructor will be invoked
void * XY::operator new(size_t size_of_item, int arg1 )
{
        cout << "The new(size_t,int) operator for XY class was invoked\n";
        cout << "size_of_item was " << size_of_item << endl;
        cout << "arg1 is " << arg1 << endl;
        XY * ptr = (XY*) malloc (size_of_item);
        return NULL;
}

// now we are going to provide another polymorphic version of the new operator
// remember this affects only new for the XY class, not any other class
// the new function will be called to get the memory and then the
// constructor will be invoked
void * XY::operator new(size_t size_of_item, int arg1, int arg2 )
{
        cout << "The new(size_t,int,int) operator for XY class was invoked\n";
        cout << "size_of_item was " << size_of_item << endl;
        cout << "arg1 is " << arg1 << endl;
        cout << "arg2 is " << arg2 << endl;
        XY * ptr = (XY*) malloc (size_of_item);
        return NULL;
}
```

```
main()
{
	int i;

	cout << "AAAAA" << endl;
	// the one argument new operator will be called
	// the one argument will be the size of an XY
	XY * X1 = new XY;			// first create an instance of an XY
	cout << "Size of X1 is " << sizeof(X1) << endl;
	cout << "Size of *X1 is " << sizeof(*X1) << endl;
	X1->show_x_and_y();
	cout << endl << endl;


	cout << "BBBBB" << endl;
	// the one argument new operator will be called
	// the one argument will be the size of an XY
	XY * X2 = new XY(4);			// cause one arg constructor to be called
	cout << "Size of X2 is " << sizeof(X2) << endl;
	cout << "Size of *X2 is " << sizeof(*X2) << endl;
	X2->show_x_and_y();
	cout << endl << endl;


	// this will NOT cause any of our new operator functions to be called
	// WHY?
	// Because C++ has a global new operator that is used for allocation
	// of memory for arrays and we don't take it over
	cout << "CCCCC" << endl;
	XY * X3 = new XY[3];			// no initializers can be specified for arrays
	cout << "Size of X3 is " << sizeof(X3) << endl;
	cout << "Size of *X3 is " << sizeof(*X3) << endl;
	X3[0].show_x_and_y();
	X3[1].show_x_and_y();
	X3[2].show_x_and_y();
	cout << endl << endl;



	cout << "DDDDD" << endl;
	// the 1 will be passed in as arg1 to the two arg new operator
	// the 4 will be translated to 4*sizeof(XY) and passed as the first arg
	XY * X4 = new (1) XY[4];
	cout << "Size of X4 is " << sizeof(X4) << endl;
	cout << "Size of *X4 is " << sizeof(*X4) << endl;
	X4[0].show_x_and_y();
	X4[1].show_x_and_y();
	X4[2].show_x_and_y();
	X4[3].show_x_and_y();
	cout << endl << endl;
```

```
        cout << "EEEEE" << endl;
        // the 10 will be passed in as arg1 to the two arg new operator
        // the 20 will be passed in as arg2 to the two arg new operator
        // the 5 will be translated to 5*sizeof(XY) and passed as the first arg
        XY * X5 = new (10,20) XY[5];
        cout << "Size of X5 is " << sizeof(X5) << endl;
        cout << "Size of *X5 is " << sizeof(*X5) << endl;
        X5[0].show_x_and_y();
        X5[1].show_x_and_y();
        X5[2].show_x_and_y();
        X5[3].show_x_and_y();
        X5[4].show_x_and_y();
        cout << endl << endl;
}
```

Output From Running Program

```
AAAAA
The new(size_t) operator for XY class was invoked
size_of_item was 6
Calling malloc.... Back from malloc
XY no arg constructor called
Size of X1 is 2
Size of *X1 is 6
x is 10 y is 11.1
BBBBB
The new(size_t) operator for XY class was invoked
size_of_item was 6
Calling malloc.... Back from malloc
XY one arg constructor called
Size of X2 is 2
Size of *X2 is 6
x is 4 y is 999.999023
CCCCC
XY no arg constructor called
XY no arg constructor called
XY no arg constructor called
Size of X3 is 2
Size of *X3 is 6
x is 10 y is 11.1
x is 10 y is 11.1
x is 10 y is 11.1
DDDDD
The new(size_t,int) operator for XY class was invoked
size_of_item was 24
arg1 is 1
The new(size_t) operator for XY class was invoked
size_of_item was 6
Calling malloc.... Back from malloc
XY no arg constructor called
XY no arg constructor called
XY no arg constructor called
XY no arg constructor called
Size of X4 is 2
Size of *X4 is 6
x is 0 y is 2.687419e+32
x is 10 y is 11.1
x is 10 y is 11.1
x is 10 y is 11.1
```

```
EEEEE
The new(size_t,int,int) operator for XY class was invoked
size_of_item was 30
arg1 is 10
arg2 is 20
The new(size_t) operator for XY class was invoked
size_of_item was 6
Calling malloc.... Back from malloc
XY no arg constructor called
XY no arg constructor called
XY no arg constructor called
XY no arg constructor called
XY no arg constructor called
Size of X5 is 2
Size of *X5 is 6
x is 0 y is 2.687419e+32
x is 10 y is 11.1
x is 10 y is 11.1
x is 10 y is 11.1
x is 10 y is 11.1
```

Section 9

This section is about C++ i/o.
Many C++ programmers just do C i/o and are quite happy.
However, C++ does introduce some new concepts which can be useful.
C++ provides persistent and non persistent formatting of output.
C++ provides different ways to interpret input streams.
C++ provides various methods to open files, via open calls or via constructors.
C++ allows the user to take over the i/o operators for user defined classes and thus remove the need for the programmer to learn special i/o subroutines to use your class.

This chapter also covers the file error handling and file controlling bits of C++

coutcin.cpp	this program illustrates very simple usage of cout and cin		
coutfmt1.cpp	do some fundamental formatting using cout		
coutfmt2.cpp	the use of ios flags for formatting using cout		
coutfmt3.cpp	persistence of formatting		
coutfmt4.cpp	width and justification		
flushit.cpp	the buffering of output, the separate buffers of printf and cout		
testws.cpp	the treatment of whitespace characters and the skipws ios flag		
ostream1.cpp	ostream functions tellp flush put write		
ostream2.cpp	ostream flags		
ostream3.cpp	ostream flags and functions, good, eof, fail, bad, clear		
istream1.cpp	istream functions get, peek, putback, read, getline, seekg, tellg		
istream2.cpp	istream flags		
istream3.cpp	istream flags and functions, good, bad, fail, clear		
finfout.cpp	file io		
finfout1.cpp	file io		
finfout2.cpp	file io		
finfout3.cpp	file io		
ostristr.cpp	stream status		
predefs.cpp	predefined C++ streams cout, cin, cerr, clog		
filebit1.cpp	filebit1.dat	filebit1.out	fstream bits
filebit2.cpp	filebit2.dat	filebit2.out	fstream bits and functions

fileop1.cpp	file operations
fileop2.cpp	file open methods

```
// coutcin.cpp
// C++ allows the programmer to use printf, scanf and other stdio.h routines
// C++ also provides new input, output and error processing abilities
// C++ introduces the concept of a stream
// You place things in an output stream
// Putting things in an output stream is a binary operation that requires
// the name of the stream to be written to and the object to be written
// the operation returns a reference to the stream written to

// You take things from an input stream
// Taking things from an input stream is a binary operation that requires
// the name of the stream to be read from and a location to write the input
// object into.

