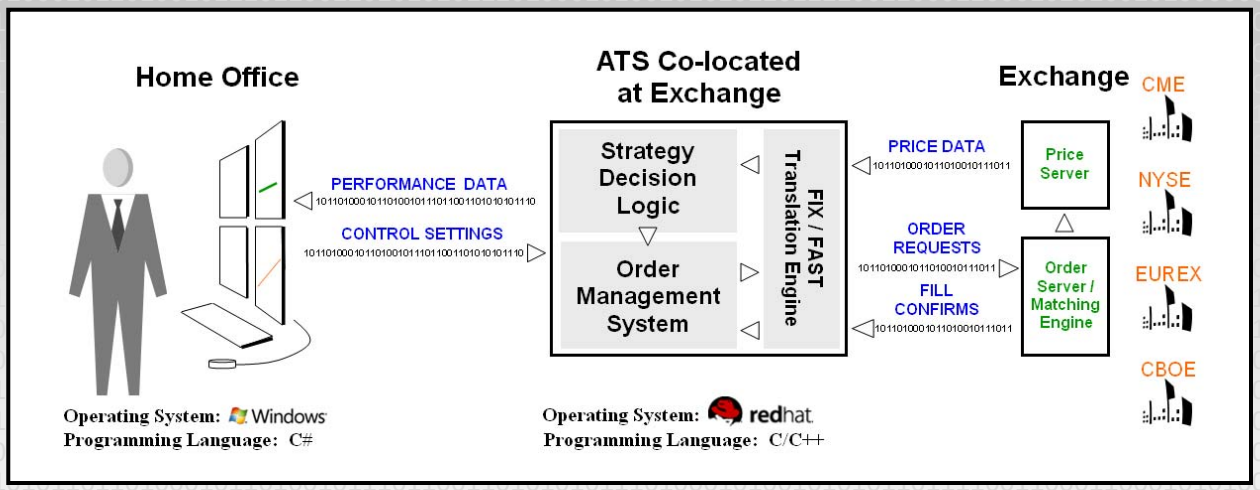
C++ for Financial Applications



Ben Van Vliet

July 9, 2011

MS Visual Studio Development Environment

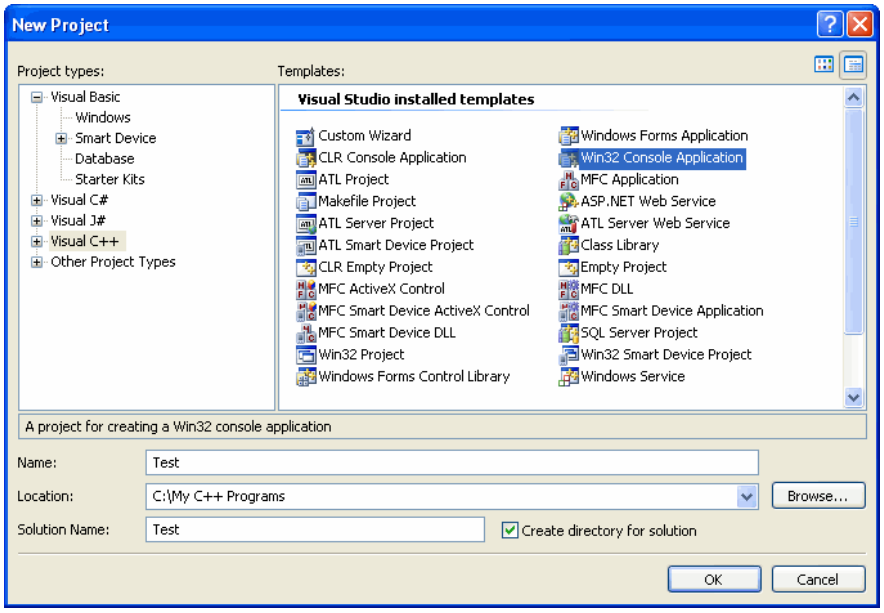
If you are not familiar with computer programming, you may ask "where do I write this code?" An integrated development environment (IDE) is a software package that enables you to write software code. For Windows users, you will likely use Microsoft Visual Studio, and so I will provide a short tutorial on getting started writing ISO/ANSI C++ projects in this IDE.

(You should be able to download the Express version of Visual C++ from the Microsoft website. Go to Google and type Visual C++ Express. The pictures in this tutorial reference the full Visual Studio 2008 package, however.)

Let's create a new Visual C++.NET program that will illustrate the use of namespaces.

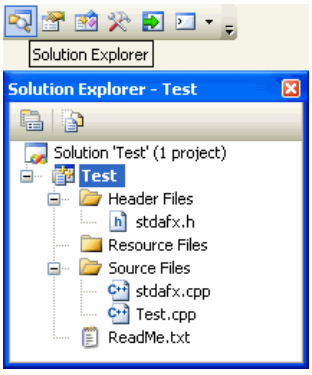
STEP 1 Start Visual Studio.

STEP 2 In the File menu item, click on New and Project… The New Project window will appear.



STEP 3 Highlight Visual C++ Projects in the Projects Types pane and Win32 Console Application in the templates pane. Also, be sure to give your project a name and a location. Click OK and then Finish.

When your solution is ready, on the right hand side of your screen, you should see the Solution Explorer window. If not, click on the icon as shown. The Solution Explorer will allow you to navigate through the files that comprise your solution.

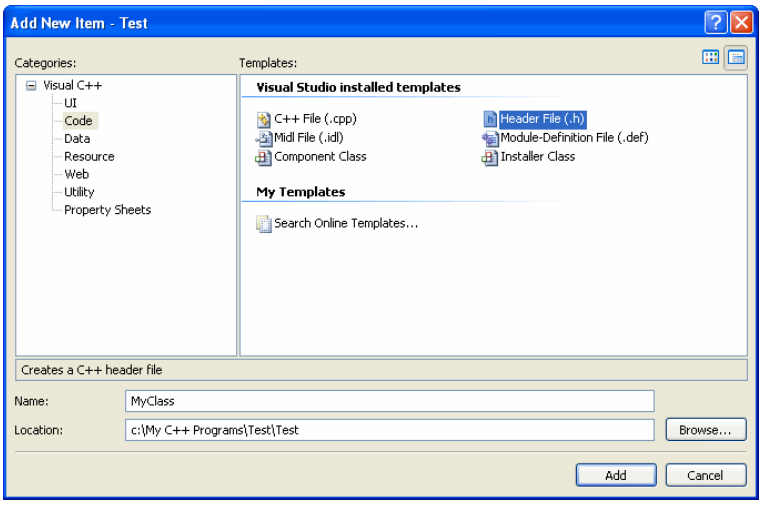


STEP 4 In the Test.cpp code window, type the following:

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

int _tmain(int argc, _TCHAR* argv[])
{
	cout << "Hello, world!" << endl;
	return 0;
}
```

STEP 5 Compile the program by clicking the Build menu item and Build Test. Run the program by clicking the Debug menu item and Start Without Debugging.

Later in the study guide, you will be asked to create classes in separate files, called .h and .cpp files. You can add files to your project by clicking on the Project menu item, then selecting Add New Item… and in the Add New Item wizard window selecting Code as the Category and the appropriate template as shown.



Website

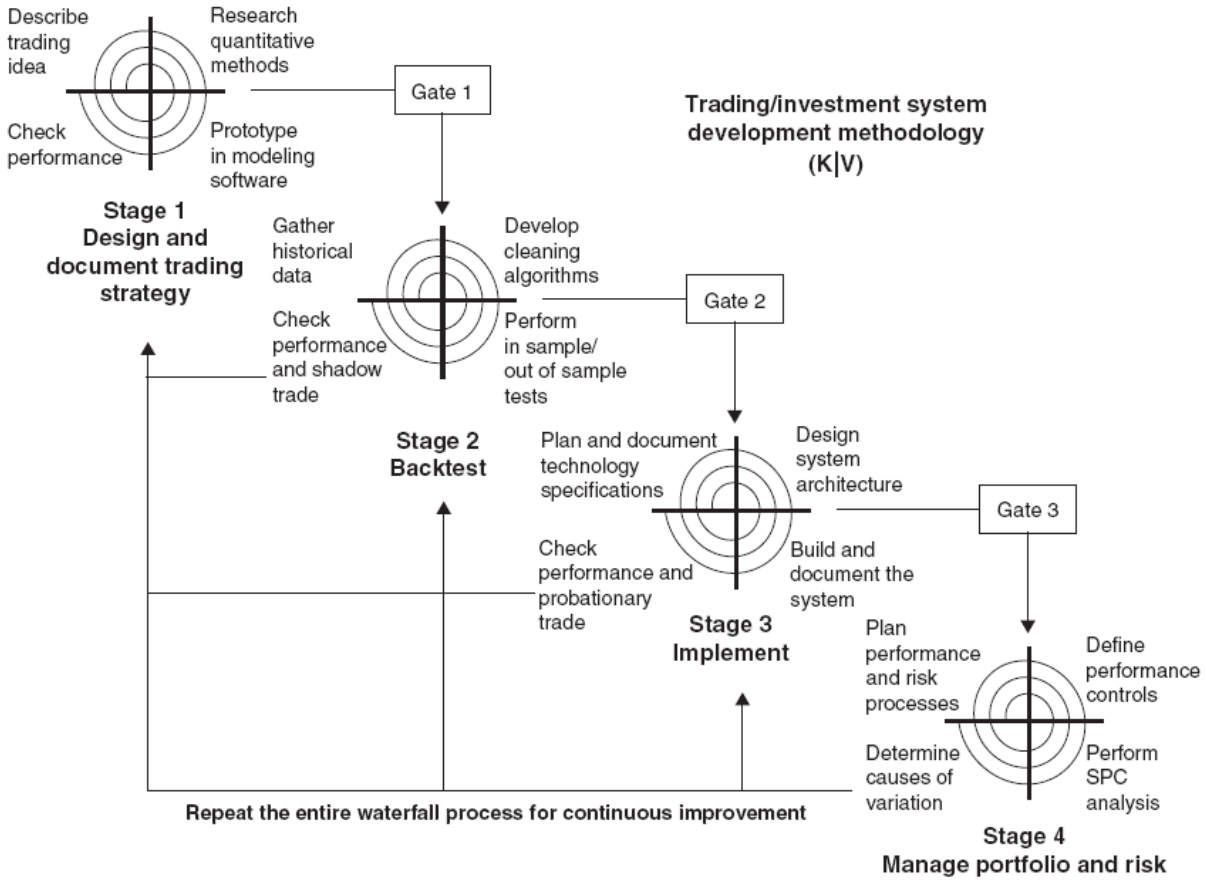
Additional code used in this text is available for download at **www.benvanvliet.com**.

INTRODUCTION

In the new age of automated trading, **finance is an engineering discipline**. This statement has broad implications for how we learn and practice finance.

Where scientists and mathematicians theorize about truth, engineers apply their knowledge of science and mathematics to solve real-world problems. Where many solutions exist, engineers evaluate different solutions and choose the one that best meets requirements. Engineers try to predict how well a solution will perform to specifications prior to full implementation. Thus, they use prototypes, simulations, and stress tests to ensure that solutions will perform as expected. They take seriously their responsibility to generate solutions that will not cause harm, but rather improve the lives of people.

Financial engineering is the application of a systematic approach to the research, development, implementation and operation of trading and investment systems. Kumiega and Van Vliet in their book *Quality Money Management* describe the systematic approach of a financial engineer.



CHAPTER 1: TYPES AND OPERATORS

1. Hello, world!

Every C++ program begins at the main function, called the entry point.

```
#include <iostream>

int main()
{
      // Double forward slash comments a line of code.
      // Commented lines are ignored by the compiler.

      /* Forward slash and an asterisk
         opens and closes several
         consecutive lines of comments. */

      std::cout << "Hello, World!" << std::endl;
      return 0;
}
```

In this short program, `cout` prints to the screen whatever the output operator, <<, sends it. `cin` gets from the keyboard and puts it where the input operator, >>, sends it.

A couple of important notes:

- C++ code is case sensitive
- Every line of code in C++ must end with a semicolon.
- All variables must be declared before they can be defined.
- All of C is encompassed within C++.
- The double forward slash is used to comment lines of code.

2. Preprocessor Directives

Above the main are preprocessor directives, or macros. The preprocessor is a part of the compiler that performs various text manipulations on the code before actually compiling it into machine language object code and preprocessor directives (i.e. instructions to the preprocessor) are not part of C++. #include <iostream> is such a directive and it tells the compiler to include another code file with the current one. Other directives perform conditional compilation--#if, #else, #endif. #define is a preprocessor directive that defines a macro name. #define creates an identifier and a sequence of characters that the compiler will substitute for the identifier each time it encounters the identifier in the source code.

```
#include <iostream>
#define CONTRACT "ES Dec05"

int main()
{
      std::cout << CONTRACT << std::endl;
      return 0;
}
```

Because there are many different compilers C++, there are various macros (i.e. #define) that automatically check for compiler or operating system features. Because C++ is portable across platforms, this is very useful. Because this text uses Microsoft's Visual C++ compiler, the macros we encounter (most notably #pragma once) are specific to it. Other C++ code, written for other compilers (or to be portable) will employ macros not covered in this text.

Preprocessor directives allow for certain calculations to be done during compilation, rather than during runtime. Usually, this kind of thing is usually done to optimize platform-specific code. In this simple example, the modulus operation is performed during compilation.

```
#include <iostream>
using namespace std;

#define m_Value 13

#if m_Value % 2
    #define outcome "ODD"
#else
    #define outcome "EVEN"
#endif

int main()
{
      cout << outcome << endl;
      return 0;
}
```


3. C++ Libraries

ISO, or standard, C++ contains the standard library, which contains functions, type and class definitions, and templates that enable more efficient programming. The declarations of the different functions and types are categorized into several header files that can be #included into your code, some of which are carry-overs from C. Over the course of this text we will make use of several of these libraries.

Standard Library Name	Description
cassert	C diagnostics library
cctype	C character handling functions
cerrno	C errors
cfloat	C characteristics of floating-point types
ciso646	C ISO 646 alternative operator spellings
climits	C sizes of integral types
clocale	C localization library
cmath	C mathematics library
csetjmp	C non-local jumps
csignal	C library to handle signals
cstdarg	C variable arguments handling
cstddef	C standard definitions
cstdio	C library to perform input/output operations
cstdlib	C standard utilities library
cstring	C string manipulation functions
ctime	C time library
limits	Numeric limits
new	Dynamic memory
typeinfo	Type information
exception	Standard exception class
stdexcept	Exception classes
utility	Utility components
iostream	Input and output operations
fstream	Input and output to files
functional	Function objects
memory	Memory elements
string	String class
locale	Localization library
bitset	Bitset class
deque	Double ended queue class
list	List class
map	Map class
multimap	Multiple-key map class
multiset	Multiple-key set class
priority_queue	Priority queue class
queue	FIFO queue class
set	Set class
stack	LIFO stack class
vector	Vector class
iterator	Iterator definitions
algorithm	Algorithms for STL collections
complex	Complex numbers library
valarray	Library for arrays of numeric values
numeric	Generalized numeric operations

4. Using std and Namespaces

C++ defines a rather large set of functions that are contained in the standard library. These functions perform many commonly needed tasks, including I/O functions, mathematical computations, and string handling. Using a using statement brings the standard library into view.

```
#include <iostream>

using std::cout;
using std::endl;

int main()
{
      cout << "Hello, World!" << endl;
      return 0;
}
```

A namespace is a declarative region and its purpose is to localize the names of identifiers so as to avoid naming collisions. The standard namespace is the one used by the C++ library. Using the std namespace causes it to be brought into the current namespace, which gives you direct access to the functions and classes defined in it. These include cout and cin which allow us to print to and get from the screen. Using a namespace brings a namespace into view.

```
#include <iostream>
using namespace std;

int main()
{
      cout << "Hello, World!" << endl;
      return 0;
}
```


5. Escape Codes

Escape codes allow us to format the appearance of screen output. The most commonly used are \t which tabs over, and \n which causes a line feed. Within a cout line of code, endl also performs a line feed and will be used more often.

```
#include <iostream>
using namespace std;

int main()
{
	cout << "Stock\t" << "Price\n";
	cout << "IBM\t" << "81.25\n";
	cout << "C++FAT\t" << "101.43\n";

	cout << "\n\"\t\\\n";

	return 0;
}
```

6. Variables

Variables are physical memory locations in the computer. They have names, addresses and values. All variables must be declared before they can be defined, that is before they can be assigned a value. Here are the basic C++ types a variable can take on:

Type	Size in Bytes (Bits)
char	1 (8)
wchar_t	2 (16)
int	4 (32)
float	4 (32)
double	8 (64)
bool	NA
void	NA

A local variable is one that is known only within the function within which is declared. A global variable is known throughout the entire program and holds its value for the duration of the program. Notice in this example the use of cin to allow the user to set the value of a variable.

```
#include <iostream>
using namespace std;

int main()
{
	int i;
	double d;
	char c;
	float f;
	bool b1;
	bool b2;

	i = 1;
	d = 2.71828;
	c = 'A';
	f = 3.1415927;
	b1 = true;
	b2 = false;

	cout << "i: " << i << endl;
	cout << "d: " << d << endl;
	cout << "c: " << c << endl;
	cout << "f: " << f << endl;
	cout << "b1: " << b1 << endl;
	cout << "b2: " << b2 << endl;

	cout << "Enter new value for d: ";
	cin >> d;
	cout << "The new value of d is: " << d << endl;

	return 0;
}
```


7. Size of Data Types

Using the sizeof operation, we can see the number of bytes each variable takes up.

```
#include <iostream>
using namespace std;

int main()
{
      int i = 1;
      double d = 2.71828;
      char c = 'A';
      float f = 3.1415927;
      bool b1 = true;
      bool b2 = false;

   // The sizeof operator returns an interger type called size_t.

      cout << "Size of i: " << sizeof i << endl;
      cout << "Size of d: " << sizeof d << endl;
      cout << "Size of c: " << sizeof c << endl;
      cout << "Size of f: " << sizeof f << endl;
      cout << "Size of b1: " << sizeof b1 << endl;
      cout << "Size of b2: " << sizeof b2 << endl;

      return 0;
}
```


8. Variable Declaration Qualifiers

Variables can be modified using short, long, signed and **unsigned** keywords. Unsigned variables can only be positive. Additionally, variables can be declared as **const**, **volatile**, **auto**, **extern** or **register**.