// The default C++ output stream is named cout
// The default C++ input  stream is named cin
// The user gains access to cout and cin through iostream.h

#include <stdio.h>                      // get printf and scanf
#include <iostream.h>                   // get cout and cerr

main()
{
        int i = 0;

        // use the i/o routines from stdio.h
        printf("Value for i please? ");         // printf is a function
        scanf("%i",&i);                         // scanf is a function
        printf("i is %i\n",i);
        printf("\n");

        cout << "Input a value for i please \n";        // cout is not a function
        // cout is a STREAM.  Specifically it is an ostream object
        // an output stream object.
        // It accepts things to be printed, strings, integers, chars ...
        // we place the string "Input a value for i please\n" in the stream
        // using the binary operator <<
        // The << operator, when used with an ostream object, like cout
        // passes the item on its right to the ostream object on its left.
        // The operation returns an ostream operator to allow for chaining.

        cin >> i;                       // cin is not a function
        // cin is a STREAM.  Specifically it is an istream object
        // an input stream object.
        // It reads in things and puts them in the specified place
        // The >> operator, when used with an istream object, like cin
        // passes a value from the istream on the left to the variable
        // on the right.  The operation returns an istream operator to allow
        // for chaining.
```

```
        cout << "You entered " << i << " \n";           // chaining
        // note that no formatting informatino about i was provided
        // in printf we would have to specify a format like %i or %o
        // the cout stream has default formats for each data type
        // in C++, each variable knows its type at run time, therefore
        // information about i, an integer, does not have to be passed
        // explicitly to the cout stream.
        cout << "\n";
}
```

Output From Running Program

coutcin.out

```
Value for i please? i is 4

Input a value for i please
You entered 12
```

```
// coutfmt1.cpp
// this program illustrates some formatting of output
// C allows the printf function to use %i %5i %05i to control width and fill
// C allows the printf function to use %f %.2f %7.2f to control width and fill
// C allows the printf function to use %i %o %x to control base

// cout is not a function, it is an ostream object
// The object maintains information about width, fill, padding, justification
// We can tell the cout object to change state via messages that we send it
// Some messages that we send are "enumerated types" hex, oct, dec, endl, ends
// Some messages that we send are "non-enumerated types"
// setw, setprecision, setbase, setfill, setiosflags, resetiosflags

#include <stdio.h>
#include <iostream.h>                  // get access to the cout and cin objects
#include <iomanip.h>                   // get access to modifiers for cout cin objects
main()
{
        int i = 13; int j = 15;  int k = 7;

        // C and C++ both print integers by default in base ten
        printf("i = %i\n",i);   cout << "i = " << i << endl;

        // the %i %o %x indicates what base to print i in
        printf("i base 10 %i  base 8 %o  base 16 %x\n",i,i,i);

        // the dec, oct, hex tells the cout object in what base to prints ints
        cout << "i base 10 " << dec << i << endl;
        cout << "i base 8  " << oct << i << endl;
        cout << "i base 16 " << hex << i << endl;

        // In C, you need to specify the base for each integer printed
        // In C++, once you've established a base for ints, it remains in
        //  effect until you change it
        cout << hex;
        cout << "A1 i " << i << "\tj " << j << "\tk " << k << endl;
        cout << "A2 i " << i << "\tj " << j << "\tk " << k << endl;
        cout << oct;
        cout << "A3 i " << i << "\tj " << j << "\tk " << k << endl;
        cout << "A4 i " << i << "\tj " << j << "\tk " << k << endl;
        cout << dec;
        cout << "i " << i << endl; cout << "j " << j << endl;
        cout << "k " << k << endl;

        // In C you may specify the width and precision for each float that
        // you want to print, in C++ you may specify the width and precision
        // for each field that you want to print
        float f = 1.234;   float g = 111.23456;  float h = 11111.23;
```

```
// C and C++ both have a default width and precision for floats
printf("f = %f\n",f);  cout << "f = " << f << endl;

printf("f %f %7.2f \t g %.5f %12.6f \t h %f %4.1f\n",f,f,g,g,h,h);
cout << "AAA f " << setw(20) << setprecision(5) << f << endl;
cout << "AAA g " << setw(10) << setprecision(15) << f << endl;
cout << "AAA h " << setw(40) << setprecision(25) << f << endl;

cout << "f " << setw(20) << setprecision(7) << setfill(65) <<
f << endl;
}
```

```
Output From Running Program
coutfmt1.out

i = 13                from first printf and cout
i = 13

i base 10 13  base 8 15  base 16 d
i base 10 13
i base 8  15
i base 16 d

A1 i d  j f     k 7
A2 i d  j f     k 7
A3 i 15 j 17    k 7
A4 i 15 j 17    k 7

i 13
j 15
k 7

f = 1.234
f = 1.234000

AAA f              1.234
AAA g 1.233999967575073
AAA h                1.2339999675750732421875

f 1.234000    1.23      g 111.23456   111.234558       h 11111.230469 11111.2

f AAAAAAAAAAAAAAA1.234

```

```
// coutfmt2.cpp
// this program illustrates some more formatting of output

#include <stdio.h>
#include <iostream.h>       // get access to the cout and cin objects
#include <iomanip.h>        // get access to modifiers for cout cin objects
main()
{

        float f = 1.2;
        float g = 12.345;
        float h = 123;

        // left justify output
        cout << "Left Justify \n";
        cout.unsetf(ios::right);
        cout.unsetf(ios::internal);
        cout.setf(ios::left);
        cout << setw(10) << f
                << setw(10) <<  g
                << setw(10) << h << endl;

        // right justify output
        cout << "\n\nRight Justify \n";
        cout.unsetf(ios::left);
        cout.unsetf(ios::internal);
        cout.setf(ios::right);
        cout << setw(10) << f
                << setw(10) <<  g
                << setw(10) << h << endl;

        // "internal" justify output
        cout << "\n\nInternal Justify \n";
        cout.unsetf(ios::left);
        cout.unsetf(ios::right);
        cout.setf(ios::internal);
        cout << setw(10) << f
                << setw(10) <<  g
                << setw(10) << h << endl;

}
```

```
Output From Running Program
Left Justify
1.2       12.345    123


Right Justify
      1.2    12.345       123


Internal Justify
      1.2    12.345       123

```

```
// coutfmt3.cpp
// this program illustrates some more formatting of output
#include <stdio.h>
#include <iostream.h>       // get access to the cout and cin objects
#include <iomanip.h>        // get access to modifiers for cout cin objects
main()
{
        int i = 13; int j = 15;  int k = 7;

        // these displays will use defaults with respect to whether
        // letters are printed upper or lower case      ios::uppercase
        // signs are added in front of positive numbers ios::showpos
        // any indication of base is given              ios::showbase

        // the dec, oct, hex tells the cout object in what base to prints ints
        cout << "BEFORE SETTING uppercase, showbase, showpos\n";
        cout << "i base 10 " << dec << i << endl;
        cout << "i base 8  " << oct << i << endl;
        cout << "i base 16 " << hex << i << endl;

        // In C++, once you've established a base for ints, it remains in
        //  effect until you change it
        cout << hex;
        cout << "A1 i " << i << "\tj " << j << "\tk " << k << endl;
        cout << "A2 i " << i << "\tj " << j << "\tk " << k << endl;
        cout << oct;
        cout << "A3 i " << i << "\tj " << j << "\tk " << k << endl;
        cout << "A4 i " << i << "\tj " << j << "\tk " << k << endl;
        cout << dec;
        cout << "A5 i " << i << endl; cout << "j " << j << endl;
        cout << "A6 k " << k << endl;

        // NOW I WILL CHANGE THE  uppercase, showpos, showbase, fields
        cout.setf(ios::uppercase);
        cout.setf(ios::showbase);
        cout.setf(ios::showpos);
        cout << "AFTER SETTING uppercase, showbase, showpos\n";
        cout << "i base 10 " << dec << i << endl;
        cout << "i base 8  " << oct << i << endl;
        cout << "i base 16 " << hex << i << endl;

        // In C++, once you've established a base for ints, it remains in  effect until you change it
        cout << hex;
        cout << "B1 i " << i << "\tj " << j << "\tk " << k << endl;
        cout << "B2 i " << i << "\tj " << j << "\tk " << k << endl;
        cout << oct;
        cout << "B3 i " << i << "\tj " << j << "\tk " << k << endl;
        cout << "B4 i " << i << "\tj " << j << "\tk " << k << endl;
        cout << dec;
        cout << "B5 i " << i << endl; cout << "j " << j << endl;
        cout << "B6 k " << k << endl;
}
```

```
Output From Running Program
BEFORE SETTING uppercase, showbase, showpos
i base 10 13
i base 8  15
i base 16 d
A1 i d    j f        k 7
A2 i d    j f        k 7
A3 i 15   j 17       k 7
A4 i 15   j 17       k 7
A5 i 13
j 15
A6 k 7
AFTER SETTING uppercase, showbase, showpos
i base 10 +13
i base 8  015
i base 16 0XD
B1 i 0XD       j 0XF    k 0X7
B2 i 0XD       j 0XF    k 0X7
B3 i 015 j 017      k 07
B4 i 015 j 017      k 07
B5 i +13
j +15
B6 k +7
```

```
// coutfmt4.cpp
// this program illustrates some more formatting of output

#include <stdio.h>
#include <iostream.h>       // get access to the cout and cin objects
#include <iomanip.h>        // get access to modifiers for cout cin objects
main()
{

        float f = 1.2;
        float g = 12.345;
        float h = 123;

        cout << "Default Output \n";
        cout << f << "\t" <<  g << "\t" << h << endl;

        cout << "\nWith width 10 and right justify specified\n";
        cout.unsetf(ios::left);
        cout.setf(ios::right);
        cout << setw(10) << f << "\t" <<
        setw(10) << g << "\t" <<
        setw(10) << h << endl;

        cout << "\nWith width 10 and left justify specified\n";
        cout.unsetf(ios::right);
        cout.setf(ios::left);
        cout << setw(10) << f << "\t" <<
        setw(10) << g << "\t" <<
        setw(10) << h << endl;

        cout << "\nWith showpoint, no width and right justify set\n";
        cout.unsetf(ios::left);
        cout.setf(ios::right);
        cout.setf(ios::showpoint);
        cout << f << "\t" <<  g << "\t" << h << endl;

        cout << "\nWith width 10 and showpoint specified\n";
        cout << setw(10) << f << "\t" <<
        setw(10) << g << "\t" <<
        setw(10) << h << endl;
}
```

```
Output From Running Program
Default Output
1.2    12.345  123

With width 10 and right justify specified
       1.2    12.345         123

With width 10 and left justify specified
1.2       12.345    123

With showpoint, no width and right justify set
1.200000     12.345000     123.000000

With width 10 and showpoint specified
  1.200000   12.345000    123.000000

```

```
// flushit.cpp

// you need to compile and run this program to SEE the differences
// that buffering may or may not make in your system

// this program shows you how to manipulate, by either using the default
// mechanism, forcing buffering on, or forcing buffering off, the buffering
// mechanism that your system wants to use

// Both C and C++ will have their output buffered in certain situations
// You can force the output to be displayed immediately by FLUSHing the
// output buffer.  If you are going to use printf and cout in the same
// program, you need to be very careful to flush after each statement.
// This is because printf and cout have seperate buffers that may fill
// and be dumped by the system at different rates and thus the sequence of
// printfs and couts that you wrote may not display in that sequence

#include <stdio.h>
#include <iostream.h>
#include <dos.h>		// this line is operating system dependant
main()
{
	int i;

	// this loop will use the default buffering that your system uses

	// this loop should produce output of
	// printf cout printf cout printf cout printf cout
	// on some systems it will print
	// printf printf printf printf cout cout cout cout
	// on other systems, it will mix them up in an undetermined way
	// on some systems they will be printed at one second intervals
	// on other systems they won't be displayed until the flushing
	// endl and cr are output

	// the point of this section is that if you are going to do output
	// you have to remember that it is buffered and that unexplained
	// output results can often be explained by examining the buffering
	// mechanism that your compiler and operating system use
	for ( i = 0; i < 4; i++ )
	{
		printf(" printf ");
		cout << " cout ";		// put cout in the output buffer
		sleep(1);		// wait a second
	}
	printf("\n");
	cout << endl;
```

```
	// cout has two functions unsetf and setf which are used to manipulate
	// the state of certain bits that control its behaviour
	// unsetf clears a flag
	// setf   sets   a flag
	// the flag ios::unitbuf, when set, forces all streams to be flushed
	//		immediately after anything is inserted into them
	// the flag ios::stdio, when set, forces stdout and stderr to be
	//		flushed immediately after anything is inserted into them

	cout.unsetf(ios::unitbuf);		// turn off buffer flushing
	cout.unsetf(ios::stdio);		// turn off buffer flushing

	for ( i = 0; i < 4; i++ )
	{
		printf(" printf ");
		cout << " cout ";		// put cout in the output buffer
		sleep(1);		// wait a second
	}
	printf("\n");
	cout << endl << endl;

	// this loop should produce output of
	// printf cout printf cout printf cout printf cout
	// this loop explicitly flushes the buffers that it is writing to
	// the standard output buffer is flushed with fflush
	// the ostream buffer is flushed using the cout reserved word flush
	// a carriage return or an endl would have the same result
	for ( i = 0; i < 4; i++ )
	{
		printf(" printf ");			// put printf in the output buffer
		fflush(stdout);
		cout << " cout " << flush;		// put cout in the output buffer
		sleep(1);			// wait a second
	}
	printf("\n");
	cout << endl;
}
```

```
Output From Running Program To Output File
cout  cout  cout  cout
cout  cout  cout  cout
printf  printf  printf  printf
printf  printf  printf  printf
printf  cout  printf  cout  printf  cout  printf  cout
Screen Dump From Running To Screen
printf cout  printf cout  printf cout  printf cout
printf  printf  printf  printf
cout  cout  cout  cout
printf  cout  printf  cout  printf  cout  printf  cout
```

```
// testws.cpp
// this program demonstrates the difference between doing input with cin
// and the default setup versus doing input with cin with skips specified
#include <iostream.h>
main()
{
	char a,b,c;
	int i,j,k;

	// this line tells the system NOT to ignore white space characters
	// on input
	cin.setf(ios::skipws);

	cout << "Enter three characters with no spaces between them\n";
	cin >> a >> b >> c ;
	cout << "You entered " << a << "  " << b << "  " << c << endl;

	cout << "Enter three characters with some spaces between them\n";
	cin >> a >> b >> c ;
	cout << "You entered " << a << "  " << b << "  " << c << endl;

	cout << "Enter three integers with spaces between them\n";
	cin >> i >> j >> k ;
	cout << "You entered " << i << "  " << j << "  " << k << endl;

	// this line tells the system TO ignore white space characters on input
	cin.unsetf(ios::skipws);

	cout << "Enter three characters with no spaces between them\n";
	cin >> a >> b >> c ;
	cout << "You entered " << a << "  " << b << "  " << c << endl;

	cout << "Enter three characters with some spaces between them\n";
	cin >> a >> b >> c ;
	cout << "You entered " << a << "  " << b << "  " << c << endl;

	cout << "Enter three integers with spaces between them\n";
	cin >> i >> j >> k ;
	cout << "You entered " << i << "  " << j << "  " << k << endl;
}
```

```
Output From Running Program
Enter three characters with no spaces between them
You entered a  b  c
Enter three characters with some spaces between them
You entered a  b  c
Enter three integers with spaces between them
You entered 1  2  3
Enter three characters with no spaces between them
You entered
  a  b
Enter three characters with some spaces between them
You entered c
Enter three integers with spaces between them
You entered 1  2  3
```

```
// ostream1.cpp

// the ostream class provides some useful functions
// flush()         to clear the output buffer
// put()           to print one character
// write()         to print many characters
// tellp()         where is the output pointer?

#include <iostream.h>

char s[] = {"This is a message"};

main()
{
        int i,j,k,l;

        i = cout.tellp();
        cout.put('A');
        cout.put('B');
        cout.put('C');
        cout.put('D');
        cout.flush();

        j = cout.tellp();
        cout.write("XXXX",4);
        cout.flush();

        k = cout.tellp();
        cout.write(s,sizeof(s) );
        l = cout.tellp();
        cout.flush();

        cout << endl << endl;
        cout << "Before ABCD The output stream pointer was at "
                << i << endl;
        cout << "Before XXX The output stream pointer was at "
                << j << endl;
        cout << "Before s the output stream pointer was at "
                << k << endl;
        cout << "After  s the output stream pointer was at "
                << l << endl;
}
```

```
Output From Running Program

ABCDXXXXThis is a message

Before ABCD The output stream pointer was at 0
Before XXX The output stream pointer was at 4
Before s the output stream pointer was at 8
After  s the output stream pointer was at 26
ABCDXXXXThis is a message

Before ABCD The output stream pointer was at 216
Before XXX The output stream pointer was at 220
Before s the output stream pointer was at 224
After  s the output stream pointer was at 242
ABCDXXXXThis is a message

Before ABCD The output stream pointer was at 439
Before XXX The output stream pointer was at 443
Before s the output stream pointer was at 447
After  s the output stream pointer was at 465
ABCDXXXXThis is a message

Before ABCD The output stream pointer was at 662
Before XXX The output stream pointer was at 666
Before s the output stream pointer was at 670
After  s the output stream pointer was at 688
ABCDXXXXThis is a message