```
#include <iostream>
using namespace std;

int main()
{
	const double a = 4.1;
	volatile int b = 2;

	auto d = 2.718;
	extern int e;

	static int f = 0;
	register double g = 3.14;

	unsigned int h = 5;

	return 0;
}
```

A const variable is constant and therefore its value cannot be changed after initialization. The volatile qualifier informs the compiler that a variable may be changed by factors outside the program. The auto specifier simply declares a local variable. Extern declares a variable, but does not allocate storage for it. Static variables are permanent variables within their own functions and maintain their values between function calls. Register requests that a variable be optimized for access speed.

9. Structures and Unions

A structure is a user-defined data type that is a group of related variables.

```
#include <iostream>
using namespace std;

struct Bar
{
	double Open;
	double Close;
	double Volume;
} Monday, Tuesday;

int main()
{
	Monday.Open = 50.00;
	Monday.Close = 51.25;

	Tuesday.Open = 51.25;
	Tuesday.Close = 49.75;

	cout << Tuesday.Close - Monday.Close << endl;

	Bar Wednesday;
	Wednesday.Open = 49.75;

	return 0;
}
```

In this example, Monday and Tuesday are variables of type Bar. We can operate on their members—Open, Close and Volume—using the (.) notation.

Unions allow one memory location to be accessed as different data types. But in fact, all are different names on the same location. A union is similar to a structure, but yet it's totally different. All the elements of the union occupy the same physical memory space.

```
#include <iostream>
using namespace std;

union Grade
{
	char letter;
	int score;
	double percent;
};

int main()
{
	Grade x;
	x.letter = 'A';
	cout << x.letter << endl;
```



```
      Grade y;
      y.percent = 92.50;
      cout << y.percent << endl;

      x.score = 43;
      cout << x.score << endl;

      return 0;
}
```


10. Enums and Typedefs

The typedef keyword allows us to put a new name on an existing data type. An enumeration, or just enum, allows us to create a list of named integer constants.

```
#include <iostream>

enum Position
{
	LONG,SHORT,FLAT
};

typedef float Return1;

int main()
{
	Position m_Position = LONG;
	Return1 returns[] = { .054, .012, .0964 };

	return 0;
}
```


11. Decltype

If we need to know the type of a variable (even the return type of a function) we can use the **decltype** operator. Furthermore, decltype returns the type in a form that allows us to declare a new variable. Decltype does not evaluate the argument expression it is passed. In this example, both *j* and *k* are integers.

```
#include <iostream>
using namespace std;

int main()
{
	decltype( 4 - 2 ) i = 3;

	int j = 5;
	decltype( j ) k = j;
}
```


12. References

Variables in memory have addresses. A reference, for all intents and purposes, is another name on an existing variable's memory location.

```
#include <iostream>
using namespace std;

int main()
{
	int a = 7;
	int &b = a;
	cout << a << endl;
	cout << b << endl;

	cout << &a << endl;
	cout << &b << endl;
	return 0;
}
```


13. Pointers

A pointer is a variable that has as its value the address of another variable. Pointers can be declared, defined and dereferenced. Pointers can point, i.e. contain the memory address, of any type: int, float, char, double, even objects and functions as we will see later. Pointers are distiguished by the type to which they point; a pointer to an integer can not have a value that is the address of any type other than an integer.

```
#include <iostream>
using namespace std;

int main()
{
	int a = 4;

	int *b;

	b = &a;

	int c = *b;

	cout << c << endl;

	return 0;
}
```


14. Pointers Dereferrenced

Dereferencing a pointer means to go to the memory location to which it points.

```
#include <iostream>
using namespace std;

int main()
{
	int a;
	int *b;

	a = 4;
	b = &a;

	cout << "The value of a: " << a << endl;
	cout << "The value of b: " << b << endl;
	cout << "The address of a: " << &a << endl;
	cout << "The address of b: " << &b << endl;
	cout << "The value of b dereferenced: " << *b << endl;

	cout << "Size of b: " << sizeof b << endl;

	return 0;
}
```

15. Pointers to Pointers

Pointers can even point to other pointers.

```
#include <iostream>
using namespace std;

int main()
{
	int a;
	int *aPtr;
	int **aPtrPtr;

	a = 3;

	aPtr = &a;
	aPtrPtr = &aPtr;

	cout << a << endl;
	cout << &a << endl;

	cout << aPtr << endl;
	cout << *aPtr << endl;

	cout << &aPtr << endl;
	cout << aPtrPtr << endl;

	cout << *aPtrPtr << endl;
	cout << **aPtrPtr << endl;

	int x = 4;
	int *y = &x;
	int *&z = y;  //reference to a pointer

	cout << *z << endl;
	cout << &y << endl;
	cout << &z << endl;

	return 0;
}
```

In this example, aPtr points to a, and aPtrPtr has a value that is the address of aPtr. Dereferencing aPtrPtr twice, **aPtrPtr, goes to the memory location to which aPtrPtr points and gets the value of aPtr. The value of aPtr is the address of a, so dereferencing again prints the value of a.

LAB 1

Create a program that asks the user to enter 2 numbers. Place these values into variables called *a* and *b*. Calculate and print the sum of the two numbers. Next, create references to *a* and *b* called *c* and *d* respectively. Calculate and print the sum of *c* and *d*. Lastly, create pointers to *a* and *b* called *e* and *f* respectively. Calculate and print the sum of the values to which *e* and *f* point.

Solution:

```
int main()
{
    int a = 0;
    int b = 0;

    cout << “Please enter two integers.” << endl;
    cout << “a = “;
    cin >> a;
    cout << “b = “;
    cin >> b;

    cout << “a + b = “ << a + b << endl;

    int &c = a;
    int &d = b;

    cout << “c + d = “ << c + d << endl;

    int *e = &a;
    int *f = & b;

    cout << “*e + *f = “ << *e + *f << endl;
    return 0;
}
```

16. Operators

C++ makes use of many, many operators. Just as in algebra, there is an established hierarchy of operations, a priority associated with each operator relative to other operators. From highest to lowest, the priorities are as follows:

Level	Operator	Description
1	::	scope
2	(), [], ., ->, ++, -- dynamic_cast , static_cast, reinterpret_cast, const_cast, typeid	postfix
3	++, --, ~, !, sizeof, new, delete	unary (prefix)
	*, &	pointer indirection and reference
	+, -	unary sign operator
4	(type)	type casting
5	.*, ->*	pointer-to-member
6	*, /, %	multiplicative
7	+, -	additive
8	<<, >>	shift
9	<, >, <=, >=	relational
10	==, !=	equality
11	&	bitwise AND
12	^	bitwise XOR
13	|	bitwise OR
14	&&	logical AND
15	||	logical OR
16	? :	conditional
17	=, *=, /=, %=, +=, -=, >>=, <<=, &= ,^=, |=	assignment
18	,	comma

17. Mathematical Operators

The mathematical operators are completely intuitive. The value computed on the right side of the equals sign is always put into the memory location on the left side.

```
#include <iostream>
using namespace std;

int main()
{
	double a = 2.5;
	double b = 3.5;
	double c;

	c = a - b;

	cout << c << endl;

	c = a * b;

	cout << c << endl;

	c = a + b * c - a / b + 2;

	cout << c << endl;

	c = ( a + b ) * ( b - a / ( b + 2 ) );

	cout << c << endl;

	return 0;
}
```

Sometimes when performing mathematical operations, you need to combine integer and floating-point types (i.e. double, float, etc.) in the same calculation. When this happens the integer is automatically converted to a floating-point form. This is a form of implicit type conversion. **Type casting** is a more explicit way to convert data types. The following program uses casts to produce a floating-point result from two integers. Notice that the cast does not change the type of the variable.

```
#include <iostream>
using namespace std;

int main()
{
   int a, b;
   cin >> a >> b;
   cout << "a / b = " << a / b << endl;
   cout << "a / b with casts = " << ( double ) a / double ( b ) << endl;
}
```


18. Incrementation Operators

Incrementation operators change the values of variables. Calling ++ on a variable adds one to its value; -- subtracts one. The += operator takes the old value of the variable on the left, add the value of the variable on the right and puts the sum back into the variable on the left. This process is similar for -=, *= and /=.

```
#include <iostream>
using namespace std;

int main()
{
	int a = 5;
	cout << a++ << endl; // Post increment
	cout << a << endl;

	cout << ++a << endl; // Pre increment
	cout << a << endl;

	a += 2;

	cout << a << endl;

	a *= 2;

	cout << a << endl;

	a /= 2;

	cout << a << endl;

	a %= 2;

	cout << a << endl;

	return 0;

}
```


19. Arrays

An array is a set of contiguous memory locations. All arrays in C++ are zero-based. Declaring an array requires the number of elements. Any element in an array can be accessed with the index, [], notation.

```
#include <iostream>
using namespace std;

int main()
{
	int m_Array[ 3 ];

	m_Array[ 0 ] = 10;
	m_Array[ 1 ] = 11;
	m_Array[ 2 ] = 12;

	cout << m_Array[ 0 ] << endl;
	cout << m_Array[ 1 ] << endl;
	cout << m_Array[ 2 ] << endl;

// We can initialize the values in an array when the array is declared
// using the bracket notation.

	int m_Array1[] = { 10, 11, 12 };

// Often, we us a constant to define the size of the array.

	const int ARRAYSIZE = 5;
	int m_Array2[ ARRAYSIZE ];

	m_Array2[ 0 ] = 10;

	cout << m_Array2[ 0 ] << endl;


	return 0;
}
```


20. Name of an Array is a Pointer

The name of an array is a constant pointer to the first element of the array.

```
#include <iostream>
using namespace std;

int main()
{
	int m_Array[] = { 10, 11, 12, 13, 14 };

	cout << *m_Array << endl;

	return 0;
}
```


21. Pointers to Arrays

Because the name of an array is a constant pointer to the first element of the array, we cannot change its value. However, we can create another pointer that is not constant, by setting its value equal to the value of the name of the array. This new pointer can be indexed, just like the name of the array, but can also be added to, subtracted from or changed using pointer arithmetic.

```
#include <iostream>
using namespace std;

int main()
{
	int myArray[] = { 10, 11, 12, 13, 14 };

	int *myPtr = myArray;

	cout << *myArray << endl;
	cout << *myPtr << endl << endl;

	cout << myArray[ 0 ] << endl;
	cout << myPtr[ 0 ] << endl << endl;

	cout << myArray[ 1 ] << endl;
	cout << myPtr[ 1 ] << endl << endl;

	myPtr++;

	cout << myArray[ 1 ] << endl;
	cout << *myPtr << endl << endl;

	myPtr += 3;

	cout << myArray[ 4 ] << endl;
	cout << *myPtr << endl << endl;

	myPtr -= 2;

	cout << myArray[ 3 ] << endl;
	cout << myPtr[ 1 ] << endl << endl;

	return 0;
}
```


22. Pointer Arithmetic

Pointer arithmetic allows us to move a non-constant pointer around in memory. Because array elements are contiguous, we can move pointers around by incrementing. Calling ++ on a pointer adds *one times the size of the element to which it points*. Likewise, adding to a pointer calculates a new address that is the sum of the value of the pointer plus *the amount times the size of the array element to which it points*. There is nothing stopping you from incrementing or indexing off either end of an array.

One of the keys to understanding pointer arithmetic is understanding logical equivalence of the following two lines of code:

```
cout << array_name[ 0 ];
cout << *( array_name + 0 );
```

```
#include <iostream>
using namespace std;

int main()
{
      int b[] = { 10, 20, 30, 40 };
      int *bPtr1 = b;
      int *bPtr2 = b;

      cout << b[ 1 ] << endl;
      cout << *( b + 2 ) << endl;
      cout << bPtr1[ 3 ] << endl;
      cout << *( bPtr1 + 1 ) << endl;
      cout << *bPtr2 << endl;
      bPtr2++;
      cout << *bPtr2 << endl;

      return 0;
}
```


23. Char Arrays

Character arrays, usually called strings, are contiguously held characters, with a null character, '\0', attached by default to the end. Passing a string to cout causes it to print until it reaches this null value.

```
#include <iostream>
using namespace std;

int main()
{
	char m_Name1[] = "John Doe";
	char *m_Name2 = "Jane Doe";

	cout << m_Name1 << endl;
	cout << m_Name2 << endl;

// We can even create arrays of pointers.

	char *m_Symbols[] = { "IBM", "INTC", "MCD", "GE", "WMT" };
	cout << m_Symbols[ 2 ] << endl;

	return 0;
}
```


24. Two-Dimensional Arrays

Two dimensional arrays are perfect for holding matrices.

```
#include <iostream>
using namespace std;

int main()
{
	int m_Array[][ 3 ] =
	{
		{ 1, 2, 3 },
		{ 4, 5, 6 },
		{ 7, 8, 9 }
	};

	cout << m_Array[ 2 ][ 1 ] << endl;

	return 0;
}
```


LAB 2

Given the following two-dimensional array:

1	2	7
8	5	6
9	3	4

Create a program prints out the 5 in as many different ways as you can think of

Solution:

```
#include <iostream>
using namespace std;

int main()
{
      int m_Array[][ 3 ] = { { 1, 2, 7 },
                             { 8, 5, 6 },
                             { 9, 3, 4 } };

      int *b = m_Array[ 1 ];

      cout << m_Array[ 1 ][ 1 ] << endl;
      cout << ( *( m_Array + 1 ) )[ 1 ] << endl;
      cout << *( m_Array[ 1 ] + 1 ) << endl;
      cout << *( b + 1 ) << endl;
      cout << ( b + 1 )[ 0 ] << endl;
      cout << b[ 1 ] << endl;
      cout << ( b + 2 )[ -1 ] << endl;
      ++b;
      cout << *b << endl;

      return 0;
}
```

CHAPTER 2: PROGRAM CONTROL

1. Logical Operators

Logical operations return true or false. && and || are the logical AND and OR operators. The == operator is used to test for equality, != for inequality. In C++, 1 represents true and 0 false. In fact, any non-zero number will evaluate to true. The conditional operator, ?, return the first, : or second value depending on whether the evaluation is true or false.

```
#include <iostream>
using namespace std;

int main()
{
	bool x = ( 3 > 2 );
	cout << x << endl;

	int a = 1;
	int b = 2;
	int c = 3;

	cout << ( a >= b && b != c ) << endl;
	cout << ( a <= b || b == c ) << endl;

	cout << ( a > b ? a : b ) << " is bigger." << endl;

	return 0;
}
```

2. If Statements

If statements execute one or potentially one of two sets of commands based upon the outcome of a logical evaluation.

```
#include <iostream>
using namespace std;

int main()
{
	int a = 4;
	int b = 3;

	cout << "Enter a value for a: ";
	cin >> a;

	cout << "Enter a value for b: ";
	cin >> b;

	if ( a >= b )
		cout << "a is greater than or equal to b." << endl;

	if ( a < b )
	{
		cout << "a is less than b..." << endl;
		cout << "...so, b is greater than a. " << endl;
	}

	if ( a == b )
	{
		cout << "a is equal to b." << endl;
	}
	else
	{
		cout << "a is not equal to b." << endl;
	}

	return 0;
}
```


3. Switch – Case Statements

A switch case statement can be thought of as an organized collection of if statements. A switch case statement can be used when there are more than two logical outcomes to an evaluation.

```
#include <iostream>
using namespace std;

int main()
{
	int x = 0;
	int counter1 = 0;
	int counter2 = 0;
	int counter3 = 0;
	int counter4 = 0;
	int counter5 = 0;

	while ( x != -1 )
	{
		cout << "Enter an integer from 1 and 5 ( -1 to exit. ): ";
		cin >> x;

		switch ( x )
		{
			case 1:
				counter1++;
				break;
			case 2:
				counter2++;
				break;
			case 3:
				counter3++;
				break;
			case 4:
				counter4++;
				break;
			case 5:
				counter5++;
				break;
			default:
				cout << "ERROR!" << endl;
				break;
		}
	}

	cout << "Counter 1: " << counter1 << endl;
	cout << "Counter 2: " << counter2 << endl;
	cout << "Counter 3: " << counter3 << endl;
	cout << "Counter 4: " << counter4 << endl;
	cout << "Counter 5: " << counter5 << endl;

	return 0;
}
```


4. Switching Chars

C++ supports switching only on integers and characters.

```
#include <iostream>
using namespace std;

int main()
{
	char x = '0';
	int counterA = 0;
	int counterE = 0;
	int counterI = 0;
	int counterO = 0;
	int counterU = 0;

	while ( x != 'Z' && x != 'z' )
	{
		cout << "Enter a vowel ( Z to exit. ): ";
		cin >> x;

		switch ( x )
		{
			case 'A':
			case 'a':
				counterA++;
				break;
			case 'E':
			case 'e':
				counterE++;
				break;
			case 'I':
			case 'i':
				counterI++;
				break;
			case 'O':
			case 'o':
				counterO++;
				break;
			case 'U':
			case 'u':
				counterU++;
				break;
			default:
				cout << "ERROR!" << endl;
				break;
		}
	}
	cout << "Counter A: " << counterA << endl;
	cout << "Counter E: " << counterE << endl;
	cout << "Counter I: " << counterI << endl;
	cout << "Counter O: " << counterO << endl;
	cout << "Counter U: " << counterU << endl;
}
```


LAB 3

Create a Tick structure that holds the price, as a double, and the TickType, as an enumeration of values BID, ASK and LAST. Write a program that will switch on the enumeration value member of a structure to set the value of a character array with the name of the enumeration value. Print out the array.

Solution:

```
#include <iostream>
using namespace std;

enum TickType
{
	BID = 0,
	ASK,
	LAST
};

typedef double Real;

struct Tick
{
	Real Price;
	TickType Type;
};

int main()
{
	Tick a;
	a.Price = 32.45;
	a.Type = ASK;

	char *m_Type;

	switch ( a.Type )
	{
		case BID:
			m_Type = "BID";
			break;
		case ASK:
			m_Type = "ASK";
			break;
		case LAST:
			m_Type = "LAST";
	};

	cout << m_Type << endl;
	return 0;
}
```


5. For Loops

A for loop will repeat execution of the code within its block. A counter variable increments either up or down until the termination condition is met.