Before ABCD The output stream pointer was at 885
Before XXX The output stream pointer was at 889
Before s the output stream pointer was at 893
After  s the output stream pointer was at 911
```

```
// ostream2.cpp
// the ostream has a number of flags that control it
// they can be read as a group
// they can be queried individually to return a true/false answer
#include <iostream.h>

main()
{
        long int first_flags  = 0;
        long int second_flags = 0;
        int i;
        long int j = 0;
        long int k = 0;
        char c;

        first_flags = cout.flags();
        cout << "The default value of the flags for cout is: ";
        cout << hex << first_flags << endl;

        cout.setf(ios::hex);
        cout.setf(ios::showbase);

        cout << "Bit for skipws is     \t" << ios::skipws << endl;
        cout << "Bit for left   is     \t" << ios::left   << endl;
        cout << "Bit for right  is     \t" << ios::right  << endl;
        cout << "Bit for internal is   \t" << ios::internal << endl;
        cout << "Bit for dec is        \t" << ios::dec << endl;
        cout << "Bit for oct is        \t" << ios::oct << endl;
        cout << "Bit for hex is        \t" << ios::hex << endl;
        cout << "Bit for showbase is   \t" << ios::showbase << endl;
        cout << "Bit for showpoint is  \t" << ios::showpoint << endl;
        cout << "Bit for scientific is \t" << ios::scientific << endl;
        cout << "Bit for fixed is      \t" << ios::fixed << endl;
        cout << "Bit for unitbuf is    \t" << ios::unitbuf << endl;
        cout << "Bit for stdio is      \t" << ios::stdio << endl;

        second_flags = cout.flags();
        cout << "The value of the flags for cout is: ";
        cout << hex << second_flags << endl;

        j = cout.width();
        cout << "The value of the width for cout is: ";
        cout << dec << j << endl;

        k = cout.precision();
        cout << "The value of the precision for cout is: ";
        cout << dec << k << endl;

        c = cout.fill();
        i = (int) c;
        cout << "The value of the fill for cout is: ";
        cout << hex << i << endl;
}
```

```
Output From Running Program
The default value of the flags for cout is: 1
Bit for skipws is          0x1
Bit for left   is     0x2
Bit for right  is     0x4
Bit for internal is        0x8
Bit for dec is        0x10
Bit for oct is        0x20
Bit for hex is             0x40
Bit for showbase is        0x80
Bit for showpoint is       0x100
Bit for scientific is      0x800
Bit for fixed is           0x1000
Bit for unitbuf is         0x2000
Bit for stdio is           0x4000
The value of the flags for cout is: 0xc1
The value of the width for cout is: 0
The value of the precision for cout is: 0
The value of the fill for cout is: 0x20
```

```cpp
// ostream3.cpp
// the ostream has a number of flags that report its status
// they can be read as a group
// they can be queried individually to return a true/false answer

#include <iostream.h>

main()
{
	long int first_state  = 0;
	int i,j,k,l,m,n;

	first_state = cout.rdstate();
	cout << "The default value of the state for cout is: ";
	cout << hex << first_state << endl;

	cout.setf(ios::hex);
	cout.setf(ios::showbase);

	cout << "Bit for goodbit  is     \t" << ios::goodbit  << endl;
	cout << "Bit for eofbit   is     \t" << ios::eofbit   << endl;
	cout << "Bit for failbit  is     \t" << ios::failbit  << endl;
	cout << "Bit for badbit   is     \t" << ios::badbit   << endl;
	cout << "Bit for hardfail is     \t" << ios::hardfail << endl;

	j = cout.good();
	cout << "The value of good for cout is: ";
	cout << dec << j << endl;

	k = cout.eof();
	cout << "The value of eof for cout is: ";
	cout << dec << k << endl;

	l = cout.fail();
	cout << "The value of fail for cout is: ";
	cout << dec << l << endl;

	m = cout.bad();
	cout << "The value of bad for cout is: ";
	cout << dec << m << endl;

	// if the ostream gets into a bad state, you can attempt to
	// clear it using
	cout.clear();
}
```

```
Output From Running Program

The default value of the state for cout is: 0
Bit for goodbit  is          0x0
Bit for eofbit   is          0x1
Bit for failbit  is          0x2
Bit for badbit   is          0x4
Bit for hardfail is          0x80
The value of good for cout is: 1
The value of eof for cout is: 0
The value of fail for cout is: 0
The value of bad for cout is: 0

```

```
// istream1.cpp
// the istream class provides some useful functions
// get          to extract a single character
// peek         to look at next char without extracting
// putback      to put character back into istream
// read         to get a specified number of characters from stream
// getline      to get up to a specified number of characters or
//              to get up until a specified line terminator is encountered
// seekg        to move the get pointer in the istream
// tellg        to tell where the get pointer is in the istream

#include <iostream.h>
#include <stdio.h>
#include <string.h>

main()
{
        long int i,j,k,l,m,n,o;
        char c;
        char s[100];

        i = cin.tellg();            // find out where the pointer is
        s[0] = NULL;                // make sure s has nothing in it

        cout << "Enter a character and press return please ";
        cin >> c;           // read a character using >> operator
        cout << "You entered " << c << endl;

        j = cin.tellg();            // find out where the pointer is
        s[0] = NULL;                // make sure s has nothing in it

        cout << "Enter a character and press return please ";
        c = cin.get();              // read a character using get operator
        cout << "You entered " << c << endl;

        // in this case we ask for a carriage return terminated string
        // we use the cin operator >> which knows how to read into a
        // character pointer
        s[0] = NULL;
        k = cin.tellg();
        cout << "Enter a cr terminated string please ";
        cin >> s;
        cout << "You entered " << s << endl;

        // in this case we ask for five characters,
        // the system has been programmed to look for ten characters
        // the system keeps reading until it sees ten characters
        // even the carriage return does not terminate the string
        s[0] = NULL;
        l = cin.tellg();
        cout << "Enter a 5 character, cr terminated string please ";
        cin.read(s,10);
        s[10] = NULL;   // stop s from running away
        cout << "You entered " << s << endl;

```

```
// in this case we ask for fifteen characters,
// the system has been programmed to look for ten characters
// the system keeps reading until it sees ten characters
// once it sees 10 characters it stops
s[0] = NULL;
m = cin.tellg();
cout << "Enter a 15 character, cr terminated string please ";
cin.read(s,10);
s[10] = NULL;   // stop s from running away
cout << "You entered " << s << endl;

// in this case we are asking for a five character string that
// is to be terminated by an X
// the system has been programmed to look for up to 10 chars
// if the system sees ten characters, or an X, it stops reading
s[0] = NULL;
n = cin.tellg();
cout << "Enter a 5 character, X terminated string please ";
cin.getline(s,10,'X');
cout << "You entered " << s << endl;

s[0] = NULL;
o = cin.tellg();
cout << "Enter a 15 character, X terminated string please ";
cin.getline(s,10,'X');
cout << "You entered " << s << endl;

cout << endl << endl;
cout << "The input stream pointer was at " << i << endl;
cout << "The input stream pointer was at " << j << endl;
cout << "The input stream pointer was at " << k << endl;
cout << "The input stream pointer was at " << l << endl;
cout << "The input stream pointer was at " << m << endl;
cout << "The input stream pointer was at " << n << endl;
cout << "The input stream pointer was at " << o << endl;

}
```

```
Output From Running Program
Enter a character and press return please You entered a
Enter a character and press return please You entered e
Enter a cr terminated string please You entered 234567890123
Enter a 5 character, cr terminated string please You entered 4567890123
Enter a 15 character, cr terminated string please You entered
abcdeX
fg
Enter a 5 character, X terminated string please You entered jkX
Enter a 15 character, X terminated string please You entered lmnopqrst


The input stream pointer was at 0
The input stream pointer was at 10
The input stream pointer was at 17
The input stream pointer was at 34
The input stream pointer was at 48
The input stream pointer was at 62
The input stream pointer was at 67

```

```
// istream2.cpp
// the istream has a number of flags that control it
// they can be read as a group
// they can be queried individually to return a true/false answer

#include <iostream.h>

main()
{
	long int first_flags  = 0;
	int i;
	char c;

	first_flags = cin.flags();
	cout << "The default value of the flags for cin is: ";
	cout << hex << first_flags << endl;

	cout << "Bit for skipws is     \t" << ios::skipws << endl;
	cout << "Bit for left   is     \t" << ios::left   << endl;
	cout << "Bit for right  is     \t" << ios::right  << endl;
	cout << "Bit for internal is   \t" << ios::internal << endl;
	cout << "Bit for dec is        \t" << ios::dec << endl;
	cout << "Bit for oct is        \t" << ios::oct << endl;
	cout << "Bit for hex is        \t" << ios::hex << endl;
	cout << "Bit for showbase is   \t" << ios::showbase << endl;
	cout << "Bit for showpoint is  \t" << ios::showpoint << endl;
	cout << "Bit for scientific is \t" << ios::scientific << endl;
	cout << "Bit for fixed is      \t" << ios::fixed << endl;
	cout << "Bit for unitbuf is    \t" << ios::unitbuf << endl;
	cout << "Bit for stdio is      \t" << ios::stdio << endl;

}
```

```
Output From Running Program
The default value of the flags for cin is: 1
Bit for skipws is           1
Bit for left   is           2
Bit for right  is      4
Bit for internal is         8
Bit for dec is        10
Bit for oct is              20
Bit for hex is              40
Bit for showbase is         80
Bit for showpoint is        100
Bit for scientific is       800
Bit for fixed is            1000
Bit for unitbuf is          2000
Bit for stdio is            4000
```

```
// istream3.cpp
// the istream has a number of flags that report its status
// they can be read as a group
// they can be queried individually to return a true/false answer

#include <iostream.h>

main()
{
	long int first_state  = 0;
	int i,j,k,l,m,n;

	first_state = cin.rdstate();
	cout << "The default value of the state for cin is: ";
	cout << hex << first_state << endl;

	cout << "Bit for goodbit  is     \t" << ios::goodbit  << endl;
	cout << "Bit for eofbit   is     \t" << ios::eofbit   << endl;
	cout << "Bit for failbit  is     \t" << ios::failbit  << endl;
	cout << "Bit for badbit   is     \t" << ios::badbit   << endl;
	cout << "Bit for hardfail is     \t" << ios::hardfail << endl;

	j = cin.good();
	cout << "The value of good for cin is: ";
	cout << dec << j << endl;

	k = cin.eof();
	cout << "The value of eof for cin is: ";
	cout << dec << k << endl;

	l = cin.fail();
	cout << "The value of fail for cin is: ";
	cout << dec << l << endl;

	m = cin.bad();
	cout << "The value of bad for cin is: ";
	cout << dec << m << endl;

	// if the ostream gets into a bad state, you can attempt to
	// clear it using
	cin.clear();
}
```

```
Output From Running Program
The default value of the state for cin is: 0
Bit for goodbit  is              0
Bit for eofbit   is              1
Bit for failbit  is              2
Bit for badbit   is              4
Bit for hardfail is              80
The value of good for cin is: 1
The value of eof for cin is: 0
The value of fail for cin is: 0
The value of bad for cin is: 0
```

```
// finfout.cpp

// fstreams inherit from iostreams
// ifstream inherits from istream
// ofstream inherits from ostream

// the streams have a collection of bits that describe their current state
// the streams have a collection of functions to help manipulate them
// the streams have a number of enumerated values that control how they
// are opened.
// subsequent programs, finfout1.cpp finfout2.cpp finfout3.cpp will explore
// these bits, functions, and open mechanisms

// this program simply opens a file and reads it

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

main()
{
	// open the data file for reading
	ifstream in("finfout.dat");		// use default constructor
	if ( in.good() )
	{
		cout << "File open succeeded\n";
	}
	else
	{
		cout << "File open failed\n";
		exit(-1);
	}

	char c;
	in >> c;
	while ( in.eof() == 0 )
	{
		cout << c;
		in >> c;
	}

	// now that we have generated an EOF
	// look at the flags and try to clear the bad state
	int i = in.rdstate();
	cout << "\n the EOF but before the clear\n";
	cout << "rdstate yielded " << i << endl;
	i = in.good();
	cout << "good yielded " << i << endl;
	i = in.bad();
	cout << "bad yielded " << i << endl;
	i = in.eof();
	cout << "eof yielded " << i << endl;
	i = in.fail();
	cout << "fail yielded " << i << endl;
```

```
        // now try to clear the bad state
        in.clear();
        i = in.rdstate();
        cout << "After the EOF and after the clear\n";
        cout << "rdstate yielded " << i << endl;
        i = in.good();
        cout << "good yielded " << i << endl;
        i = in.bad();
        cout << "bad yielded " << i << endl;
        i = in.eof();
        cout << "eof yielded " << i << endl;
        i = in.fail();
        cout << "fail yielded " << i << endl;

        // be nice and clean up after myself
        in.close();
}
```

```
Output From Running Program
File open succeeded
1234567890abcdefghih
 the EOF but before the clear
rdstate yielded 1
good yielded 0
bad yielded 0
eof yielded 1
fail yielded 0
After the EOF and after the clear
rdstate yielded 0
good yielded 1
bad yielded 0
eof yielded 0
fail yielded 0
```

```
// finfout1.cpp
// this program opens a file for reading
// queries you for the name of the file you want to create and writes
// the file opened for reading into the file opened for writing

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

main()
{
	// open the data file for reading
	ifstream in("finfout.dat");		// use default constructor
	if ( in.good() )
	{
		cout << "File open succeeded\n";
	}
	else
	{
		cout << "File open failed\n";
		exit(-1);
	}

	// open the data file for writing
	char out_name[100];	// should handle most file names
	cout << "Enter file name to copy to \n";
	cin >> out_name;
	cout << "You entered " << out_name << endl;

	ofstream out(out_name);		// automatically opened for writing
	if ( out.good() )
	{
		cout << "Output File open succeeded\n";
	}
	else
	{
		cout << "Output File open failed\n";
		exit(-2);
	}

	char c;
	in >> c;
	while ( in.eof() == 0 )
	{
		cout << c;	// write to the screen
		out << c;	// write to the file
		in >> c;
	}
	// be nice and clean up after myself
	in.close();
	out.close();
}
```

```
Output From Running Program
File open succeeded
Enter file name to copy to
You entered copyto1
Output File open succeeded
1234567890abcdefghih
```

```
// finfout2.cpp
// this program opens a file for reading and writing
// it reads every other character from the file
#include <iostream.h>
#include <fstream.h>
#include <stdio.h>
#include <stdlib.h>

// this program uses
// seekg(number,position)
// seekp(number,position)

// the number may be positive or negative
// positive  => move forward  number characters from position
// negative  => move backward number characters from position

// position may be
// ios::beg          beginning of file
// ios::cur          current position in file
// ios::end          ending of file


// this program also uses
// tellg()           report the current position of get pointer
// tellp()           report the current position of put pointer
main()
{
        char d;

        // open the data file for reading and writing
        // uses the two argument constructor
        fstream inout("finfout2.dat", ios::in | ios::out | ios::nocreate );
        if ( inout.good() )
        {
                cout << "File open succeeded\n";
        }
        else
        {
                cout << "File open failed\n";
                exit(-1);
        }

        char c;

        // find out where the end of the file is
        inout.seekp(0,ios::end);                // move to end of file
        int lastpos = inout.tellp();            // find out where end is
        cout << "Last position in file is " << lastpos << endl;

        int jtgetpos = 0;
        inout.seekg(jtgetpos,ios::beg);         // position the input pointer
        int curgetpos = inout.tellg();
        cout << "Initial get position is " << curgetpos << endl;
        inout >> c;
        cout << "Read " << c << " from initial position \n";
```

```
	while ( inout.eof() == 0 )
	{
		jtgetpos += 2;
		inout.seekg(jtgetpos,ios::beg);		// position the input pointer
		curgetpos = inout.tellg();
		cout << "New get position is " << curgetpos << endl;
		inout >> c;
		cout << "Read " << c << " from new position \n";
	}

	// be nice and clean up after myself
	inout.close();
}
```

Output From Running Program

```
File open succeeded
Last position in file is 14
Initial get position is 0
Read a from initial position
New get position is 2
Read c from new position
New get position is 4
Read e from new position
New get position is 6
Read g from new position
New get position is 8
Read i from new position
New get position is 10
Read i from new position
```

```
// finfout3.cpp
// this program opens a file for reading and writing
// it reads every other character from the file and overwrites the
// next character in the file
// for example, an input file of
// abcdefghij
//			would be changed to
// aacceeggii