```
#include <iostream>
using namespace std;

int main()
{
	for ( int i = 0; i < 10; i++ )
	{
		cout << i << endl;
	}
	return 0;
}
```


6. While Loops

In a while loop, the block of code will execute repetitively until the termination condition is met.

```
#include <iostream>
using namespace std;

int main()
{
	int x = 0;

	while ( x != 1 )
	{
		cout << "Enter a new value for x: ";
		cin >> x;
	}

	return 0;
}
```


7. Do – While Loops

In a do-while loop, the block of code will execute repetitively until the termination condition is met. Notice that even if the termination condition is met prior to entering into the loop, the block will nevertheless execute at least once.

```
#include <iostream>
using namespace std;

int main()
{
	int x = 1;

	do
	{
		cout << "Enter a new value for x: ";
		cin >> x;
	} while ( x != 1 );

	return 0;
}
```


8. Iterating Through Arrays

For loops and arrays go together well. The counter variable in the for loop can be used as the index on the array. The number of elements in the array is the termination condition. Two dimensional arrays require an embedded loop structure to visit every element.

```
#include <iostream>
using namespace std;

int main()
{
	int myArray1[] = { 10, 11, 12, 13, 14, 15 };

	for ( int x = 0; x < 6; x++ )
	{
		cout << myArray1[ x ] << endl;
	}

	int myArray2[][ 3 ] =
	{
		{ 1, 2, 3 },
		{ 4, 5, 6 },
		{ 7, 8, 9 }
	};

	for ( int x = 0; x < 3; x++ )
	{
		for ( int y = 0; y < 3; y++ )
		{
			cout << myArray2[ x ][ y ] << endl;
		}
	}
	return 0;
}
```


9. Break Statement

A break statement will cause execution to break out of the loop regardless of whether or not the termination condition has been met.

```
#include <iostream>
using namespace std;

int main()
{
	int x;
	for( x = 1; x <= 10; x++ )
	{
		if ( x == 5 )
		{
			break;
		}
		cout << x << endl;
	}

	cout << "Broke out of the loop at x equal to " << x << endl;
	return 0;
}
```


LAB 4

Given the following two-dimensional array:

1	2	7
8	5	6
5	3	4

Create a program that increments a pointer to a one-dimension array to add up all the elements in the array.

Solution:

```
#include <iostream>
using namespace std;

int main()
{
	int m_Array[][ 3 ] = { { 1, 2, 7 },
	                       { 8, 5, 6 },
	                       { 5, 3, 4 } };


	// int **a = m_Array;  This does not work.

	int ( *a )[ 3 ] = m_Array;
	int sum = 0;

	for ( int r = 0; r < 3; r++ )
	{
		for ( int c = 0; c < 3; c++ )
		{
			sum += ( *a )[ c ];
		}
		a++;
	}
	cout << sum << endl;

	return 0;
}
```


10. Memory and Functions

When you run a computer program it is loaded into memory. Memory is organized into the text (or code), stack, and heap segments. The text segment is where the compiled, or machine language code resides. These are the steps the program will execute, which are encapsulated in functions. The stack and the heap are where the compiler allocates memory for data storage. The stack (i.e. last in, first out (LIFO)) is memory allocated to automatic variables within functions. New storage locations on the stack are allocated and deallocated only at the top of the stack.

C++ programs all begin at the main function. When the main function runs, the operating system allocates new space on the stack for all variables that are declared in the main. If subsequent functions are called, additional memory is allocated for the variables in those functions, again starting from the top of the stack. When a function returns, storage for its local variables is deallocated. Thus, stack-based memory is used over and over again while the program is running. Stack memory locations will then necessarily contain garbage values that are left over from previous uses.

So, now let's walk through what happens to the stack when a function is called.

1. The address of the line of code following the function call is saved on the stack, so the CPU knows where to go after the function returns.
2. Space is made on the stack for the function's return value.
3. Program execution proceeds to the function definition.
4. A cursor (really a pointer call the stack frame) holds the location of the current top of the stack, so that all new allocations must be local to the function.
5. All function parameters, or input arguments, are allocated on the stack.
6. The function executes.
7. When the function is done running, the return value is copied into the space allocated in step 2.
8. All memory allocated above the stack frame pointer is deallocated, or popped, destroying all local variables.
9. The return value is popped and its value is assigned to a receiving location. If no space is allocated to receive it, the value is lost.
10. The address of the next line of code, stored in step 1. is popped and execution resumes at that line of code.

Heap-based memory, as we will see later, provides for more stable storage. Memory allocated on the heap exists forever, unless it is explicitly deallocated. Global, or external, and static variables are allocated on the heap.

11. Creating Functions

All C++ code that executes is found in functions. A function is a segregated block of code that performs some task. Every function has a name, a list of parameters. or input arguments, and a return value, unless the return type is void. A function's name is used to call that function from another function, parameters are passed to the function, and the return value sent back to the calling function. Functions in C++ must be prototyped, which is a declaration of the function and its type, or its signature, collectively its parameter types and the return type. The function definition comes later.

```
#include <iostream>
using namespace std;

int Add( int, int );

int main()
{
      int c = Add( 1, 2 );

      cout << c << endl;

      return 0;
}

int Add( int a, int b )
{
      return a + b;
}
```

We also use the terms early binding and late binding with respect to function calls. Early binding resolves a function call a compile time, late binding resolves a function call at run time.

12. Passing by Value

When we pass a parameter to a function by value, a local copy of that value is made for use over the course of the function definition.

```
#include <iostream>
using namespace std;

int SquareByValue( int );

int main()
{
      int a = 2;

      int b = SquareByValue( a );

      cout << b << endl;

      return 0;
}

int SquareByValue( int x )  // x is a copy of a in the main
{
      return x * x;
}
```


13. Passing by Reference

Passing by reference does not make a copy of the input argument, but rather a reference to the original variable in memory. This is inherently unsafe as any change made to the value of the reference variable of over the course of the function definition will necessarily also be a change to the variable in the calling function.

```
#include <iostream>
using namespace std;

void SquareByReference( int & );

int main()
{
	int a = 2;

	SquareByReference( a );

	cout << a << endl;

	return 0;
}

void SquareByReference( int &x )  // Any change to x changes the value of a.
{
	x *= x;
}
```


14. Passing by Pointer

Passing by pointer creates a copy of the pointer. This is also inherently unsafe as any change to the value to which the pointer points, over the course of the function definition, will necessarily change the value of the underlying variable in the calling function.

```
#include <iostream>
using namespace std;

void SquareByPointer( int * );

int main()
{
      int a = 2;

      SquareByPointer( &a );

      int *b = &a;

      SquareByPointer( b );


      cout << a << endl;

      return 0;
}

void SquareByPointer( int *x )  // Any change to *x will change a.
{
       *x = *x * *x;
}
```


15. Passing the Return Value as a Parameter

```
#include <iostream>
using namespace std;

void AddOne( int, int & );

int main()
{
      int x = 4;
      int y = 0;

      AddOne( x, y );

      cout << x << endl;
      cout << y << endl;

      return 0;
}

void AddOne( int a, int &b )
{
      b = ++a;
}
```

LAB 5

Create a function that accepts three integers, one by value, one by reference, and one by pointer. This function should return the sum of the three values.

Solution:

```
#include <iostream>
using namespace std;

int Add( int, int &, int * );

int main()
{
      int a = 4;
      int b = 5;
      int c = 6;

      cout << Add( a, b, &c ) << endl;

      return 0;
}

int Add( int x, int &y, int *z )
{
      return x + y + *z;
}
```


16. Incrementing Dereferenced Pointers

Just like with other operators, dereferencing of a pointer will occur relative to the presence and precedence of other operators. In this example, dereferencing of the pointer in the Increment function must be done inside parentheses to force that operation to occur before the incrementation.

```
#include <iostream>
using namespace std;

void Increment( int * );

int main()
{
	int a;
	int *aPtr;

	a = 3;
	aPtr = &a;

	Increment( aPtr );

	cout << a << endl;
	cout << *aPtr << endl;

	return 0;
}

void Increment( int *myPtr )
{
	( *myPtr )++;
}
```


17. Passing Arrays to Functions

Arrays can also be passed to functions. Because array names are just pointers and do not know the size of the array to which they point, we usually pass the size of the array to the function as well.

```
#include <iostream>
using namespace std;

void PrintArray( double [], int );

int main()
{
	const int SIZE = 5;

	double myArray[ SIZE ];

	myArray[ 0 ] = 0.1;
	myArray[ 1 ] = 0.1;
	myArray[ 2 ] = 0.2;
	myArray[ 3 ] = 0.3;
	myArray[ 4 ] = 0.4;

	PrintArray( myArray, SIZE );

	return 0;
}

void PrintArray( double myA[], const int mySize )
{
	for ( int x = 0; x < mySize; x++ )
	{
		cout << myA[ x ] << endl;
	}
}
```


LAB 6

A common problem in programming is sorting an array of numbers from low to high or vice versa. Several well-known algorithms for sorting exist, including **bubble sort**, **quick sort**, **insertion sort**, **selection sort**, and **heap sort**, just to name a few. (Most of these algorithms are available in C++ somewhere on the internet.) Some are these algorithms are simple and fast, some are complex and slower but more robust.

We use **Big O notation** to describe the complexity of an algorithm by the number of steps it takes to complete, always describing the worst-case scenario. As with all things C++, the best way to understand Big O is by code examples. So, below are some common orders of Big O magnitude with descriptions.

O(1) describes an algorithm that always executes in the same number of steps. In this example, the IsNull function always executes in the same number of steps, regardless of how many elements are in the array *x*.

```
bool IsNull( int x[] )
{
	if( x == NULL )
	{
		return true;
	}
	return false;
}
```

Alternatively, **O(N)** describes an algorithm where the number of steps will grow linearly in proportion to the size of the array. In this example, the algorithm tests each element in the array *x* for equality to *y*. Notice that O(N) notation assumes that the for loop will always iterate to the end of the array, the worst case scenario.

```
bool Contains( int x[], int length, int y )
{
	for( int i = 0; i < length; i++ )
	{
		if( x[ i ] == y )
		{
			return true;
		}
	}
	return false;
}
```

O(N^2) describes an algorithm where the number of steps is directly proportional to the square of the size of the array. This is common with algorithms that involve nested loops. **O(2^N)** describes an algorithm where the number of steps will double with each additional element in the array. Other algorithms, for example those that use bisection methods, grow in **O(log N)**. That is, the iterative halving of the data flattens out the number of steps, even as the data set grows very large. So, for example, tripling the size of the input array has little effect on the number of steps. This makes bisection algorithms more efficient for large data sets.

In this lab, use the bubble sort algorithm to sort the elements in a one dimensional array of integers. Also, describe in Big O notation the number of steps the bubble sort algorithm takes to complete.

Solution:

```
#include <iostream>
using namespace std;

void BubbleSort( int [], int );

int main()
{
	int a[] = { 5, 3, 1, 4, 2, 6 };
	BubbleSort( a, 6 );

	for ( int x = 0; x < 6; x++ )
	{
		cout << a[ x ] << " ";
	}
	cout << endl;
	return 0;
}

void BubbleSort( int a[], int length )
{
	bool swap_flag = true;
	int temp;
	for( int i = 1; ( i < length ) && swap_flag; i++ )
	{
		swap_flag = false;
		for( int j = 0; j < length - 1; j++ )
		{
			if ( a[ j + 1 ] > a[ j ] )
			{
				temp = a[ j ];
				a[ j ] = a[ j + 1 ];
				a[ j + 1 ] = temp;
				swap_flag = true; // has a swap occurred?
			}
		}
	}
}
```

Bubble sort grows in **O(N^2)**.

18. Passing Char Arrays

Character arrays are terminated by the null character, so there often is no need to pass the size.

```
#include <iostream>
using namespace std;

void Print( char * );

int main()
{
	char *myName = "John Doe";

	Print( myName );

	return 0;
}

void Print( char *myText )
{
	cout << myText << endl;
}
```


19. Returning a Pointer?

Functions can return value, references and pointers. However, as this example shows, the local variable *c* in the Add function will be overwritten in subsequent calls.

```
#include <iostream>
using namespace std;

int *Add( int, int );

int main()
{
      int *x = Add( 3, 2 );
      cout << *x << endl;

      int *y = Add( 4, 5 );
      cout << *x << endl;    // The value to which x points has changed!

      return 0;
}

int *Add( int a, int b )
{
      int c = a + b;
      return &c;
}
```


20. Function Pointers and C-Style Callbacks

```
#include <iostream>
using namespace std;

int add( int i )
{
      return i + 1;
}

int main()
{
      int ( *function_ptr )( int );

      function_ptr = add;

      cout << function_ptr( 2 ) << endl;

      return 0;
}
```

We can even pass pointers to functions to functions. In this example, the extreme() function accepts an integer array, and integer, and a pointer to a function—but not just any function, a function with a particular type, or signature. That is, the function pointer argument passed to extreme() must be a pointer to a function that accepts two integers as input parameters and returns an integer. greater() and less() are two such functions. So, when we pass greater to extreme(), f becomes a pointer to greater() over the course of the extreme() definition. When we call f(), greater() runs. Try it with lesser() too.

```
#include <iostream>
using namespace std;

int extreme( int [], int, int (*)( int, int ) );
int greater( int, int );
int lesser( int, int );

int main()
{
      int a[] = { 4, 5, 3, 1, 2 };
      cout << extreme( a, 5, greater ) << endl;

      return 0;
}

int extreme( int a[], int count, int ( *f )( int, int ) )
{
      int v = a[ 0 ];
      for( int i = 1; i < count; i++ )
            v = f( a[ i ], v );

      return v;
}
```



```
int greater( int a, int b )
{
	if( a > b )
		return a;
	else
		return b;
}

int lesser( int a, int b )
{
	if( a < b )
		return a;
	else
		return b;
}
```


21. Array of Function Pointers

Here is a more advanced example that uses an array, f, of function pointers. These functions all have the same type, or signature; i.e. they all accept an int and return void.

```
#include <iostream>
using namespace std;

void function0( int );
void function1( int );
void function2( int );

int main()
{
      void ( *f[ 3 ] )( int ) = { function0, function1, function2 };
      int choice;

      cout << "Enter 0, 1 or 2: ";
      cin >> choice;

      ( *f[ choice ] )( choice );

      return 0;
}

void function0( int a )
{
      cout << "Function 0 was called." << endl;
}

void function1( int a )
{
   cout << "Function 1 was called." << endl;
}

void function2( int a )
{
   cout << "Function 2 was called." << endl;
}
```


LAB 7

Create a function that accepts an array of characters, and a pointer to a function that accepts a character pointer. From within the function definition, dereference the function pointer and pass in the array. This function should print out the elements in the array one by one.

Solution:

```
#include <iostream>
using namespace std;

void MyFunction( char [], void (*)( char * ) );
void Print( char * );

int main()
{
	char name[] = "John Doe";

	MyFunction( name, Print );

	return 0;
}

void MyFunction( char n[], void (*Write)( char * ) )
{
	Write( n );
}

void Print( char *x )
{
	int i = 0;
	while ( x[ i ] != '\0' )
	{
		cout << x[ i ] << endl;
		i++;
	}
}
```


22. Overloaded Functions

Overloaded functions are two or more functions that have the same name, but have different signatures, either the number or type(s) of input parameters. The compiler will first check to see which version of the function you mean to call and associate the call with the correct definition.

```
#include <iostream>
using namespace std;

int Add( int, int );
int Add( int, int, int );
double Add( double, double );

int main()
{
	int a = 1;
	int b = 2;
	int c = 3;

	double d = 1.1;
	double e = 2.2;

	cout << Add( a, b ) << endl;
	cout << Add( a, b, c ) << endl;
	cout << Add( d, e ) << endl;

	return 0;
}

int Add( int x, int y )
{
	return x + y;
}

int Add( int x, int y, int z )
{
	return x + y + z;
}

double Add( double x, double y )
{
	return x + y;
}
```


23. Library Functions

The math.h library, cmath in this implementation, contains many common mathmatical functions.

```
#include <iostream>
#include <cmath>

using namespace std;

int main()
{
	double a = 4.0;
	int b = 3;

	int c = pow( a, b );
	cout << c << endl;

	double d = exp( 2.0 );
	cout << d << endl;

	cout << sqrt( 36.0 ) << endl;

	return 0;
}
```

The string.h library (#include <cstring>) contains string manipulation functions. (A string is a character array.) The string functions include strcpy(), string copying, strcat(), string concatenation, strlen(), for string length, and strcmp(), for string comparison.

24. Using Time in a Function

```
#include <iostream>
#include <ctime>

using namespace std;

void sleep( unsigned int );

int main()
{
    time_t now = time( NULL );
    cout << ctime( &now ) << endl;

    cout << "Going to sleep." << endl;
    sleep( 5000 );
    cout << "Im awake!" << endl;

    return 0;
}

void sleep(unsigned int milliseconds)
{
    clock_t goal = milliseconds + clock();
    while ( goal > clock() );
}
```

25. Converting and printf Functions

The atof() function converts a character array into a float-type numerical representation. atoi() converts to an integer. Notice that this example makes use of the stdlib library rather than iostream. Stdlib is the C library which iostream essentially replaced. Nevertheless, the printf function, replaced by cout, is still commonly used. Within a printf function call, %s, %i and %e are used to hold the places of strings, integers and doubles respectively within the argument string. The inclusion list of these values is included as additional parameters.