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

// this program uses
// seekg(number,position)
// seekp(number,position)

// the number may be positive or negative
// positive  => move forward  number characters from position
// negative  => move backward number characters from position

// position may be
// ios::beg		beginning of file
// ios::cur		current position in file
// ios::end		ending of file

// this program also uses
// tellg()		report the current position of get pointer
// tellp()		report the current position of put pointer
main()
{
	char d;

	// open the data file for reading and writing
	// uses the two argument constructor
	fstream inout("finfout2.dat", ios::in | ios::out | ios::nocreate );
	if ( inout.good() )
	{
		cout << "File open succeeded\n";
	}
	else
	{
		cout << "File open failed\n";
		exit(-1);
	}

	char c;

	// find out where the end of the file is
	inout.seekp(0,ios::end);	// move to end of file
	int lastpos = inout.tellp();	// find out where end is
	cout << "Last position in file is " << lastpos << endl;

	int jtgetpos = 0;
```

```
        inout.seekg(jtgetpos,ios::beg);   // position the input pointer
        int curgetpos = inout.tellg();
        cout << "Initial get position is " << curgetpos << endl;
        inout >> c;
        cout << "Read " << c << " from initial position \n";

        while ( inout.eof() == 0 )
        {
                jtgetpos += 2;
                inout.seekp(jtgetpos - 1, ios::beg);
                inout.put(c);

                inout.seekg(jtgetpos,ios::beg);   // position the input pointer
                curgetpos = inout.tellg();
                cout << "New get position is " << curgetpos << endl;
                inout >> c;
                cout << "Read " << c << " from new position \n";
        }

        // be nice and clean up after myself
        inout.close();
}
```

Output From Running Program

```
File open succeeded
Last position in file is 14
Initial get position is 0
Read a from initial position
New get position is 2
Read c from new position
New get position is 4
Read e from new position
New get position is 6
Read g from new position
New get position is 8
Read i from new position
New get position is 10
Read i from new position
```

```
// ostristr.cpp
// ostreams and istreams can be checked to be good or not
// without using the function call cout.good() or cin.good()

#include <iostream.h>

main()
{
	char c;

	cout << "Enter a letter and then press return\n";
	cin >> c;
	cout << "You entered " << c << endl;

	if ( cout )
	{
		cout << "Cout is still in a good state\n";
	}

	if ( cin )
	{
		cout << "Cin is still in a good state \n";
	}
}
```

Output From Running Program

```
Enter a letter and then press return
You entered a
Cout is still in a good state
Cin is still in a good state
```

```
// predefs.cpp
// C++ provides four predefined streams for you
// cout		default output stream (stdout)
// cin		default input stream  (stdin)
// cerr		default error stream  (stderr)
// clog

#include <iostream.h>

main()
{
	char c;
	clog << "First Message sent to clog \n";
	cout << "Message sent to cout \n";
	clog << "Second Message sent to clog \n";
	cerr << "Message sent to cerr \n";
	clog << "Third Message sent to clog \n";

	cout << "Enter a letter and press return \n";
	cin >> c;
	cout << "You entered " << c << endl;

	clog << "Fourth Message sent to clog \n";
}


Output From Running Program
Message sent to cout
Enter a letter and press return
You entered b
```

```
// filebit1.cpp
// the fstreams have a number of flags that control it
// they can be read as a group
// they can be queried individually to return a true/false answer

#include <iostream.h>
#include <fstream.h>

main()
{
	long int first_flags  = 0;
	int i;
	char c;

	ifstream in("filebits.in");    // use default constructor
	ofstream out("filebits.out");		// use default constructor

	first_flags = in.flags();
	cout << "The default value of the flags for in is: ";
	cout << hex << first_flags << endl << endl;

	first_flags = out.flags();
	cout << "The default value of the flags for out is: ";
	cout << hex << first_flags << endl << endl;

	cout << "Bit for skipws is     \t" << ios::skipws << endl;
	cout << "Bit for left   is     \t" << ios::left   << endl;
	cout << "Bit for right  is     \t" << ios::right  << endl;
	cout << "Bit for internal is   \t" << ios::internal << endl;
	cout << "Bit for dec is        \t" << ios::dec << endl;
	cout << "Bit for oct is        \t" << ios::oct << endl;
	cout << "Bit for hex is        \t" << ios::hex << endl;
	cout << "Bit for showbase is   \t" << ios::showbase << endl;
	cout << "Bit for showpoint is  \t" << ios::showpoint << endl;
	cout << "Bit for scientific is \t" << ios::scientific << endl;
	cout << "Bit for fixed is      \t" << ios::fixed << endl;
	cout << "Bit for unitbuf is    \t" << ios::unitbuf << endl;
	cout << "Bit for stdio is      \t" << ios::stdio << endl;

}
```

```
Output From Running Program
The default value of the flags for in is: 1

The default value of the flags for out is: 1

Bit for skipws is               1
Bit for left   is       2
Bit for right  is       4
Bit for internal is             8
Bit for dec is          10
Bit for oct is          20
Bit for hex is                  40
Bit for showbase is             80
Bit for showpoint is            100
Bit for scientific is           800
Bit for fixed is                1000
Bit for unitbuf is              2000
Bit for stdio is                4000

```

```
// filebit2.cpp
#include <iostream.h>
#include <fstream.h>
main()
{
	long int first_state  = 0;
	int i,j,k,l,m,n;

	ifstream in("filebit2.in");
	first_state = in.rdstate();
	cout << "The default value of the state for in is: ";
	cout << hex << first_state << endl;

	cout << "Bit for goodbit  is     \t" << ios::goodbit  << endl;
	cout << "Bit for eofbit   is     \t" << ios::eofbit   << endl;
	cout << "Bit for failbit  is     \t" << ios::failbit  << endl;
	cout << "Bit for badbit   is     \t" << ios::badbit   << endl;
	cout << "Bit for hardfail is     \t" << ios::hardfail << endl;

	j = in.good();
	cout << "The value of good for in is: ";
	cout << dec << j << endl;

	k = in.eof();
	cout << "The value of eof for in is: ";
	cout << dec << k << endl;

	l = in.fail();
	cout << "The value of fail for in is: ";
	cout << dec << l << endl;

	m = in.bad();
	cout << "The value of bad for in is: ";
	cout << dec << m << endl;

	// if the ostream gets into a bad state, you can attempt to
	// clear it using
	in.clear();

	ofstream out("filebit2.out");
	first_state = out.rdstate();
	cout << "The default value of the state for out is: ";
	cout << hex << first_state << endl;

	cout << "Bit for goodbit  is     \t" << ios::goodbit  << endl;
	cout << "Bit for eofbit   is     \t" << ios::eofbit   << endl;
	cout << "Bit for failbit  is     \t" << ios::failbit  << endl;
	cout << "Bit for badbit   is     \t" << ios::badbit   << endl;
	cout << "Bit for hardfail is     \t" << ios::hardfail << endl;

	j = out.good();
	cout << "The value of good for out is: ";
	cout << dec << j << endl;

	k = out.eof();
```

```
        cout << "The value of eof for out is: ";
        cout << dec << k << endl;

        l = out.fail();
        cout << "The value of fail for out is: ";
        cout << dec << l << endl;

        m = out.bad();
        cout << "The value of bad for out is: ";
        cout << dec << m << endl;

        // if the ostream gets into a bad state, you can attempt to
        // clear it using
        out.clear();

}
```

Output From Running Program

```
The default value of the state for in is: 0
Bit for goodbit  is         0
Bit for eofbit   is         1
Bit for failbit  is         2
Bit for badbit   is         4
Bit for hardfail is         80
The value of good for in is: 1
The value of eof for in is: 0
The value of fail for in is: 0
The value of bad for in is: 0
The default value of the state for out is: 0
Bit for goodbit  is         0
Bit for eofbit   is         1
Bit for failbit  is         2
Bit for badbit   is         4
Bit for hardfail is         80
The value of good for out is: 1
The value of eof for out is: 0
The value of fail for out is: 0
The value of bad for out is: 0
```

```
// fileop1.cpp
// C++ allows you to work with files
// C++ can work with C files just the way you already know them
// I recommend staying with the C file routines because they are
// widely and correctly implemented

#include <stdio.h>
#include <iostream.h>
#include <fstream.h>    // get access to the file routines