```
#include <cstdlib>

int main()
{
      printf( "%f\n" );

      char *m_String = "  -2309 ";
      int m_Int = atoi( m_String );

      printf( "atoi( \"%s\" ) = %i\n", m_String, m_Int );

      char *m_OtherString = "3.1415926";
      double m_Double = atof( m_OtherString );

      printf( "atof( \"%s\" ) = %e\n", m_OtherString, m_Double );

      return 0;
}
```


26. Typeid Function

The typeid function can be used to compare types of different variables.

```
#include <iostream>
using namespace std;

int main()
{
    int a = 7;
    int b = 6;

    cout << "a is an " << typeid( a ).name() << endl;

    if ( typeid( a ) == typeid( b ) )
         cout << "a and b are of the same type" << endl;

    return 0;
}
```


27. Recursive Functions

A recursive function is a function that calls itself from within its own definition.

```
#include <iostream>
using namespace std;

int Sum( int );

int main()
{
	cout << Sum( 10 ) << endl;
	return 0;
}

int Sum( int a )
{
	if ( a == 0 )
	{
		return 0;
	}
	else
	{
		return a + Sum( a - 1 );
	}
}
```


28. Default Function Parameters

Functions can have default parameter values. If the function is called and a parameter is not passed, the local variable in the function will take on the default value.

```
#include <iostream>
using namespace std;

double Square( double x = 2 );

int main()
{
	cout << Square() << endl;
	cout << Square( 3 ) << endl;
	return 0;
}

double Square( double x )
{
	return x * x;
}
```


29. Inline Functions

An inline function is a function whose code is compiled and placed within the compiled code of the calling function. Inline functions are usually very short.

```
#include <iostream>
using namespace std;

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

int main()
{
      cout << square( 2.0 ) << endl;
      return 0;
}
```


30. Hexadecimal Notation

A byte consists of 8 bits, which take on the value 0 or 1.

Index	7	6	5	4	3	2	1	0
Power of 2	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Decimal	128	64	32	16	8	4	2	1
Bits	**0**	**1**	**0**	**1**	**1**	**1**	**1**	**0**

In binary numbers, the places are represented by powers of 2, so that the bit sequence shown here is 64 + 16 + 8 + 4 + 2 = 94.

Often, we use hexadecimal, or base 16, notation to refer to the bit sequence. The 16 digits in hexadecimal notation describe the two 4-bit sequences in a byte. In the example, the two 4-bit sequences in 94 are 0101 and 1110. Individually, these are 5 and 14 in decimal. The hex digits are:

Decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Hexadecimal	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f

Each of the 16 hexadecimal characters can be represented by four binary digits: 0000 through 1111. Thus, there are 16 possible combinations of 0's and 1's. Using these digits, 94 is represented as 5e in hexadecimal notation. In C++ we add the prefix 0x, so that 94 is 0x5e.

For *unsigned* integers (i.e. non-negatives of type unsigned int), the binary notation above works nicely. However, for *signed* integers (i.e. any integer of type int) that are negative, the **two's complement** representation of the absolute value is used. For signed numbers, the left-hand-most bit (i.e. in position 7) is zero for positive numbers. For negative numbers, it is one. To create the two's complement binary value for a negative number:

1. Find the binary value for the absolute value.
2. Complement that value.
3. Add one.
4. Add leading ones till you get the proper total number of bits.

Bits									Decimal Value
7	**6**	**5**	**4**	**3**	**2**	**1**	**0**		
0	1	1	1	1	1	1	1	**=**	**127**
0	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**=**	**1**
0	0	0	0	0	0	0	0	**=**	**0**
1	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**=**	**-1**
1	1	1	1	1	1	1	0	**=**	**-2**
1	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**=**	**-128**

Floating point numbers use negative powers of 2, which are set off by the radix point. For example, the decimal value 13.625 consists of 13 to the left of the radix point, and 625 to the right.

Power of 2	2^3	2^2	2^1	2^0	Radix	2^{-1}	2^{-2}	2^{-3}
Decimal	8	4	2	1		.5	.25	.125
Bits	1	1	0	1	**.**	1	0	1

Here, the binary notation is 1101.101. Thus, the decimal value is calculated as: 8 + 4 + 1 + .5 + .125 = 13.625.

```
#include <iostream>
using namespace std;

int main()
{
	int x = 0x5e;
	cout << x << endl;
	cout << hex << x << endl;
	return 0;
}
```

31. Bitwise Operators

C++ contains operators can test, set and shift bits in character and integer bytes. (These operators cannot be used on any other data types than chars and ints.) These operators are &, |, ^, ~, <<, and >>. A bitwise operator produces a value in accordance with a specific operation. The following program uses bitwise AND operator, &, to compare a given integer to 128 (10000000), 64 (0100000), 32 (00100000), etc. consecutively. Take the case where 94 is compared to 16:

94	0	1	0	1	1	1	1	0
16	0	0	0	1	0	0	0	0
&	0	0	0	1	0	0	0	0

In this case, the & operation generates a bit by bit comparison between the two bytes. Only where both 94 and 16 contain a 1 does the operation return 1. The final outcome is a byte whose integer value is 16, which evaluates to true in the following boolean test.

```
#include <iostream>
using namespace std;

void print_binary( int & );

int main()
{
      int x = 26;
      print_binary( x );
      cout << x << endl;

      int y = x >> 1;
      print_binary( y );
      cout << y << endl;

      int z = ~y;
      print_binary( z );
      cout << z << endl;

      return 0;
}

void print_binary( int &a )
{
   for( int b = 128; b > 0; b /= 2 )
   {
            if ( a & b )  // Non-zero outcomes are true.
            {
                  cout << "1 ";
            }
            else
            {
                  cout << "0 ";
            }
   }
   cout << endl;
}
```


If a bit is set to 1 in either operand, the bitwise OR operator, |, will set the result to 1. If and only if one is 1 and the other bit is 0, will the bitwise XOR operator, ^, set the resulting bit to 1. The shift operators, << and >>, move all the bits in an integer or character to the right or left by some number of bits. New end bits are replaced with 0. Left and right shifts are equivalent to integer multiplication and division by 2 respectively. The bitwise complement operator ~ flips every bit, from 0 to 1 or from 1 to 0.

In some case, bit operators can replace mathematical operators. In the particular case shown here, the alternative implementation has performance gains relative to the more intuitive first algorithm that uses iteration.

```
#include <iostream>
using namespace std;

int main()
{
	int n = 10;

	int sum1 = 0;
	for( int i = 1; i <= n; i++ )
		sum1 += i;

	cout << sum1 << endl;

	// As an alternative, we can shift right by one bit.
	// This is equivalent to divide by 2 when n is
	// non-negative.

	int sum2 = ( n * ( n + 1 ) ) >> 1;
	cout << sum2 << endl;

	return 0;
}
```


LAB 8

The FAST messaging protocol (an acronym for **F**IX **A**dapted for **ST**reaming) standardizes the structure of market data for multicast communication as a stream of bytes. A FAST **message** consists of a sequence of one or more fields. A **field** consists of a sequence of one or more bytes. Each **field** has a name and a data type. A **presence map** (PMap) is a field, where each bit signals the presence of a field in a message. A **stop bit** (SBit) is the initial bit in a byte and, if set to 1, indicates that the byte is the last byte in the field. The use of SBits allows field lengths to be of any size. Without SBits, field lengths would have to be fixed.

Consider a message that contains the following information:

Field	Value
Buy/Sell	Sell
Symbol	GM
Quantity	2900
Price	16.68

Taking one field at a time, the first field (i.e. the PMap) indicates the presence of the four fields: Buy/Sell, Symbol, Quantity and Price. With the SBit coded in red, the PMap is:

	Byte	Value
Hexadecimal	0xf8	0x78
Binary	**1**111 1000	111 1000

The **Buy/Sell** field (i.e. Buy = 1, Sell = 0) is a one byte representation indicating a sell:

	Byte	Value
Hexadecimal	0x80	0x00
Binary	**1**000 0000	000 0000

The **Symbol** field is an ASCII character where a letter is one byte. The symbol GM is coded as:

	Byte	Value
Hexadecimal	0x47cd	0x474d
Binary	**0**100 0111 **1**100 1101	100 0111 100 1101

The **Quantity** field is an unsigned integer for 2900:

	Byte	Value
Hexadecimal	0x16d4	0x0b54
Binary	**0**001 0110 **1**101 0100	001 0110 101 0100

Integer values that are longer than one byte are effectively shifted to the right when evaluating the binary representation. For example, the remaining value above without the stop bits and the resulting binary evaluation is:

Binary with SBits:	**0**001 0110 **1**101 0100
Binary evaluation:	**00**00 1011 0101 0100

The most significant bits are filled in with zeros. This 2-byte long field has two new zeros fill in the most significant bits, while the other bits shift to the right one position.

The **Price** field is a scaled number using a single field implementation. Single field scaled numbers are represented as one field. The first byte in the field is the exponent represented as a signed integer and the following bytes represent the mantissa as a signed integer. The value for $16.68 shown, where a negative exponent is given by a two's complement representation.

	Byte	**Value**
Hexadecimal	0x7e0d84	0x7e0684
Binary	**0**111 1110 **0**000 1101 **1**000 0100	111 1110 000 1101 000 0100

0x7e 0x 0d 84: **0**111 1110 **0**000 0110 **1**000 0100 -> value = 0x7e 0x 06 84 -> **Exponent = -2** Mantissa = 1668 -> **111 1110** 000 0110 1000 0100. The exponent and mantissa are evaluated separately.

*Note: Since the exponent value can be negative and the two's complement notation uses the most significant bit to indicate negative values (i.e. 1111 1110 = -2); and since we are using an SBit in the most significant bit, the two's complement negative value indication begins with the next bit (i.e. _**1**11 1110 = -2).*

The entire message is the composite byte stream is: **0x f8 80 47 cd 16 d4 7e 0d 84**

For sample code that parses (or reads this FAST message), see the FAST Example available at **www.benvanvliet.com**.

CHAPTER 3: CLASSES AND OBJECTS

1. Abstraction and Encapsulation

In object oriented programming, abstraction is the process of decomposing a problem into its constituent parts. Each part becomes a self-contained class that contains its own data and functions, usually called methods when inside a class. Encapsulation is a mechanism that binds together code and the data it manipulates, and that keeps both safe from outside interference. Encapsulation binds data and functionality together into a class type, a blue print for an object. We say that an object then is an instance of a class. Within a class, the data is private, and therefore is accessible only from within the methods that are part of the class. Methods may be public or private. The collection of public methods on a class define the object's interface, the parts of the object that other parts of our program may interact with. Lastly, because the data and the methods in a class are bound together, there is no need to pass internal data members back and forth between member functions; they all have direct access to the data members that are part of the class.

2. Creating a Class

A class encapsulates data and functionality. In C++ the typical file structure of a class is to contain the class header in a .h file, and the corresponding method definitions in a .cpp file. The .h file contains the declarations of data members and the function prototypes.

The this pointer is a pointer to the object that invokes, or calls, a member function. Each time a member function is called, it is automatically passed a pointer called this, to the object on which it is called. The this pointer is an implicit parameter to all member functions.

MyClass.h

```
#pragma once

class MyClass
{
private:
	int myValue;

public:
	MyClass(void);
	~MyClass(void);

	void set_Value( int );
	int get_Value();
};
```

MyClass.cpp

```
#include ".\myclass.h"

MyClass::MyClass(void)
{
}

MyClass::~MyClass(void)
{
}

void MyClass::set_Value( int a )
{
	myValue = a;
}

int MyClass::get_Value()
{
	return myValue;
}
```


3. Creating a Value Type Object on the Stack

An object is an instance of a class and we can create objects using the same syntax we use to create variables. Value type objects can be created on the stack just like variables.

```
#include <iostream>
using namespace std;

#include "MyClass.h"

int main()
{
	MyClass myObj;

	myObj.set_Value( 3 );

	cout << myObj.get_Value() << endl;

	return 0;
}
```


4. Dynamic Memory Allocation

There are two ways to store data in memory. The first, as we have seen, is through the use of variables. The second way is through the use of dynamic memory allocation, where memory is allocated during runtime as needed from the free area of memory that lies between your program and the stack. This region is called the heap.

In C the functions malloc and free, found in stdlib, obtain and release memory during execution. This is called dynamic memory allocation. Each time a malloc memory request is made, a portion of the remaining free memory is allocated. When free is called, that memory is returned to the operating system. Malloc requires the number of bytes to be allocated as a parameter and returns the memory location as a pointer to void, which must be cast to convert it to the type of pointer you need.

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

using namespace std;

int main()
{
      int i;
      char *buffer;

      cout << "How long do you want the string? " << endl;
      cin >> i;

      buffer = ( char* ) malloc( i + 1 );

      for ( int n = 0; n < i; n++ )
      {
            buffer[ n ] = 'x';
      }
      buffer[ i ] = '\0';

      cout << "String: \t" << buffer << endl;
      free( buffer );

      return 0;
}
```

```
#include <iostream>
#include <new>
using namespace std;

int main()
{
      int *a = new int( 5 );
      cout << *a << endl;
      delete a;

      ////////////////////////////////
```



```
      int *b = new (nothrow) int[ 5 ];

      if ( b == 0 )
      {
            cout << "Allocation failed."  << endl;
            return 0;
      }

      for( int i = 0; i < 5; i++ )
      {
            b[ i ] = i + 10;
            cout << b[ i ] << endl;
      }

      delete [] b;

      return 0;
}
```

In this example, the nothrow constant is used as an argument for the new operator to indicate that these functions should not throw an exception on failure, but rather return a null pointer instead. In C++, as we will see, the new and delete operators take the place of malloc and free. New allocations sufficient memory to hold the value requested and returns a pointer to it. Failing to free or delete heap-allocated memory results in a memory leak that may, if many such leaks occur, exhaust the heap!

5. Creating a Reference Type Object on the Heap

Reference type objects can be created on the heap using dynamic memory allocation via the new and delete operators.

```
#include <iostream>
using namespace std;

#include "MyClass.h"

int main()
{
	MyClass *m_ObjPtr = new MyClass;

	m_ObjPtr->set_Value( 3 );

	cout << m_ObjPtr->get_Value() << endl;

	delete m_ObjPtr;

	return 0;
}
```


6. Constructor Methods

An object's constructor is the method that runs whenever a new instance of the class is created. You can recognize the constructor because it will have the same name as the class itself. Constructors can accept parameters but no return type is defined and no return values can be passed. Constructors are very often overloaded. In this example, notice the syntax used to pass a parameter to the constructor when the object is instantiated.

MyClass.h

```
#pragma once

class MyClass
{
private:
	int myValue;

public:

	MyClass( int );

	void set_Value( int );
	int get_Value();
};
```

MyClass.cpp

```
#include ".\myclass.h"

MyClass::MyClass( int x )
{
	myValue = x;
}

void MyClass::set_Value( int x )
{
	myValue = x;
}

int MyClass::get_Value()
{
	return myValue;
}
```

Main.cpp

```
#include <iostream>
#include "MyClass.h"
```



```
using namespace std;

int main()
{
      MyClass m_Obj( 3 );    // Passing 3 to the constructor here.
      cout << m_Obj.get_Value() << endl;

      MyClass *m_ObjPtr = new MyClass( 4 );   // Passing 4 here.
      cout << m_ObjPtr->get_Value() << endl;
      delete m_ObjPtr;

      return 0;
}
```

7. Destructor Methods

An object's destructor method runs when the object is destroyed, which occurs, in the case of a value type object, when the object goes out of scope (as with all value types), and in the case of a reference type object, when delete is called. The destructor is easily recognized because it has the same name as the class itself, but is preceded with a tilde, ~.

MyClass.h

```
#pragma once
#include <iostream>
using namespace std;

class MyClass
{
private:

      int myValue;

public:

      MyClass( int );
      ~MyClass(void);

      void set_Value( int );
      int get_Value();
};
```

MyClass.cpp

```
#include ".\myclass.h"

MyClass::MyClass( int x )
{
      myValue = x;
}

MyClass::~MyClass(void)
{
      cout << "Object " << myValue << " is dying!" << endl;
}

void MyClass::set_Value( int x )
{
      myValue = x;
}

int MyClass::get_Value()
{
      return myValue;
}
```


Main.cpp

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

int main()
{
	MyClass m_Obj( 1 );
	MyClass *m_ObjPtr = new MyClass( 2 );

	delete m_ObjPtr;

	return 0;
}
```

8. Initialization

Class member variables are often assigned values in the constructor, as with the previous example. An alternative way to set the value of a data member is through initialization. Notice the syntax in the constructor of the following example.

MyClass.h

```
#pragma once

class MyClass
{
private:
	int m_Value;
	int m_Other;

public:
	MyClass( int, int );
	~MyClass(void);

	int get_Value();
	int get_Other();
};
```

MyClass.cpp

```
#include ".\myclass.h"

MyClass::MyClass( int x, int y ) : m_Value( x ), m_Other( y )
{
}

MyClass::~MyClass(void)
{
}

int MyClass::get_Value()
{
	return m_Value;
}

int MyClass::get_Other()
{
	return m_Other;
}
```

Main.cpp

```
#include <iostream>
using namespace std;
```



```
#include "MyClass.h"

int main()
{
	MyClass *m_Obj = new MyClass( 1, 2 );

	cout << m_Obj->get_Value() << endl;
	cout << m_Obj->get_Other() << endl;

	delete m_Obj;

	return 0;
}
```

9. Value Types and Functions

Often we need to pass objects to functions. While it is possible to pass objects by value, this is a complicated and expensive process. Value type objects should be passed to functions by reference.

MyClass.h

```
#pragma once

class MyClass
{
private:
	int myValue;
public:
	MyClass( int );
	~MyClass(void);
	void set_Value( int );
	int get_Value();
};
```

MyClass.cpp

```
#include ".\myclass.h"

MyClass::MyClass( int x )
{
	myValue = x;
}

MyClass::~MyClass(void)
{
}

void MyClass::set_Value( int x )
{
	myValue = x;
}

int MyClass::get_Value()
{
	return myValue;
}
```

Main.cpp

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

void Change_Value( MyClass & );
```



```
int main()
{
      MyClass m_Obj( 5 );
      Change_Value( m_Obj );

      cout << m_Obj.get_Value() << endl;

      return 0;
}

void Change_Value( MyClass &pObj )
{
      pObj.set_Value( 6 );
}
```


10. Reference Types and Functions

Reference type objects must be passed to functions by pointer.

Main.cpp

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

void Change_Value( MyClass * );

int main()
{
	MyClass *m_Obj = new MyClass( 5 );
	Change_Value( m_Obj );

	cout << m_Obj->get_Value() << endl;

	delete m_Obj;

	return 0;
}

void Change_Value( MyClass *pObj )
{
	pObj->set_Value( 6 );
}
```


11. Static Members

A class can contain static data and methods. A member declared as static tells the compiler to create only one copy of the static member for all instances of the class, regardless of how many instances are created. Static members are shared by all objects of the class. The declaration of a static data member in a class does not define it. You must provide its definition elsewhere by redeclaring it elsewhere.

A static member function can be accessed as independently of an instance of the class as in this example. Of course, static functions cannot access non-static member data, i.e. instance data members. In this example, the use of static members allows the class to keep track of how many instances have been created of itself.

MyClass.h

```
#include <iostream>
using namespace std;

class MyClass
{
private:
	static int count;

public:
	// Instance methods can refer to static data.
	MyClass()
	{
		count++;
	}
	~MyClass()
	{
		count--;
	}
	// Static method refers to static data member.
	// Static methods cannot refer to instance members.
	static int get_count()
	{
		return count;
	}
};

// Initialize static member here.
int MyClass::count = 0;
```

Main.cpp

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

int main()
{
```



```
        MyClass a;
        MyClass b;
        MyClass c;

        // Static methods can be called from the class.
        cout << MyClass::get_count() << endl;

        // Static methods can be called on an instance.
        cout << a.get_count() << endl;

        return 0;
}
```

LAB 9

Create a Stock class that calculates the stock's dividend yield.

Stock.h

```
#pragma once

class Stock
{
private:
    double price;
    double div_amt;
    double div_yield;

    void calc_yield()
    {
        div_yield = div_amt / price;
    }
public:
      void set_div_amt( double d )
      {
            div_amt = d;
      }
    void set_price( double p )
    {
        price = p;
    }
    double get_price()
    {
        return price;
    }
    double get_div_amt()
    {
        return div_amt;
    }
    double get_div_yield()
    {
        calc_yield();
        return div_yield;
    }
};
```

Display.h

```
#pragma once
#include <iostream>
#include "Stock.h"
using namespace std;

class Display
```



```
{
public:
	static void print_data( Stock &x )
	{
		cout << "Price\tDiv_Amt\tDiv_Yield\n";
		cout << x.get_price() << "\t" << x.get_div_amt() << "\t" <<
			x.get_div_yield() << "\n\n\n";
	}
	static void print_data( Stock *x )
	{
		cout << "Price\tDiv_Amt\tDiv_Yield\n";
		cout << x->get_price() << "\t" << x->get_div_amt() << "\t" <<
			x->get_div_yield() << "\n\n\n";
	}
};
```

Main.cpp

```
#include "Stock.h"
#include "Display.h"

int main()
{
	Stock a;
	a.set_price( 100.0 );
	a.set_div_amt( 5.0 );

	Display::print_data( a );

	return 0;
}
```

12. Overloaded Methods

Class methods can be overloaded just like C-style functions. Constructors are very often overloaded as in this example.

MyClass.h

```
#pragma once

class MyClass
{
private:
	int m_Value;
public:
	MyClass();
	MyClass( int );
};
```

MyClass.cpp

```
#include "MyClass.h"

MyClass::MyClass()
{
}

MyClass::MyClass( int x ) : m_Value( x )
{
}
```

Main.cpp

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

int main()
{
	MyClass a;
	MyClass b( 3 );
	return 0;
}
```


13. Operator Overloading

Just like functions, operators, both unary, like ++ and binary like +, can be overloaded. That is, they can have different meaning based upon the way in which they are used or called. We can define overloaded operators and their meanings in classes with an operator function. When a binary operator is used, the object to the left of the operator invokes its operator function, which is passed the object to the right. This example illustrates overloading the + operator.

MyClass.h

```
#pragma once

class MyClass
{
private:
      int m_Value;

public:
    MyClass( int );
    int get_Value();
    int operator+( MyClass & );
};
```

MyClass.cpp

```
#include "MyClass.h"

MyClass::MyClass( int x )
{
      m_Value = x;
}

int MyClass::get_Value()
{
      return m_Value;
}

int MyClass::operator+( MyClass &m_Obj )
{
      int sum = m_Value + m_Obj.get_Value();
      return sum;
}
```

Main.cpp

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



```
int main()
{
	MyClass m_Obj_1( 2 );
	MyClass m_Obj_2( 3 );

	int value = m_Obj_1 + m_Obj_2;

	cout << value << endl;

	return 0;
}
```

This example illustrates overloading the [] operator.

Bar.h

```
#pragma once
using namespace std;

enum Type
{
	Open,High,Low,Close
};

class Bar
{
private:
	double open;
	double high;
	double low;
	double close;

public:
	double &operator[]( Type t )
	{
		switch( t )
		{
			case Open:
				return open;
			case High:
				return high;
			case Low:
				return low;
			default:
				return close;
		};
	}
};
```


Main.cpp

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

int main()
{
	Bar a;
	a[ Open ] = 50.0;
	a[ High ] = 51.0;
	a[ Low ] = 49.0;
	a[ Close ] = 49.5;

	cout << a[ High ] << endl;

	return 0;
}
```

This example illustrates overloading the () operator. Notice how the () is used to call the function. In this case, we say that a is a callable object.

```
#include <iostream>
using namespace std;

class MyClass
{
public:
	int operator()()
	{
		return 7;
	}
};

int main()
{
	MyClass a;
	cout << a() << endl;

	return 0;
}
```


14. Copy Constructor

A copy constructor is called whenever a copy of an object is created. This happens when an instance is declared and initialized from another object.

MyClass.h

```
#include <iostream>
using namespace std;

class MyClass
{
private:
      int value;
public:
      MyClass( int v ) : value( v )
      {
          cout << "Constructor running. " << endl;
      }
      MyClass( const MyClass &o )
      {
          cout << "Copy constructor running. " << endl;
      }
      ~MyClass()
      {
          cout << "Destructor running. " << endl;
      }
      int get_value()
      {
          return value;
      }
};
```

Main.cpp

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

int main()
{
   MyClass a( 3 );  // Constructor builds a.
   MyClass b( a );  // Copy constructor builds b.
   MyClass c = a;   // Copy constructor initializes in declaration.
   a = c;           // Assignment, no constructor or copy constructor.
}
```

The copy constructor will also run when a value type object is passed by value to a function. This is why we should only pass value type objects by reference. The copy constructor may also run when an object is returned by a function. Consider the following two functions:


```
MyClass Foo()
{
      return MyClass( 4 );
}

MyClass Bar()
{
      MyClass x( 5 );
      return x;
}
```

Main.cpp

```
int main()
{
   MyClass a = Foo();  // Constructor runs.  Destructor runs.
   MyClass &b = Foo(); // Constructor runs.  Destructor runs.
   MyClass c = Bar();  // Constructor runs.  Copy constructor runs.
                       // Destructor runs twice.
   MyClass &d = Bar(); // Constructor runs.  Copy constructor runs.
                       // Destructor runs twice.
}
```

Notice the advantage of using nameless object return type in Foo() to avoid calling the copy constructor. Consider also the following cases:

```
MyClass &Foo()
{
      return MyClass( 4 );
}

MyClass &Bar()
{
      MyClass x( 5 );
      return x;
}
```

Main.cpp

```
int main()
{
   MyClass &a = Foo(); // Constructor runs.  Destructor runs.
   MyClass &b = Bar(); // Constructor runs.  Destructor runs.
}
```


Returning a reference does not call the copy constructor, nor does it call the destructor when the instance goes out of scope when the function returns the instance.

15. Member Function Pointers

As we have seen, C++ has function pointers. But, an ordinary function pointer cannot point to a member function. Instead, we have to use a member function pointer.

```
class MyClass
{
public:
	int add( int i )
	{
		return i + 1;
	}
};
```

```
int main()
{
	int ( MyClass::*method_ptr )( int );
	method_ptr = &MyClass::add;

	MyClass a;
	cout << ( a.*method_ptr )( 2 ) << endl;

	MyClass *b = new MyClass;
	cout << ( b->*method_ptr )( 3 ) << endl;
	delete b;

	return 0;
}
```


LAB 10

Create an Instrument class that has a constructor and destructor, a static and instance method, a copy constructor and an overloaded operator. Create an instance of the class on the stack and the heap and demonstrate each method.

Solution:

Main.cpp

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

int main()
{
	Instrument::print_type();

	Instrument m_Instr1( 1 );
	cout << m_Instr1.get_value() << endl;

	Instrument *m_Instr2 = new Instrument( 2 );

	Instrument m_Instr3 = ( *m_Instr2 ) + m_Instr1;
	Instrument m_Instr4 = m_Instr3;

	delete m_Instr2;

	return 0;
}
```

```
#include <iostream>
using namespace std;

class Instrument
{
private:
	int value;
public:
	Instrument( int v ) : value( v )
	{
		cout << "Constructor running on " << value << endl;
	}

	Instrument( Instrument &i )
	{
		value = i.get_value() + 1;
		cout << "Copy constructor running on " << value << endl;
	}

	~Instrument()
	{
```



```
            cout << "Destructor running on " << value << endl;
      }

      Instrument operator+( Instrument &i )
      {
            return Instrument( value + i.get_value() );
      }

      int get_value() { return value; }

      static void print_type()
      {
            cout << "Instrument class!" << endl;
      }
};
```

16. Inheritance

Inheritance is the process by which one object can acquire the properties—the data and methods—of another object. If, for example, we consider a call, we say that a call is an option and therefore contains all of the data and methods that are common to all options. An option is an instrument and therefore should contain all the data and methods that are common to all instruments. A stock class may also inherit from the instrument class. A child, or derived, class will have all the data and functionality of the class it inherits from, called the parent or base class.

It is important to clarify the three levels of scope with respect to objects—private, public, and protected. Private members are visible, or accessible, only from within the definitions of class member functions. Public members are accessible by anywhere. Protected methods are accessible only from within the methods defined in the class itself and the methods defined in classes which inherit from the class.

17. Polymorphism

Polymorphism (meaning “many forms”) allows one interface, or set of public methods, to be used for a general class of functionality. The specific action is determined at the derived class level. The instrument class discussed earlier may contain a method called EnterOrder(). So, through inheritance both a stock and an option will have an enter order method, but the course of action to send a stock order to the NYSE is much different that sending on option order to the CBOE. So, at the derived class level, the stock and option class each implement the EnterOrder() method differently. To the user of an instance of either the stock or option class, however, the method call is the same—EnterOrder().

Instrument.h

```
#pragma once

class Instrument
{
protected:
      char *symbol;

public:
      Instrument( char * );
      char *get_Symbol();
};
```

Instrument.cpp

```
#include "Instrument.h"

Instrument::Instrument( char *s )
{
      symbol = s;
}

char *Instrument::get_Symbol()
{
      return symbol;
}
```

The stock class inherits from the Instrument class.

Stock.h

```
#pragma once
#include "Instrument.h"

class Stock : public Instrument
{
```



```
protected:
      double price;

public:
      Stock( char *, double );
      double get_Price();
};
```

Stock.cpp

```
#include "Stock.h"

Stock::Stock( char *s, double p ) : Instrument( s )
{
      price = p;
}

double Stock::get_Price()
{
      return price;
}
```

Main.cpp

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

int main()
{
      Stock *m_Stock = new Stock( "IBM", 52.45 );

      cout << m_Stock->get_Symbol() << endl;
      cout << m_Stock->get_Price() << endl;

      delete m_Stock;

      return 0;
}
```

18. Virtual Methods

A virtual function is one that can be redefined in a derived class. When a virtual function is redefined in a derived class, we say that the method has been overridden. A class that declares or inherits a virtual method is called a polymorphic class.

Base.h

```
#pragma once
#include <iostream>
using namespace std;

class Base
{
public:
	virtual void Hello()
	{
		cout << "Hello from the Base class!" << endl;
	}
};
```

Derived.h

```
#pragma once
#include "Base.h"
#include <iostream>
using namespace std;

class Derived : public Base
{
public:
	void Hello()
	{
		cout << "Hello from the Derived class!" << endl;
	}
};
```

Main.cpp

```
#include "Base.h"
#include "Derived.h"
#include <iostream>
using namespace std;

int main()
{
	Base myObj_1;
	Derived myObj_2;

	Base *myPtr;
```



```
      myPtr = &myObj_1;
      myPtr->Hello();

      myPtr = &myObj_2;
      myPtr->Hello();

      return 0;
}
```

Virtual destructors work differently. The difference between a destructor and other methods is that if a regular method overrides a base class definition, only the derived class definition gets executed. In the case of a destructor, however, both the derived and base class definitions will be executed. By declaring the base class destructor as virtual, both definitions will be called in order. Without virtual, only the base class definition would run in the case where a base class pointer is pointing to an instance of the derived class.

```
#include <iostream>
using namespace std;

class MyBase
{
public:
    virtual ~MyBase()
    {
        cout << "Base class destructor running." << endl;
    }
};

class MyDerived : public MyBase
{
public:
    ~MyDerived()
    {
        cout << "Derived class destructor running." << endl;
    }
};

void main()
{
      // Without virtual, the derived class destructor will not run.
      MyBase *a = new MyDerived();
      delete a;
}
```


19. Pure Virtual Functions and Abstract Classes

A pure virtual function is a virtual function for which no definition is provided at the base class level. The function must be overridden in a derived class. The presence of a pure virtual function makes the entire class pure virtual, or abstract. You cannot instantiate an abstract class. If a pure virtual function is not overridden in the derived class, then the derived class itself becomes abstract.

PureBase.h

```
#pragma once
#include <iostream>
using namespace std;

class PureBase
{
public:
	virtual void Hello() = 0;
};
```

Derived.h

```
#pragma once
#include "PureBase.h"
#include <iostream>
using namespace std;

class Derived : public PureBase
{
public:
	// Cannot create an instance of Derived unless we
	// override the pure virtual function in the base class.
	void Hello()
	{
		cout << "Hello from the Derived class!" << endl;
	}
};
```

Main.cpp

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

int main()
{
	Derived myObj;
	return 0;
}
```


We use the term interface to mean the collection of public methods of an object. However, an interface can also mean an abstract class, one that contains only pure virtual methods. This class then defines a collection of public methods for classes that inherit from it.

LAB 11

Create an Option and Bond class that inherit from a base Instrument class. Demonstrate polymorphism by overriding a base class method in the derived classes.

Solution:

Main.cpp

```
#include <iostream>
#include "Bond.h"
#include "Option.h"

using namespace std;

int main()
{
     Option o;
     Bond b;

     cout << o.get_value() << endl;
     cout << b.get_value() << endl;

     return 0;
}
```

```
class Instrument
{
protected:

     double value;
     virtual void calc_value() = 0;

public:
     double get_value()
     {
          calc_value();
          return value;
     }
};
```

```
#include "Instrument.h"

class Bond : public Instrument
{
protected:
     void calc_value()
     {
          value = 3.00;
     }
};
```



```
#include "Instrument.h"

class Option : public Instrument
{
protected:
     void calc_value()
     {
           value = 4.00;
     }
};
```

20. Casting

In addition to the C-style casting we saw earlier, C++ also contains four cast operators: dynamic_cast, const_cast, reinterpret_cast, and static_cast. Dynamic_cast performs a run-time cast that verifies at run-time the validity of the cast. A base class pointer can point to an instance of a derived class, and a dynamic_cast enables casting relative to an inheritance hierarchy. A static_cast., on the other hand, performs a cast for any standard conversion and no run-time checks are performed.

```
#include <iostream>
using namespace std;

class Instrument
{
public:
	virtual ~Instrument() {}
	virtual void Say_Hello()
	{
		cout << "Hello from the Instrument class." << endl;
	}
};

class Stock : public Instrument
{
public:
	Stock() {}
	void Say_Hello()
	{
		cout << "Hello from Stock." << endl;
	}
};

int main()
{
	Instrument *x = new Stock();
	Stock *y = dynamic_cast< Stock * >( x );

	y->Say_Hello();  // The dynamic_cast worked!

	delete x;

	return 0;
}
```

Run-time Type Information (RTTI) is what makes the dynamic_cast work. RTTI is what keeps information about an object's type in memory at runtime. This is the C++ version of what is more commonly known as reflection. RTTI is only available for polymorphic classes (i.e. they have at least one virtual method, which should not be a problem since base classes must have a virtual destructor as previously discussed.

21. Friends

Declaring one function or class as a friend of another class gives the friend access to the private member variables of the class.

MyClass.h

```
#pragma once

class MyClass
{
friend class MyOther;

private:
	int m_Value;
public:
	MyClass( int v ) : m_Value( v ) {}
	int get_Value() { return m_Value; }
};
```

MyOther.h

```
#pragma once
#include "MyClass.h"

class MyOther
{
private:
	MyClass &m_Obj;
public:
	MyOther( MyClass &m ) : m_Obj( m ) {}
	int Test()
	{
		return m_Obj.m_Value;
	}
};
```

Main.cpp

```
#include "MyClass.h"
#include "MyOther.h"
#include <iostream>
using namespace std;

int main()
{
	MyClass m_Obj( 5 );
	MyOther m_Other( m_Obj );
	cout << m_Other.Test() << endl;
	return 0;
}
```


22. Exception Class

Exception handling is a structured means by which a program can handle and deal with run-time errors when they occur. A try…catch blocks perform exception handling. The lines of code you think may cause run-time exceptions should be placed inside the try block. If an error occurs (or is thrown), it will be caught by the catch block. We can explicitly cause errors using the throw statement.

```
#include <iostream>
using namespace std;

double Divide( double, double );

int main()
{
	try
	{
		cout << Divide( 1, 0 ) << endl;
	}
	catch ( exception &myExc )
	{
		cout << myExc.what() << endl;
	}
	return 0;
}

double Divide( double x, double y )
{
	if ( y == 0 )
	{
		throw exception( "Divide by zero error!" );
	}
	else
	{
		return x / y;
	}
}
```

The **assert** function is also used to facilitate exception handling at *run-time* while debugging. If the argument evaluates to zero (i.e. false), the program is terminated. The exception message will include the line number. Asserting is a debugging tool. Asserts are not present in release builds. You use them to eliminate run-time errors without the use of try…catch blocks, which would be present in release builds.

```
#include <iostream>
#include <cassert>
using namespace std;

int main ()
{
	int a = 5;
	int *b = &a;
```



```
      int *c = NULL;

      assert( b != NULL );

      cout << *b << endl;

      assert( c != NULL );

      return 0;
}
```


23. File Streams

A stream is a consistent, logical interface to the various devices that comprise the computer, either consuming or producing data. A text stream is used with characters, while a binary stream is used with other types of data. The ofstream class enables us to stream data to a file. The ifstream class enables us to read data from a file.