// ifstream inherits from istream
// ofstream inherits from ostream

// There are several bits that control how a file is opened
// they are enumerated in iostream.h

//      in              open for reading
//      out             open for writing

//      ate             seek to eof on original open

//      app             append, seek to eof on open

//      trunc           delete file if it already exists
//      nocreate        open fails if file doesn't already exist
//      noreplace       open fails if file already does exist

//      binary          open file in binary mode, not a text file

// there are several functions for querying the status of a file
// rdstate()
// eof()
// fail()
// bad()
// good()

// if these calls look familiar, they are, remember that ifstream and ofstream
// inherit from istream and ostream
main()
{
        char c;

        ofstream out("fileopen.dat");           // open for output, use default constructor
        out << "This message came from fileopen.cpp \n";
        out.close();                            // close the file

        ifstream in("fileopen.dat");            // open for input, use default constructor
        in >> c;        // get character from file
        while ( in.eof() == 0 )
        {
                cout << c;      // print that character on  screen
                in >> c;        // get the next character
        }
        in.close();
        cout << endl << endl;
```

```

        // now look at the difference that skipws will have when we use
        // the exact same code to read from the file
        ifstream wsin("fileopen.dat");       // open for input, use default constructor
        wsin.unsetf(ios::skipws);   // DO NOT SKIP WS ON INPUT

        wsin >> c;                  // get character from file
        while ( wsin.eof() == 0 )
        {
                cout << c;          // print that character on  screen
                wsin >> c;          // get the next character
        }
        // now that we have encountered an EOF, let's see what that did to
        // the other calls available
        int after = wsin.rdstate();
        int eof_stat = wsin.eof();
        int fail_stat = wsin.fail();
        int bad_stat  = wsin.bad();
        int good_stat = wsin.good();

        cout << "\n\nBefore clear flags for wsin are " << after << endl;
        cout << "eof stat for wsin is " << eof_stat << endl;
        cout << "fail stat for wsin is " << fail_stat << endl;
        cout << "bad stat for wsin is " << bad_stat << endl;
        cout << "good stat for wsin is " << good_stat << endl;

        // now try to clear the stream using clear and then
        // look at the bits again
        wsin.clear();
        after = wsin.rdstate();
        eof_stat = wsin.eof();
        fail_stat = wsin.fail();
        bad_stat  = wsin.bad();
        good_stat = wsin.good();

        cout << "\n\nAfter clear flags for wsin are " << after << endl;
        cout << "eof stat for wsin is " << eof_stat << endl;
        cout << "fail stat for wsin is " << fail_stat << endl;
        cout << "bad stat for wsin is " << bad_stat << endl;
        cout << "good stat for wsin is " << good_stat << endl;

        wsin.close();
        cout << endl << endl;
}
```

```
Output From Running Program
Thismessagecamefromfileopen.cpp

This message came from fileopen.cpp


Before clear flags for wsin are 3
eof stat for wsin is 1
fail stat for wsin is 2
bad stat for wsin is 0
good stat for wsin is 0


After clear flags for wsin are 0
eof stat for wsin is 0
fail stat for wsin is 0
bad stat for wsin is 0
good stat for wsin is 1
```

```
// fileop2.cpp
// this program illustrates the different ways files can be opened

#include <stdio.h>
#include <iostream.h>
#include <fstream.h>        // get access to the file routines

// There are several bits that control how a file is opened
// they are enumerated in iostream.h

//          in                  open for reading
//          out                 open for writing

//          ate                 seek to eof on original open

//          app                 append, seek to eof on open

//          trunc               delete file if it already exists
//          nocreate            open fails if file doesn't already exist
//          noreplace           open fails if file already does exist

//          binary              open file in binary mode, not a text file

main()
{
            int rdstat,eof_stat,fail_stat,bad_stat,good_stat;

            // open for reading
            ifstream in1("fileop2.in", ios::in );

            rdstat    = in1.rdstate();
            eof_stat  = in1.eof();
            fail_stat = in1.fail();
            bad_stat  = in1.bad();
            good_stat = in1.good();

            cout << "Status bits for in1 are \n";
            cout << "rdstat for is " << rdstat << endl;
            cout << "eof stat   is " << eof_stat << endl;
            cout << "fail stat  is " << fail_stat << endl;
            cout << "bad stat   is " << bad_stat << endl;
            cout << "good stat  is " << good_stat << endl;

            in1.close();

            // open for reading, and writing
            ifstream inout1("fileop2.in", ios::in  | ios::out );

            rdstat    = inout1.rdstate();
            eof_stat  = inout1.eof();
            fail_stat = inout1.fail();
            bad_stat  = inout1.bad();
            good_stat = inout1.good();

            cout << "Status bits for inout1 are \n";
```

```
        cout << "rdstat for is " << rdstat << endl;
        cout << "eof stat   is " << eof_stat << endl;
        cout << "fail stat  is " << fail_stat << endl;
        cout << "bad stat   is " << bad_stat << endl;
        cout << "good stat  is " << good_stat << endl;

        inout1.close();

        // open for writing
        ifstream out1("fileop2.out", ios::out );

        rdstat    = out1.rdstate();
        eof_stat  = out1.eof();
        fail_stat = out1.fail();
        bad_stat  = out1.bad();
        good_stat = out1.good();

        cout << "Status bits for out1 are \n";
        cout << "rdstat for is " << rdstat << endl;
        cout << "eof stat   is " << eof_stat << endl;
        cout << "fail stat  is " << fail_stat << endl;
        cout << "bad stat   is " << bad_stat << endl;
        cout << "good stat  is " << good_stat << endl;

        out1.close();
}
```

```
Output From Running Program
Status bits for in1 are
rdstat for is 0
eof stat   is 0
fail stat  is 0
bad stat   is 0
good stat  is 1
Status bits for inout1 are
rdstat for is 0
eof stat   is 0
fail stat  is 0
bad stat   is 0
good stat  is 1
Status bits for out1 are
rdstat for is 0
eof stat   is 0
fail stat  is 0
bad stat   is 0
good stat  is 1
```

```
// fileop3.cpp
// this program illustrates more different ways files can be opened

#include <stdio.h>
#include <iostream.h>
#include <fstream.h>    // get access to the file routines

// There are several bits that control how a file is opened
// they are enumerated in iostream.h

//          in              open for reading
//          out             open for writing

//          ate             seek to eof on original open

//          app             append, seek to eof on open

//          trunc           delete file if it already exists
//          nocreate        open fails if file doesn't already exist
//          noreplace       open fails if file already does exist

//          binary          open file in binary mode, not a text file

main()
{
        int rdstat,eof_stat,fail_stat,bad_stat,good_stat;

        // open for reading, open fails if file doesn't exist
        ifstream in1("fileop3.in1", ios::in | ios::nocreate );

        rdstat    = in1.rdstate();
        eof_stat  = in1.eof();
        fail_stat = in1.fail();
        bad_stat  = in1.bad();
        good_stat = in1.good();

        cout << "Status bits for in1 are \n";
        cout << "rdstat for is " << rdstat << endl;
        cout << "eof stat   is " << eof_stat << endl;
        cout << "fail stat  is " << fail_stat << endl;
        cout << "bad stat   is " << bad_stat << endl;
        cout << "good stat  is " << good_stat << endl;

        in1.close();

        // open for reading, fails if file already exists
        // DOES THIS ONE MAKE SENSE?
        ifstream in2("fileop3.in2", ios::in | ios::noreplace );

        rdstat    = in2.rdstate();
        eof_stat  = in2.eof();
        fail_stat = in2.fail();
        bad_stat  = in2.bad();
        good_stat = in2.good();

```

```
	cout << "Status bits for in2 are \n";
	cout << "rdstat for is " << rdstat << endl;
	cout << "eof stat   is " << eof_stat << endl;
	cout << "fail stat  is " << fail_stat << endl;
	cout << "bad stat   is " << bad_stat << endl;
	cout << "good stat  is " << good_stat << endl;

	in2.close();

	// open for reading, delete if file already exists
	// DOES THIS ONE MAKE SENSE?
	ifstream in3("fileop2.in3", ios::in | ios::trunc );

	rdstat    = in3.rdstate();
	eof_stat  = in3.eof();
	fail_stat = in3.fail();
	bad_stat  = in3.bad();
	good_stat = in3.good();

	cout << "Status bits for in3 are \n";
	cout << "rdstat for is " << rdstat << endl;
	cout << "eof stat   is " << eof_stat << endl;
	cout << "fail stat  is " << fail_stat << endl;
	cout << "bad stat   is " << bad_stat << endl;
	cout << "good stat  is " << good_stat << endl;

	in3.close();