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
	ofstream m_OutFile( "myMessage.txt", ios::out );
	m_OutFile << "Hello, World!" << endl;
	m_OutFile.close();

	ifstream m_InFile( "myMessage.txt", ios::in );
	char m_Line[ 20 ];
	m_InFile.getline( m_Line, 20 );
	cout << m_Line << endl;

	return 0;
}
```


LAB 12

Parsing .csv files in C++ is not easily done. Much of the difficulty revolves around recognizing the end-of-line character. By formatting the data with ‘,’ at the end of the line, this problem is solved.

Create a program that reads the given open-high-low-close data into a structure.

```
128.34,130.68,128.34,130.35,
128.93,129.95,128.37,129.79,
128.5,129.97,128.49,128.5,
126.73,128.8,126.44,128.45,
125.99,128.22,125.8,127.68,
124.83,125.84,123.58,123.9,
124.26,124.46,122.82,123.72,
125.57,125.86,124.13,124.13,
126.37,127.1,124.67,125.28,
127.75,128.22,126.46,127.96,
```

```
#include <iostream>
#include <fstream>
#include <cstring>
using namespace std;

struct Bar
{
	double Open;
	double High;
	double Low;
	double Close;
};

void main()
{
	ifstream m_File( "C:\\temp\\IBM.csv", ios::in );

	int index = 0;
	int count = 0;

	char buffer[ 128 ];
	Bar m_Data[ 11 ];

	while( m_File.getline( buffer, 128, ',' ) )
	{
		switch ( count )
		{
			case 0:
				m_Data[ index ].Open = atof( buffer );
				count++;
				break;
			case 1:
				m_Data[ index ].High = atof( buffer );
				count++;
```



```
                        break;
                case 2:
                        m_Data[ index ].Low = atof( buffer );
                        count++;
                        break;
                case 3:
                        m_Data[ index ].Close = atof( buffer );
                        count = 0;
                        index++;
            }
        }
        m_File.close();

        for ( int i = 0; i < 10; i++ )
        {
            cout << m_Data[ i ].Open << '\t';
            cout << m_Data[ i ].High << '\t';
            cout << m_Data[ i ].Low << '\t';
            cout << m_Data[ i ].Close << '\n';
        }
}
```

CHAPTER 4: TEMPLATES AND COLLECTIONS

1. Template Functions

A template, sometimes called generic, function is one that can overload itself. Its definition can be applied to various data types. The data type that the function will substitute for T, the type placeholder in this example, is passed to it as a parameter. The compiler will generate as many different versions of the template function as are necessary to handle the various ways that your program call the function.

```
#include <iostream>
using namespace std;

template < class T >
T Add( T, T );

int main()
{
	double a = 2.2;
	double b = 3.7;

	double c = Add( a, b );

	cout << c << endl;

	int d = 4;
	int e = 5;

	int f = Add( d, e );

	cout << f << endl;

	return 0;
}

template < class T >
T Add( T x, T y )
{
	T z;
	z = x + y;
	return z;
}
```


2. Template Classes

A template, or generic, class extends the template function concept and applies it to an entire class. In the example below, the class MyClass contains a data member of type T. Notice that in the main file, when an instance of the template class is created, the type T for m_Obj_1 is explicitly defined as int with the MyClass< int > m_Obj_1(4); syntax. In m_Obj_2, the type is double.

MyClass.h

```
#pragma once
#include <iostream>
using namespace std;

template < class T >
class MyClass
{
private:

	T m_Value;

public:

	MyClass( T );
	~MyClass(void);

	T get_Value();
};

template < class T >
MyClass< T >::MyClass( T x ) : m_Value ( x )
{
}

template < class T >
MyClass< T >::~MyClass(void)
{
	cout << "MyClass object is dying!" << endl;
}

template < class T >
T MyClass< T >::get_Value()
{
	return m_Value;
}
```

Main.cpp

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

int main()
```



```
{
	MyClass< int > m_Obj_1( 4 );
	cout << m_Obj_1.get_Value() << endl;

	MyClass< double > *m_Obj_2 = new MyClass< double >( 3.14159 );
	cout << m_Obj_2->get_Value() << endl;
	delete m_Obj_2;

	return 0;
}
```

3. Another Template Class Example

In this slightly more complex example, the type T is a pointer to an instance of the Element class.

Element.h

```
#include <iostream>
using namespace std;

class Element
{
private:
	int m_Data;

public:
	Element( int x ) : m_Data( x ) {}

	Element( const Element &a )
	{
		cout << "Copy constructor is running!" << endl;
		m_Data = a.m_Data;
	}

	~Element(void)
	{
		cout << "Element " << m_Data << " is dying!" << endl;
	}

	int get_Data()
	{
		return m_Data;
	}
};
```

Main.cpp

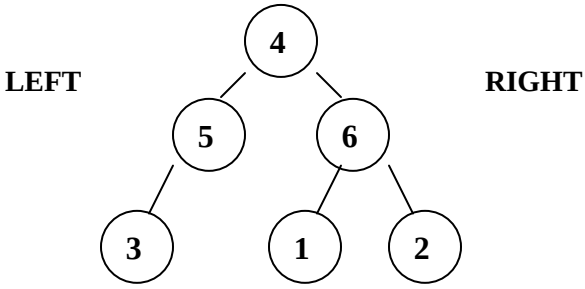
```
#include "MyClass.h"
#include "Element.h"
using namespace std;

int main()
{
 {
   MyClass< Element > m_Obj1( Element( 5 ) );
   cout << m_Obj1.get_Value().get_Data() << endl;
 }

 cout << "\n\nLet's try it as reference types.\n\n\n";

 {
   MyClass < Element * > *m_Obj2 = new MyClass < Element * >( new Element( 6 ));
   cout << m_Obj2->get_Value()->get_Data() << endl;
```



```
   delete m_Obj2->get_Value();
   delete m_Obj2;
 }
 return 0;
}
```


4. Trees

Managing groups of objects is a common problem in programming. One way to do it is to create a tree of self-referential objects. Consider the following tree of nodes, where each node has a value and a pointer to a left and right node.



Node.h

```
template< class T >
struct Node
{
	T Value;

	Node *Right;
	Node *Left;

	Node( T v ) : Value( v )
	{
		Right = NULL;
		Left = NULL;
	}
	~Node()
	{
		delete Right;
		delete Left;
	}
};
```

Main.cpp

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

template< class T >
void print_values( Node< T > *r )
{
   if( r != NULL )
   {
      cout << r->Value << endl;
      print_values( r->Right );
      print_values( r->Left );
```



```
   }
}

int main()
{
      Node< int > *root = new Node< int >( 4 );
      root->Left = new Node< int >( 5 );
      root->Right = new Node< int >( 6 );
      root->Left->Left = new Node< int >( 3 );
      root->Right->Right = new Node< int >( 2 );
      root->Right->Left = new Node< int >( 1 );

      print_values( root );
}
```

For an expanded example using the tree structure to maintain stock price data, see the Tree Class Example available at **www.benvanvliet.com**.

5. Function Objects

Function objects, or functors, are instances of classes that can be used with syntax similar to function calls. This is achieved by overriding operator() in their class. One use of function objects is callback functions (see section C++FA 1.2, 20). Unlike a simple function pointer, a functor enables the function to carry its own state.

```
#include <iostream>
using namespace std;

class MyClass
{
public:
  int operator()( int a )
  {
        return a + 10;
  }
} MyObject;

int main()
{
      MyClass x;
      cout << x( 2 )<< endl;

      // or

      cout << MyObject( 3 ) << endl;

      return 0;
}
```

The standard library provides standard definitions for several function objects in the <functional> library. These function objects are often used with STL algorithms and for comparison.

```
#include <iostream>
#include <functional>
#include <algorithm>
using namespace std;

int main ()
{
  int a[] = { 2, 4, 5, 1, 3 };

  sort( a, a + 5, greater< int >() );

  for( int i = 0; i < 5; i++ )
  {
      cout << a[ i ] << " ";
  }

  cout << endl;

  return 0;
```



```
}
```

Functors can be thought of as functions with data members. A functor is a class with data members that hold the state values and with an overload of the operator() which will execute the function. In this example, we create an array of pointers to functions and call them via a function pointer and the operator(). Of course, all the functions must have the same signature. You can also use function pointers and functors to implement callbacks with objects.

```
#include <iostream>
using namespace std;

template < class T >
class MyFunctor
{
private:
    int ( T::*f )( int, int );
    T *o;

public:
    MyFunctor( T *optr, int ( T::*fptr )( int, int ) )
    {
            o = optr;
            f = fptr;
    }
    virtual int operator()( int a, int b )
    {
            return ( *o.*f )( a, b );
    }
    virtual int fu( int a, int b )
    {
            return ( *o.*f )( a, b );
    }
};

class MyClass
{
public:
     int Add( int a, int b )
     {
         return a + b;
     }
};

int main()
{
      MyClass x;

      MyFunctor< MyClass > func1( &x, &MyClass::Add );
      MyFunctor< MyClass > func2( &x, &MyClass::Add );

      MyFunctor< MyClass > *v[] = { &func1, &func2 };

      cout << v[ 0 ]->fu( 5, 6 ) << endl;     // Call via function fu()
      cout << ( *v[ 1 ] )( 6, 7 ) << endl;    // Call via operator()
}
```


The primary benefit of function objects is faster running code. This is achieved because, unlike a function pointer, a function object can be inlined by the compiler. Here is a simple example that increments a number either as a function object:

```
class Plus_Functor
{
public:
	void operator()( int &x )
	{
		++x;
	}
};
```

or, as a C-style function:

```
void Plus_Function( int &x )
{
	++x;
}
```

Here is a simple For_Each() function:

```
template< class Pointer, class Function >
Function For_Each( Pointer begin, Pointer end, Function f )
{
	while( begin != end )
	{
		f( *begin );
		++begin;
	}
	return f;
}
```

Suppose we apply our For_Each() function to Plus_Functor and Plus_Function:

```
int main()
{
	int a[] = { 1, 2, 3, 4, 5 };

	For_Each( a, a + 5, Plus_Functor() );
	For_Each( a, a + 5, Plus_Function );

	cout << a[ 4 ] << endl;

	return 0;
}
```


Within For_Each call passing Plus_Functor(), the compiler will inline the function object because it is known at compile time. In the For_Each call passing Plus_Function(), the function pointer's value is not known at compile time.

6. Limits Library

The limits library contains, among other things, the numeric_limits template class, which describes arithmetic properties of C++'s native numerical types, including max and infinity constants.

```
#include <limits>

int x = numeric_limits< int >::max();
double y = numeric_limits< double >::infinity();
```

7. Standard Template Library

The Standard Template Library is a set of general-purpose template container classes, algorthims and iterators. Containers, or collections, are objects that hold and manage groups of other objects. Algorithms are functions that act on containers and iterators are essentially pointers to elements in a container. Here is a list of the containers in the STL.

Container	Description
Bitset	A set of bits.
Deque	A double-ended queue.
List	A doubly linked list.
Map	An associative container for pairs.
Multimap	An associative container for pairs.
Multiset	A set of non-unique elements.
priority_queue	A queue.
Queue	A first-in, first-out list.
Set	A set of unique elements.
Stack	A last-in, first-out list.
Vector	A dynamic array.

9. STL List Class

For sample code on the inner workings of a list class, see the List Class Example available at **www.benvanvliet.com**.

This example illustrates the use of a STL list class. In this code, notice the iterator and the use of the for loop.

```
#include <iostream>
#include <list>
using namespace std;

int main()
{
  list< int > *m_List = new list< int >;

  m_List->push_front( 2 );
  m_List->push_front( 4 );
  m_List->push_front( 3 );
  m_List->push_front( 1 );

  m_List->pop_back();

  list< int >::iterator m_Iter;
  for ( m_Iter = m_List->begin(); m_Iter != m_List->end(); m_Iter++ )
  {
        cout << *m_Iter << " ";
  }
  cout << endl;

  delete m_List;

  return 0;
}
```


10. STL Vector Class

This example illustrates the use of the commonly used vector class with a reverse_iterator just for fun. Notice also that the vector can be used with array-like, index-style notation.

```
#include <vector>
#include <iostream>
using namespace std;

int main()
{
	vector< int > m_Ints;
	cout << "Size: " << m_Ints.size() << endl;
	cout << "Capacity: " << m_Ints.capacity() << endl;

	m_Ints.push_back( 2 );
	m_Ints.push_back( 3 );
	m_Ints.push_back( 4 );

	cout << "Size: " << m_Ints.size() << endl;
	cout << "Capacity: " << m_Ints.capacity() << endl;

	return 0;
}
```

This example shows the use of a vector with index notation.

```
#include <iostream>
#include <vector>
using namespace std;

double Average( vector< double > & );

int main()
{
	vector< double > m_Data;
	for ( double i = 0; i < 100; i++ )
	{
		m_Data.push_back( i );
	}
	cout << Average( m_Data ) << endl;
	return 0;
}

double Average ( vector< double > &d )
{
	int z = d.size();
	double sum = 0;
	for ( int y = 0; y < z; y++ )
	{
		sum += d[ y ];
	}
	return sum / z;
}
```


11. STL String Class

The string class is an alternative to a character array. The advantage of the string class over a character array is that a string has several methods and overloaded operators that support concatenation, assignment, testing for equality, insertion, finding substrings, and others.

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
      string string1( "IBM" );
      string string2;
      string string3;

      string2 = string1;
      string3.assign( string1 );

      cout << "string1: " << string1 << endl;
      cout << "string2: " << string2 << endl;
      cout << "string3: " << string3 << endl;

      string2[ 0 ] = string3[ 2 ] = 'R';

      cout << "After modification of string2 and string3:" << endl;
      cout << "string1: " << string1 << endl;
      cout << "string2: " << string2 << endl;
      cout << "string3: " << string3 << endl;

      int j = string3.length();
      for ( int i = 0; i < j; i++ )
      {
            cout << string3.at( i );
      }

      cout << endl;

      string string4( string1 + "DP" );
      string string5;

      string3 += "ER";
      string1.append( "DF" );

      string5.append( string1, 4, string1.length() );

      cout << "After concatenation:" << endl;
      cout << "string1: " << string1 << endl;
      cout << "string2: " << string2 << endl;
      cout << "string3: " << string3 << endl;
      cout << "string4: " << string4 << endl;
      cout << "string5: " << string5 << endl;
}
```


12. STL Bitset Class

A bitset is a collection of bits. The class is like an array, since each bit can be accessed individually using the bracket notation. Bitsets support the bitwise operators and allow them to be extracted directly to and from streams.

```
#include <iostream>
#include <bitset>
using namespace std;

int main ()
{
      bitset< 4 > m_bits1;

      cout << m_bits1 << endl;                  // 0000
      m_bits1[ 0 ] = 1;                         // 0001
      m_bits1[ 2 ] = 1;                         // 0101
      cout << m_bits1 << endl;

      bitset< 4 > m_bits2( string( "1011" ) );

      cout << ( m_bits1 & m_bits2 ) << endl;    // Bitwise AND
      cout << ( m_bits1 | m_bits2 ) << endl;    // Bitwise OR
      cout << ( m_bits1 ^ m_bits2 ) << endl;    // Bitwise XOR
      cout << ( m_bits1 &= m_bits2 ) << endl;
      cout << ( m_bits1 ^= m_bits2 ) << endl;
      cout << ( m_bits1 |= m_bits2 ) << endl;

      cout << ( m_bits1 <<= 1 ) << endl;        // Shift Left
      cout << ( m_bits1 >>= 2 ) << endl;        // Shift Right

      cout << ( ~m_bits2 ) << endl;             // Bitwise NOT

      return 0;
}
```


13. Key and Value Pairs

A pair object contains a key, which should be unique, and a value. In this example, notice also the use of typedef, which allows you to create a new name for an existing data type.

```
pair < int, double > m_Pair( 5, 2.123e-6 );
cout << m_Pair.first << "  " << m_Pair.second << endl;

typedef pair< int, double > MyPairType;
MyPairType m_OtherPair( 1, 2.71828 );
cout << m_OtherPair.first << "  " << m_OtherPair.second << endl;
```


14. STL Hash_map

For sample code on the inner workings of a hash_map class, see the Hash_Map Class Example available at **www.benvanvliet.com**.

A hash_map is a dictionary-style, or associative, container. In a hash_map, unique keys are mapped with values. The key is the unique name of the value. Keys are hashed to locations in the underlying container and values can be retrieved using the key. Strictly speaking, the hash_map is not part of the STL. It is found in the extended standard library, stdext.

```
#include "Element.h"
#include <hash_map>
#include <iostream>
using namespace std;
using namespace stdext;

int main()
{
   hash_map < int, Element > m_HashMap;
   hash_map < int, Element >::iterator m_Iterator;

   typedef pair< int, Element > Int_Pair;
   for ( int i = 0; i < 5; i++ )
   {
         m_HashMap.insert( Int_Pair( i, Element( i * 10 ) ) );
   }

   cout << "New key and value pairs:" << endl;
   for ( m_Iterator = m_HashMap.begin();
         m_Iterator != m_HashMap.end();
         m_Iterator++ )
   {
      cout << m_Iterator->first << " " <<
              m_Iterator->second.get_Data() << endl;
   }

   cout << m_HashMap.find(3)->second.get_Data() << endl;
}
```

Here is another example. In this example, the value is an object, not just an integer.

```
#include <iostream>
#include <hash_map>
#include "Element.h"
using namespace std;
using namespace stdext;

int main()
{
   hash_map < int, Element * > m_Table;
   hash_map < int, Element * >::iterator m_Iter;
```



```
   typedef pair< int, Element * > Int_Pair;
   for ( int i = 0; i < 5; i++ )
   {
         m_Table.insert( Int_Pair( i, new Element( i * 10 ) ) );
   }

   for ( m_Iter = m_Table.begin(); m_Iter != m_Table.end(); m_Iter++ )
   {
      cout << m_Iter->first << " " << m_Iter->second->get_Data() << endl;
   }

   cout << m_Table.find( 3 )->second->get_Data() << endl;

   for ( m_Iter = m_Table.begin(); m_Iter != m_Table.end(); m_Iter++ )
   {
      delete m_Iter->second;
   }
}
```

15. STL Algorithms

The STL algorithms is a set of functions that perform common operations on STL containers. There are 50 or such algorithms in the STL. The definitions of the different algorithms is available anywhere you find C++ documentation as well as on the internet.

Commonly used algorithms in quantitative finance:			
for_each	swap	sort	min
find	fill	upper_bound	max
search	remove	merge	replace
copy	random_shuffle	includes	transform

16. For_each Algorithm

This example shows the use of the for_each algorithm used for applying a function to a range of elements in a container.

```
#include <vector>
#include <iostream>
#include <algorithm>
using namespace std;

template < class T >
class MultiplyObject
{
private:
   T m_Value;

public:
   MultiplyObject( T &v ) : m_Value ( v ) {}

   void operator()( T &m_Element )
   {
      m_Element *= m_Value;
   }
};

int main()
{
   vector <int> m_Vector;
   vector <int>::iterator m_Iter;

   for ( int i = 10; i <= 100; i += 10 )
   {
      m_Vector.push_back( i );
   }

   int a = -2;

   for_each( m_Vector.begin(), m_Vector.end(), MultiplyObject< int > ( a ));

   cout << "New vector values: ";
   for ( m_Iter = m_Vector.begin(); m_Iter != m_Vector.end(); m_Iter++ )
   {
      cout << *m_Iter << " ";
   }
   cout << endl;
}
```


17. Find Algorithm

The find algorithm returns an iterator to the first element in the range that compares equal to value, or last if not found.

```
#include <list>
#include <algorithm>
#include <iostream>
using namespace std;

int main()
{
	list <int> m_List;
	list <int>::iterator m_Result;

	for ( int i = 10; i <= 100; i += 10 )
	{
		m_List.push_back( i );
	}

	m_Result = find( m_List.begin(), m_List.end(), 60 );

	if  ( m_Result == m_List.end( ) )
	{
		cout << "Not Found." << endl;
	}
	else
	{
		cout << "Found: " << *m_Result << endl;
	}
}
```


18. Remove Algorithm

The remove algorithm removes elements from a range in the container where the value is equal to a specified value.

```
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

int main()
{
	vector<int> m_vector;
	vector<int>::iterator m_Iter, m_End;

	for ( int i = 0; i <= 9; i++ )
	{
		m_vector.push_back( i );
	}
	cout << m_vector.size() << endl;

	m_End = remove( m_vector.begin(), m_vector.end(), 7 );

	m_vector.erase( m_End, m_vector.end() );

	cout << m_vector.size() << endl;

	for ( m_Iter = m_vector.begin();
		m_Iter != m_vector.end();
		m_Iter++ )
	{
		cout << *m_Iter << " ";
	}
	cout << endl;
	return 0;
}
```


CHAPTER 5: DESIGN PATTERNS

Design patterns are reusable solutions to problems commonly encountered in software design. A design pattern is a template for how to solve a particular problem which can be used in different situations. They usually define relationships and interactions between classes. The well-known Gang of Four computer scientists defined 23 design patterns. Today there are hundreds of patterns. Here are some relevant ones;

Name	Description
Abstract Factory	Defines an interface for creating families of related objects.
Builder	Separates the construction and representation of complex objects.
Factory Method	Defines an interface for creating objects, while deferring actual instantiation to subclasses.
Prototype	Creates new objects by copying an existing prototypical object.
Singleton	Ensures that only one instance of a class can created.
Adapter	Converts the interface of a class into another interface by wrapping it.
Bridge	Separates an abstraction from its implementation so the two can change independent of one another.
Composite	Combines objects so clients can treat individual objects and combinations the same way.
Decorator	Adds (or overrides) additional behavior to an existing object dynamically, while maintaining its original interface. Thus, it is an alternative to subclassing.
Facade	Provides for a simpler interface to a complex set of interfaces in a subsystem.
Flyweight	Uses sharing to reduce the cost of supporting large numbers of similar objects.
Proxy	Provides a placeholder for another object in order to control access to it.
Chain of Responsibility	Delegates handling of requests to a hierarchy of objects.
Command	Encapsulates requests within objects, to enable their parameterization.
Interpreter	Interprets code written in a specialized, ad hoc language.
Iterator	Enables sequential access to the elements of a collection or aggregation of objects.
Mediator	Defines a class that encapsulates how different objects interact, so that they do not refer to each other directly. This promotes loose coupling.
Memento	Stores an object's state so that it can be restored later if necessary.
Observer	Publish/subscribe pattern define a one-to-many relationship between objects so that when one object's state changes, all subscribers are notified.
State	Allows an object to change its behavior when its internal state changes.
Strategy	Provides for run time selection from a family of algorithms.
Template Method	Define a skeleton of an algorithm, deferring certain steps to subclasses.
Visitor	Separates an algorithm into its own object from the objects that use it.

1. Factory

The factory method pattern handles the problem of creating objects without specification as to the exact class to be created, by defining a separate method for object creation. More generally, a factory method can refer to any method whose main purpose is to create objects.

Class_A.h

```
#include <iostream>
using namespace std;

////////////////////////////////////////////////

class Class_A
{
public:
	virtual void Display() = 0;
};

////////////////////////////////////////////////

class Class_A1 : public Class_A
{
public:
	void Display()
	{
		cout << "Class A1" << endl;
	}
};

////////////////////////////////////////////////

class Class_A2 : public Class_A
{
public:
	void Display()
	{
		cout << "Class A2" << endl;
	}
};

////////////////////////////////////////////////
```

Main.cpp

```
#include <iostream>
#include “Class_A.h”

using namespace std;

class Factory
{
```

```
public:
      static Class_A *get_object( int a )
      {
            switch ( a )
            {
            case 1:
                  return new Class_A1;
            case 2:
                  return new Class_A2;
            default:
                  return NULL;
            }
      }
};

int main()
{
      int m;
      cin >> m;

      Class_A *x = Factory::get_object( m );
      x->Display();

      Class_A *y = Factory::get_object( 2 );
      y->Display();

      delete x;
      delete y;

      return 0;
}
```


2. Singleton

The singleton pattern restricts instantiation of a class to a single object.

MyClass.h

```
class MyClass
{
private:
      int value;

      static MyClass *inst;

      MyClass() {}
      MyClass( const MyClass & ) {}

public:
      static MyClass *create_instance()
      {
            if ( inst == NULL )
            {
                  inst = new MyClass();
            }
            return inst;
      }
      void set_value( int v )
      {
            value = v;
      }
      int get_value()
      {
            return value;
      }
};
MyClass *MyClass::inst = 0;
```

Main.cpp

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

int main()
{
      MyClass *a = MyClass::create_instance();
      a->set_value( 4 );

      MyClass *b = MyClass::create_instance();
      cout << b->get_value() << endl;
}
```


3. Observer

The observer, or publish/subscribe, design pattern monitors the state of an object. This pattern is mainly used to implement event handling systems which are common in real-time finance.

Subject.h

```
#pragma once
#include <ctime>
#include <cmath>
#include "Observer.h"

class Subject
{
private:
   int price;
   Observer *m_Observer;

   void sleep( unsigned int m )
   {
      clock_t goal = m + clock();
      while ( goal > clock() );
   }

public:
   void add_observer( Observer &o )
   {
      m_Observer = &o;
      price = 1250;
   }

   int get_price()
   {
      return price;
   }

   void start()
   {
      while ( true )
      {
         sleep(-1000.0 * log((double)( rand() % 5000 + 1 ) / 5000.0 ));
         if ( rand() % 2 + 1 > 1 )
         {
            price++;
         }
         else
         {
            price--;
         }
         m_Observer->price_changed( this );
      }
   }
};
```


Observer.h

Notice in this file the use of a forward declaration, which declares a class type name prior to the definition of the class. The is necessary because the declaration of Subject occurs inside the Observer before the Subject is declared.

```
class Subject;  // Forward declaration.

class Observer
{
public:
      void ( *MyFunctionPtr )( Subject * );

      void price_changed( Subject *s )
      {
            MyFunctionPtr( s );
      }
};
```

Main.cpp

```
#include <iostream>
#include "Observer.h"
#include "Subject.h"
using namespace std;

void get_new_data( Subject * );

int main()
{
      Subject a;
      Observer o;
      o.MyFunctionPtr = get_new_data;

      a.add_observer( o );

      a.start();
}

void get_new_data( Subject *s )
{
      system( "cls" );
      cout << s->get_price() << endl;
}
```


4. Smart Pointers

Smart pointers are in fact not pointers at all, they are objects that look and feel like pointers because they have the same interface that pointers do. That is, they overload pointer operations like dereferencing (operator*()) and indirection (operator->()). (An object that looks like something else is called a proxy object.) Because the most common bugs in C++ are related to pointers and dynamic memory management—dangling pointers, memory leaks, allocation failures—having a smart pointer automatically take care of these things is a big help. The simplest example of a smart pointer is auto_ptr, which is included in the standard C++ library in the <memory.h>. Here is a simple smart_ptr implementation:

smart_ptr.h

```
#pragma once

template< class T >
class smart_ptr
{
private:
	T *ptr;

public:
	smart_ptr( T *p = 0 ) : ptr( p )
	{
	}

	~smart_ptr()
	{
		delete ptr;
	}

	T &operator*()
	{
		return *ptr;
	}

	T *operator->()
	{
		return ptr;
	}
};
```

Main.cpp

```
#include <iostream>
#include "MyClass.h"
#include "smart_ptr.h"

using namespace std;

int main()
```



```
{
      MyClass *p = new MyClass( 5 );
      cout << p->get_value() << endl;
      delete p;

      smart_ptr< MyClass > m( new MyClass( 4 ) );
      cout << m->get_value() << endl;

      return 0;
}
```

5. Delegates

C++ does not have delegates, which are object-oriented function pointers, though it does have functors. Delegates can be used to simplify some design patterns, like the Observer. Notice in this example, created by Bohan Liu, that what appears to be a function pointer, m_Del, is in fact an instance of the Delegate class.

Delegate.h

```
template< class ClassType, class RetType, class ArgType >
class Delegate
{
	typedef RetType ( ClassType::*Function )( ArgType );

private:
	ClassType *m_Obj;
	Function   m_Func;

public:
	Delegate( ClassType *a, Function f ) : m_Obj( a ), m_Func( f ) {}

	RetType operator()( ArgType arg )
	{
		return ( m_Obj->*m_Func )( arg );
	}
};
```

MyClass.h

```
class MyClass
{
private:
	int m_Val;

public:
	MyClass( int a ) : m_Val( a ) {}

	int add( int i )
	{
		return m_Val + i;
	}
};
```

Main.cpp

```
#include <iostream>
#include "MyClass.h"
#include "Delegate.h"
using namespace std;
```



```
int main()
{
	MyClass *m_Obj = new MyClass( 4 );

	Delegate< MyClass, int, int > m_Del( m_Obj, &MyClass::add );

	int x = m_Del( 2 );
	cout << x << endl;

	delete m_Obj;

	return 0;
}
```

Delegates enable something called events. For sample code on the inner workings of an event class, see the Event Class Example available at **www.benvanvliet.com**.

CHAPTER 6: TEMPLATE METAPROGRAMMING

Template meta-programming (TMP) uses C++ templates to perform computations and code optimizations at compile-time. After compilation, computations that have already been made can thus be left out of the executable code. TMP requires that a template be defined and instantiated. The template definition describes the generic form of the generated source code, and the instantiation causes a specific set of source code to be generated from the generic form.

Though TMP syntax looks very different from regular C++ code, it has practical uses. One reason to use TMP is to optimize performance. Doing something once at compile-time is faster than doing it repeatedly at run-time. For example, having the compiler unroll loops at compile-time to eliminate jumps and loop count decrements at run-time.

TMP is sometimes called programming-with-types. Using types for calculation enables the use of type-inference rules. The outputs of the template types include compile-time constants, data structures, and complete functions. The variables in TMP are not really variables since their values can not be changed at run-time, though you can add some named values that you use like ordinary variables. What this means that it is necessary to use compile-time recursion rather than run-time iteration. In TMP, integers are usually stored as an enumeration:

```
#include <iostream>
using namespace std;

struct MyStruct
{
   enum
   {
      value = 2                  // faster
   };
   static int const v = 4;       // slower
};

int main()
{
      int x = MyStruct::value;
      cout << x << endl;

      return 0;
}
```

Here, we are storing the *int* so that it can be accessed with the name value. We could otherwise use the *static int*, but performance is slower. The TMP analog to a function is a template class that performs computations at compile time:

```
#include <iostream>
using namespace std;

template< int X, int Y >
struct MyStruct
{
    enum
    {
```



```
        value = X + Y
    };
};

int main()
{
      int a = MyStruct< 1, 2 >::value;
      cout << a << endl;

      return 0;
}
```

This time, MyStruct adds the two template parameter values at compile time and stores the sum in value, which is used as a literal 3 at run-time.

We can create a conditional statement by writing two **specializations** of a template class. The compiler selects the definition that fits the template parameter values or types provided.

```
#include <iostream>
using namespace std;

template< int X >
struct MyStruct
{
   enum
   {
      value = 1
   };
};

template<>
struct MyStruct< 0 >
{
   enum
   {
      value = 0
   };
};

int main()
{
      if( MyStruct< 5 >::value )
      {
            cout << "True" << endl;
      }

      if( MyStruct< 0 >::value )
      {
            cout << "False" << endl;
      }
      return 0;
}
```

Since variables in TMP are immutable, iteration at run-time impossible. However, we can use recursion at compile time. This can be done through a template class which instantiates itself (or specializations of itself). As a commonly used example, factorials can be calculated in this fashion using specialization to provide the terminating condition for the recursion

```
#include <iostream>
using namespace std;

template< int X >
class Factorial
{
public:
	enum
	{
		value = X * Factorial< X - 1 >::value
	};
};

template<>
struct Factorial< 0 >
{
public:
	enum
	{
		value = 1
	};
};

int main()
{
    int x = Factorial< 4 >::value;
    cout << x << endl;
}
```

As you can see, the specialization for a template parameter 0 evaluates to 1. This is the termination condition to the recursion.

1. Compile-time Code Optimization

TMP can be used to create a vector class (basically a dynamically allocated array) as long as the vector's length is known at compile time. The benefit over a traditional STL vector is that loops can be **unrolled** at compile time. This generates highly optimized code. Consider the following example using operator+=.

Vector.h

```
template< class T, int N >
class Vector
{
public:
     T values[ N ];
     int mark;

     Vector() : mark( 0 ) {}
     enum
     {
          size = N
     };
     T *begin()
     {
          return values;
     }
     T *end()
     {
          return values + size;
     }
     void push( T v )
     {
          values[ mark ] = v;
          mark++;
     }
     void operator+=( const Vector< T, N > &rhs )
     {
          for( int i = 0; i < N; ++i )
               values[ i ] += rhs.values[ i ];
     }
};
```

When the compiler instantiates the operator+= template function, something like the following code should be produced:


```
void operator+=( const Vector< int, 3 > &rhs )
{
    values[ 0 ] += rhs.values[ 0 ];
    values[ 1 ] += rhs.values[ 1 ];
    values[ 2 ] += rhs.values[ 2 ];
}
```

Main.cpp

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

int main()
{
      Vector< int, 5 > a;
      Vector< int, 5 > b;

      for( int i = 0; i < 5; i++ )
      {
            a.push( i );
            b.push( i * 10 );
      }

      a += b;

      int *iter = a.begin();
      while( iter != a.end() )
      {
            cout << *iter << endl;
            iter++;
      }
      return 0;

      return 0;
}
```

When programming with types, named values are typedefs:

```
struct ValueHolder
{
   typedef int value;
};
```

Here, we are storing the *int* type so that it can be accessed with the name `value`.

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

template< typename T >
inline typename AccumulationTraits< T >::AccT accum( T const *begin, T
const *end )
{
      typedef typename AccumulationTraits< T >::AccT AccT;
      AccT total = T();
      while ( begin != end )
      {
            total += *begin;
            ++begin;
      }
      return total;
}

int main()
{
      double a[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };
      cout << accum( &a[ 0 ], &a[ 5 ] ) / 5 << endl;

      return 0;
}
```

```
template< typename T >
class AccumulationTraits;

template<>
class AccumulationTraits< char >
{
public:
      typedef int AccT;
      static AccT zero()
      {
            return 0;
      }
};
```



```
template<>
class AccumulationTraits< int >
{
public:
      typedef int AccT;
      static AccT zero()
      {
            return 0;
      }
};

template<>
class AccumulationTraits< double >
{
public:
      typedef double AccT;
      static AccT zero()
      {
            return 0;
      }
};
```



```
#include <iostream>
using namespace std;

////////////////////////////////////////////////
template< bool C, typename Ta, typename Tb >
class IfThenElse;

template< typename Ta, typename Tb >
class IfThenElse< true, Ta, Tb >
{
public:
	typedef Ta ResultT;
};

template< typename Ta, typename Tb >
class IfThenElse< false, Ta, Tb >
{
public:
	typedef Tb ResultT;
};
////////////////////////////////////////////////

template< int N >
class Value
{
public:
	enum
	{
		result = N
	};
};

template< int N, int I = 1 >
class Sqrt
{
public:
	typedef typename IfThenElse< ( I * I < N ), Sqrt< N, I + 1 >,
Value< I > >::ResultT SubT;

	enum
	{
		result = SubT::result
	};
};

int main()
{
	// find the sqrt of 25;

	//int i;
	//for( i = 0; i * i < 25; ++i )
	//{
	//}
	//cout << i << endl;
```



```
        cout << Sqrt< 25 >::result << endl;

        return 0;
}
```


2. Matrix Multiplication

The old way.