}
```

```
Output From Running Program
Status bits for in1 are
rdstat for is 4
eof stat   is 0
fail stat  is 4
bad stat   is 4
good stat  is 0
Status bits for in2 are
rdstat for is 4
eof stat   is 0
fail stat  is 4
bad stat   is 4
good stat  is 0
Status bits for in3 are
rdstat for is 4
eof stat   is 0
fail stat  is 4
bad stat   is 4
good stat  is 0
```

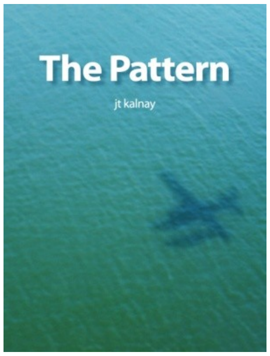
1

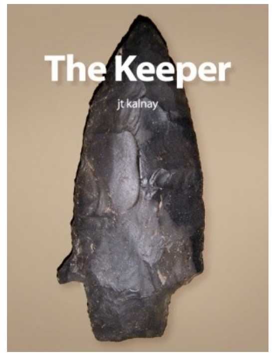
After JT Decided He Couldn't Write One More Program....

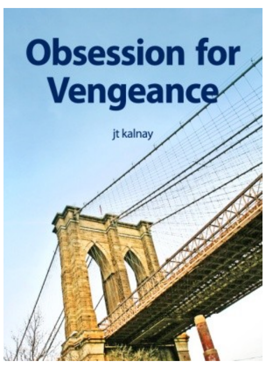
He Started Writing Novels,

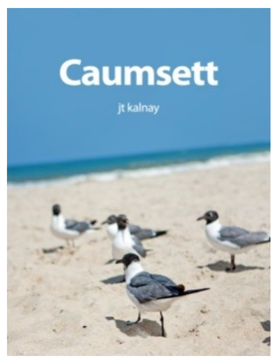
Please Consider Reading The Pattern (it's about a programming error)

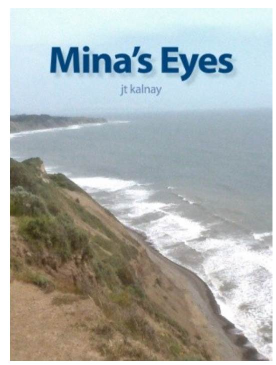
Or Consider Sampling Others of JT Kalnay's Novels

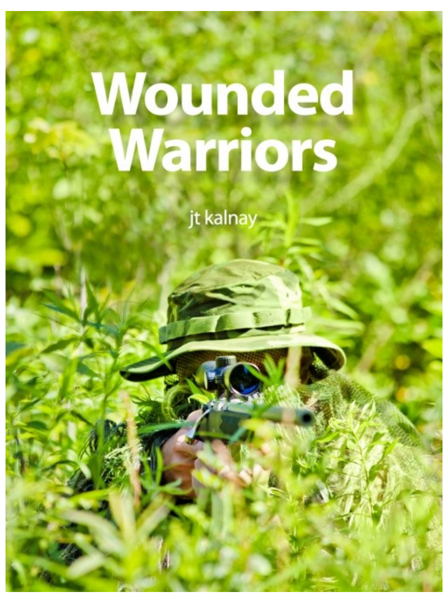












Reviews for JT’s Programming Technothriller The Pattern

The first of JT Kalnay's works I've read, this early effort compares nicely with Ryan's "Adolescence of P-1" or Grisham's "The Firm" but wisely navigates around Powers' "Galatea 2.2" territory. You get a good sense this writer has "been there" but there is more to "The Pattern" than just an insider's view of an industry and culture that is pretty much a black box to those that haven't. This one gets a 4 out of 5 simply for not quite cracking the level of the big boys: Clancy, Ludlum, Cussler et al. Will be interested to see how this author develops in this genre.

I was surprised to enjoy this book so much as it comes from a not so well known author. Fantastic fiction.

I was thinking about the HAL 9000 malfunction in 2001 A Space Odyssey while reading The Pattern. Decades ago, I wondered if people would risk their lives on software. Now we have fly-by-wire controls in our airplanes and we depend on software in our hospital equipment as well as our cars. Software glitches can now kill. It's a really scary thought and I really enjoyed the thrilling journey the author takes us on in this techno-thriller treat. In the best spirit of science fiction it gives us pause to consider the dependency we freely give to our technology. In addition, as this story unfolds our humanity is laid bare in the face of technological realities that are seldom realized by most of us.

Please Enjoy This Sample From The Pattern

June 19, 1994

Chantilly Virginia

Assembled From News Wire Reports

A chartered executive Lear Jet inbound from Mexico City crashed today in heavy fog during final approach to Dulles National Airport in Washington D.C. Ten passengers and two crew members were killed instantly. There were no Americans on the flight and there were no survivors. Although the airplane had the latest electronics, it had aborted one landing due to the fog and was in the process of lining up for a second attempt when the accident occurred. The black box flight recorder has been recovered from the wreckage and the bodies have been identified. The last transmission from the cockpit was, "There seems to be something wrong with the electronics. Going around." The plane disappeared from radar less than ten seconds later.

June 20, 1994

San Francisco, California

Thin clouds drifted high above the city by the Bay. Craig and Stacey sat behind the APSoft building on the large cedar deck. A gentle breeze caressed Stacey's long, summer golden hair. Craig was having a very hard time concentrating on the report in his hands.

"Do you want to hear something weird?" Stacey asked.

"I don't know. Do I?" Craig answered.

"Yes. You do," Stacey said.

"Okay. Let's have it," Craig said.

"We're three for three this year," Stacey said.

"I don't get it," Craig said.

"On airplane crashes. We're three for three."

"I still don't get it," Craig said.

"Listen. First you know that guy in Turkey where the Blackhawks got shot down. Second, we both know Rakesh who's been in Hong Kong where the plane that crashed in Nagoya originated. Third, my friend in Mexico works for that company that chartered that plane that crashed in Virginia the other day. We're three for three."

"Better call the National Enquirer," Craig said.

"Jerk," Stacey said.

"We know somebody at almost every airline or aircraft manufacturer in the world Stacey. It'd be a miracle if we didn't know someone somehow related to every crash," Craig said.

"You're still a jerk," Stacey said.

"Yeah I know. It's part of my charm," he replied.

Stacey made a face at him and rolled her eyes.

"Please," she said.

"But you know what? You've piqued my curiosity. I'm going to do some research and see how many wrecks there have been in the last year. It does seem like there's been an unusual amount doesn't it?" Craig asked.

"Nice try," Stacey said.

"No. I'm totally serious. Now that you've pointed it out, I really am curious."

"Um huh," she said dismissively.

"Ready to throw it some more," Stacey asked, dangling Craig's birthday Frisbee on the end of a long slender finger.

"Not right now," Craig said. I better get started on that research.

http://jtkalnaynovels.wordpress.com/

www.jtkalnay.com

JT Kalnay is an attorney and an author. He has been an athlete, a soldier, a professor, a programmer, an Ironman, and mountain climber. JT now divides his time between being an attorney, being an author, and helping his wife chase after seven nieces and nephews.

JT was born and raised in Belleville, Ontario, Canada. Growing up literally steps from the Bay of Quinte, water, ice, fishing, swimming, boating, and drowning were very early influences and appear frequently in his work.

Educated at the Royal Military College, the University of Ottawa, the University of Dayton, and Case Western Reserve University, JT has spent countless hours studying a wide range of subjects including math, English, computer science, physics, and law. Many of his stories are set on college campuses. JT (along with MC and KR) is one of the founding members of the Stone Frigate Military Academy English Society.

JT is a certified rock climbing guide and can often be found atop crags in West Virginia, California, Texas, New Mexico, Nevada, Kentucky, Mexico, and Italy. Rock climbing appears frequently in his writing.

JT has witnessed firsthand many traumatic events including the World Trade Center Bombing, the Long Island Railroad Shooting, a bear attack, a plane crash, and numerous fatalities, in the mountains and elsewhere.

Disasters, loss, and confronting personal fear are common themes in his writing.

www.jtkalnay.com