```
#include <iostream>
using namespace std;

template< typename T >
inline T dot_product( int dim, T *a, T *b )
{
	T result = T();
	for ( int i = 0; i < dim; ++i )
	{
		result += a[ i ] * b[ i ];
	}
	return result;
}

int main()
{
	int a[] = { 2, 4, 6 };
	int b[] = { 3, 5, 7 };

	int sum = dot_product( 3, a, b );
	cout << sum << endl;
	return 0;
}
```

The new way.

```
#include <iostream>
using namespace std;

// primary template
template< int DIM, typename T >
class DotProduct
{
public:
   static T result ( T *a, T *b )
   {
      return *a * * b + DotProduct< DIM - 1, T >::result( a + 1, b + 1 );
      }
};

template< typename T >
class DotProduct< 1, T >
{
public:
	static T result( T *a, T *b )
	{
		return *a * * b;
	}
};
```



```
// Convenience function
template< int DIM, typename T >
inline T dot_product( T *a, T *b )
{
      return DotProduct< DIM, T >::result( a, b );
}

int main()
{
      int a[] = { 2, 4, 6 };
      int b[] = { 3, 5, 7 };

      int prod = dot_product< 3 >( a, b );
      cout << prod << endl;

      return 0;
}
```

3. The Curiously Recurring Template Pattern

Polymorphism describes the case where a base class pointer points to an instance of a derived, or child, class. In such a case, the child class method definitions run. This behavior is enabled by the virtual function look-up tables (and RTTI). Use of the look-up table entails run-time overhead. In many cases, polymorphic behavior can be resolved at compile-time using what is called the Curiously Recurring Template Pattern (CRTP). The CRTP achieves static polymorphism, which imitates polymorphism but without a look-up table. The CRTP passes a child class as a template argument to its base class.

Instrument.h

```
template< class Child >
class Instrument
{
public:
	void calc_value()
	{
		static_cast< Child * >( this )->theo();
	}
};
```

Child Classes

```
#include <iostream>
using namespace std;

class Option : public Instrument< Option >
{
public:
	void theo()
	{
		cout << "Hello from the Option class" << endl;
	}
};

class Bond : public Instrument< Bond >
{
public:
	void theo()
	{
		cout << "Hello from the Bond class" << endl;
	}
};
```

Main.cpp

```
int main()
```



```
{
	Instrument< Option > a;
	a.calc_value();

	Option b;
	b.calc_value();

	Instrument< Bond > *c = new Bond;
	c->calc_value();
	delete c;

	Bond *d = new Bond;
	d->calc_value();
	delete d;

	return 0;
}
```

4. Barton-Nackman Trick

The Barton–Nackman trick, also called restricted template expansion, shows that common functionality can be placed in a base class to enforce conformant behavior.

```
#include <iostream>
using namespace std;

template< typename T >
class Comparsion
{
    friend bool operator==( T &a, T &b )
    {
        return a.equal_to( b );
    }
    friend bool operator!=( T &a, T &b )
    {
        return !a.equal_to( b );
    }
};

class MyClass : private Comparsion< MyClass >
{
public:
    int value;
    bool equal_to( MyClass &rhs )
    {
            return ( value == rhs.value );
    }
};

int main()
{
      MyClass x;
      x.value = 4;

      MyClass y;
      y.value = 4;

      cout << ( x == y ) << endl;

      return 0;
}
```


5. Static_assert

Unlike the assert function, **static_assert** is a simple way to create *compile-time* error messages. To call **static_assert,** enter the condition you want to test for and then the error message you would like the compiler to output if the condition is false. Because templates are instantiated after pre-processing and before run-time, we would otherwise have no way to test the validity of template parameters. The example below generates an error message when we declare FiveYear as a Bond type with template parameter of type int.

```
#include<string>
using namespace std;

template< typename T >
struct Bond
{
   static_assert( !std::numeric_limits< T >::is_integer, "Need double!" );
   string bondName;
   T Price;
};

int main()
{
   Bond< double > TenYear;
   Bond< int > FiveYear;
   return 0;
}
```


CHAPTER 7: FINANCIAL APPLICATIONS

1. Present Value Function

The present value function accepts a vector containing the timing of future cash flows, as well as a vector containing the amounts of those cash flows and the rate at which to discount those cash flows.

```
double present_value_discrete( const vector< double > &times,
                               const vector< double > &cash_flows,
                               double r )
{
     double present_value = 0.0;
     for ( int i = 0; i < times.size(); i++ )
     {
          present_value += cash_flows[ i ] / pow( 1.0 + r, times[ i ] );
     }
     return present_value;
}
```

The following program prints the present value of 1000.

```
#include <iostream>
#include <vector>
#include <cmath>
using namespace std;

double present_value_discrete( const vector< double > &,
                               const vector< double > &,
                               double );
int main()
{
      vector< double > times;
      vector< double > cf;
      double rate = .05;

      times.push_back( 1 );
      times.push_back( 2 );
      times.push_back( 3 );

      cf.push_back( 50 );
      cf.push_back( 50 );
      cf.push_back( 1050 );

      cout << present_value_discrete( times, cf, rate ) << endl;
      return 0;
}
```


2. Yield to Maturity

The yield to maturity is the interest rate that makes the present value of the future coupon payments equal to the price of the bond. The algorithm for finding yield to maturity is a simple bisection technique. The bond yield is between zero, the lower bound, and the upper bound, which can be found by increasing the rate until the bond price is negative. Then, bisect the interval between the upper and lower until the rate is within tolerance.

```
double yield_to_maturity( const vector< double > &times,
                          const vector< double > &cash_flows,
                          double bond_price )
{
     const double TOLERANCE = .00001;

     double lower = 0, upper = 1.0;

     while ( present_value_discrete( times, cash_flows, upper ) >
             bond_price )
     {
          upper = upper * 2;
     }

     double r = 0.5 * ( upper + lower );

     for ( int i = 0; i < 200; i++ )
     {
        double diff = present_value_discrete( times, cash_flows, r ) –
                      bond_price;
          if ( fabs( diff ) < TOLERANCE )
          {
              return r;
          }
          if ( diff > 0.0 )
          {
              lower = r;
          }
          else
          {
               upper = r;
          }
          r = 0.5 * ( upper + lower );
     }
     return r;
}
```

The following program should also include the present_value_discrete() function and its header, and prints the yield to maturity as .05.

```
#include <iostream>
#include <vector>
#include <cmath>
using namespace std;
```



```
double yield_to_maturity( const vector< double > &,
                          const vector< double > &,
                          double );

int main()
{
      vector< double > times;
      vector< double > cf;
      double rate = .05;

      times.push_back( 1 );
      times.push_back( 2 );
      times.push_back( 3 );

      cf.push_back( 50 );
      cf.push_back( 50 );
      cf.push_back( 1050 );

      cout << yield_to_maturity( times, cf, 1000.0 ) << endl;
      return 0;
}
```


3. Linear Interpolation

Linear interpolation is a method of curve fitting using linear polynomials. If the coordinates of two points are known, the linear interpolant is the straight line between these points. For a point in the interval , the coordinate values, x and y, along the straight line are given by:

$$\frac{y - y_0}{y_1 - y_0} = \frac{x - x_0}{x_1 - x_0}$$

Given a set of yields for a set of maturities, the simplest way to construct a term structure is by linear interpolation. This will help us find yields between known observations. This example shows linear interpolation of spot rates.

```
double linearly_interpolated_yield( const double time,
                                    const vector< double > &times,
                                    const vector< double > &yields)
{
     // assume the yields are in increasing time to maturity order.
     int no_obs = times.size();
     if ( no_obs < 1 )
     {
          return 0;
     }
     double t_min = times[0];

     if ( time <= t_min )
     {
          return yields[0]; // earlier than lowest observation.
     }
     double t_max = times[ no_obs – 1 ];

     if ( time >= t_max )
     {
          return yields[ no_obs – 1 ]; // later than latest obs
     }
     int t = 1; // find which two observations we are between

     while ( ( t < no_obs ) && ( time > times[ t ] ) )
     {
          ++t;
     }
     double lambda = ( times[ t ] - time) /
                     ( times[ t ] – times[ t – 1 ] );
     // by ordering assumption, time is between t-1,t

     double r = yields[ t – 1 ] * lambda +
                yields[ t ] * ( 1.0 – lambda );
     return r;
}
```

The following program prints out .32 as the interpolated yield at time .7.


```
#include <iostream>
#include <vector>
#include <cmath>
using namespace std;

double linearly_interpolated_yield( const double,
                                    const vector< double > &,
                                    const vector< double > & );
int main()
{
      vector<double> times;
      vector<double> yields;

      times.push_back(0.5);
      times.push_back(1);

      yields.push_back(0.2);
      yields.push_back(0.5);

      cout << linearly_interpolated_yield( 0.7, times, yields )
           << endl;

      return 0;
}
```


4. Cubic Spline

Cubic splines are well known for their good interpolation behaviour. The cubic spline parameterization can be used to estimate the term structure of interest rates as well as to approximate the discount factor. If the spline knots are known, this is a simple linear regression.

```
double term_structure_cubic_spline( const double t,
                                    const double b1,
                                    const double c1,
                                    const double d1,
                                    const vector< double > &f,
                                    const vector< double > &knots )
{
  double d = 1.0 + b1 * t + c1 * ( pow( t, 2 ) ) + d1 * ( pow( t, 3 ));
  for ( int i = 0; i < knots.size(); i++ )
  {
       if ( t >= knots[i] )
       {
            d += f[ i ] * ( pow(( t - knots[ i ] ), 3));
       }
       else
       {
            break;
       }
  }
  return d;
}
```

This test program prints out the discount factor as 1.1.

```
#include <iostream>
#include <vector>
#include <cmath>
using namespace std;

double term_structure_cubic_spline( const double,
                                    const double,
                                    const double,
                                    const double,
                                    const vector< double > &,
                                    const vector< double > & );
int main()
{
      double b = 0.1;
      double c = 0.1;
      double d = -0.1;

      vector< double > f;
      f.push_back( 0.01 );
      f.push_back( 0.01 );
      f.push_back( -0.01 );
```



```
      vector<double> knots;
      knots.push_back( 2 );
      knots.push_back( 7 );
      knots.push_back( 12 );

      cout << term_structure_cubic_spline( 1, b, c, d, f, knots )
           << endl;

      return 0;
}
```


4. Binomial Option Pricing

Pricing an option using a binomial tree is done by working backwards from the expiration date, where we know all the outcomes for the value of the underlying. For each outcome, we calculate the payoff of the option, and find what the possible option prices are one period prior. We do this again and again, working our way back to the root of the tree, time zero.

```
double binomial_european_call( const double S, // underlying price
                               const double X, // excerise price
                               const double r, // interest rate
                               const double u, // up movement
                               const double d, // down movement
                               const int periods)
{
     double Rinv = exp( -r ); // inverse of interest rate
     double uu = u * u;
     double p_up = ( exp( r ) – d ) / ( u – d );
     double p_down = 1.0 – p_up;

     vector< double > prices( periods + 1 );
     prices[ 0 ] = S * pow( d, periods );

     for ( int i = 1; i <= periods; ++i )
     {
          prices[ i ] = uu * prices[ i – 1 ];
     }
     vector< double > call_values( periods + 1 );

     for ( int i = 0; i <= periods; ++i )
     {
          call_values[ i ] = max( 0.0, ( prices[ i ] – X ));
     }

     for ( int step = periods - 1; step >= 0; --step)
     {
          for ( int i = 0; i <= step; ++i )
          {
              call_values[ i ] = ( p_up * call_values[ i + 1 ] +
                                   p_down * call_values[ i ] ) * Rinv;
          }
     }
     return call_values[ 0 ];
}
```

The following test programs prints the call price as 5.44255.

```
#include <iostream>
#include <vector>
#include <cmath>
using namespace std;
```



```
double binomial_european_call( const double,
                               const double,
                               const double,
                               const double,
                               const double,
                               const int );

int main()
{
      double S = 100.0;
      double X = 100.0;
      double r = 0.025;
      double u = 1.05;
      double d = 1 / u;
      int periods = 2;
      cout << binomial_european_call( S, X, r, u, d, periods ) << endl;
      return 0;
}
```

5. Black Scholes Option Pricing

The Black Scholes equation makes use of the cumulative normal distribution function, N(), shown later in this study guide.

```
double black_scholes_european_call( const double S, // underlying price
                                    const double X, // strike price
                                    const double r, // interest rate
                                    const double t, // time to expiry
                                    const double sigma ) // volatility
{
     double time = sqrt( t );
     double d1 = ( log( S / X ) + r * t ) /
                 ( sigma * time ) + 0.5 * sigma * time;
     double d2 = d1 - ( sigma * time );
     double c = S * N( d1 ) – X * exp( -r * t ) * N( d2 );
     return c;
}
```

This program prints out the call price as 5.4325.

```
#include <iostream>
#include <vector>
#include <cmath>
using namespace std;

double black_scholes_european_call( const double,  // underlying price
                                    const double,  // strike price
                                    const double,  // interest rate
                                    const double,  // time to expiry
                                    const double ); // volatility
double N(const double z);

int main()
{
     double S = 50;
     double X = 50;
     double r = 0.10;
     double sigma = 0.30;
     double time = 0.50;
     cout << black_scholes_european_call( S, X , r, time, sigma )
          << endl;
     return 0;
}
```


6. Newton-Raphson Method for Finding Implied Volatility

```
double IV_black_scholes_call( const double S,     // underlying price
                              const double X,     // strike price
                              const double r,     // interest rate
                              const double t,     // time to expiration
                              const double price )
{
     double sigma_low = .0001;
     double px = black_scholes_call( S, X, r, t, sigma_low );

     if ( px > price)
     {
          return 0.0;
     }

     const double TOLERANCE = .0001;
     double time = sqrt( t );
     double sigma = ( price / S ) / ( 0.398 * time );

     for ( int i = 0; i < 100; i++ )
     {
         px = black_scholes_call( S, X, r, sigma, t );
         double diff = price - px;
         if ( fabs( diff ) < TOLERANCE )
         {
             return sigma;
         }
         double d1 = ( log( S / X ) + r * t ) /
                     ( sigma * time ) + 0.5 * sigma * time;
         double vega = S * time * N( d1 );
         sigma = sigma + diff / vega;
     }
     return 0;
}
```


6. Explicit Finite Difference Method

```
double finite_diff_european_put( const double &S,
                                 const double &X,
                                 const double &r,
                                 const double &sigma,
                                 const double &t,
                                 const int &s_steps,
                                 const int &t_steps )
{
     double sigma_squared = sigma * sigma;
     unsigned int m;

     if ( ( s_steps % 2) == 1 )
     {
          m = s_steps + 1;
     }
     else
     {
          m = s_steps;
     }
     double delta_s = 2.0 * S / m;

     vector< double > s_values( m + 1 );
     for ( unsigned i = 0; i <= m; i++ )
     {
          S_values[ i ] = i * delta_s;
     }

     int n = t_steps;
     double delta_t = t / n;

     vector< double > a( m );
     vector< double > b( m );
     vector< double > c( m );

     double r1 = 1.0 / ( 1.0 + r * delta_t );
     double r2 = delta_t / ( 1.0 + r * delta_t );

     for ( unsigned int j = 1; j < m; j++ )
     {
          a[ j ] = r2 * 0.5 * j * ( -r + sigma_squared * j );
          b[ j ] = r1 * ( 1.0 – sigma_squared * j * j * delta_t );
          c[ j ] = r2 * 0.5 * j * ( r + sigma_squared * j );
     }
     vector< double > f_next( m + 1 );

     for ( unsigned i = 0; i <= m; ++i )
     {
          f_next[ i ] = max( 0.0, X – s_values[ i ] );
     }

     double f[ m + 1 ];

     for ( int t = n - 1; t >= 0; --t)
```



```
     {
          f[ 0 ] = X;
          for ( unsigned i = 1; i < m; ++i )
          {
               f[ i ] = a[ i ] * f_next[ i – 1 ] + b[ i ] *
                         f_next[ i ] + c[ i ]* f_next[ i + 1 ];
          }
          f[ m ] = 0;
          for (unsigned i = 0; i <= m; ++i )
          {
               f_next[ i ] = f[ i ];
          }
     }
     return f[ m / 2 ];
}
```


7. Using Simulation to Price Options

```
double simulated_european_call( const double S, // price of underlying
                                const double X, // exercise price
                                const double r, // risk free rate
                                const double sigma, // volatility
                                const double time, // time to expiry
                                const int sims)
{
     double R = ( r - 0.5 * pow( sigma, 2 )) * time;
     double SD = sigma * sqrt( time );
     double sum_payoffs = 0.0;
     for ( int n = 1; n <= sims; n++ )
     {
          double ST = S * exp( R + SD * random_normal() );
          sum_payoffs += max( 0.0, ST – X );
     }
     return exp( -r * time ) * ( sum_payoffs / double( sims ));
}
```


8. Standard Normal Distribution

This is the standard normal probability density function.

```
#define PI 3.14159265359

double n( const double z )
{
     return ( 1.0 / sqrt( 2.0 * PI )) * exp( -0.5 * z * z );
}
```

This is the standard normal cumulative density function.

```
double N(const double z)
{
     if ( z > 6.0 )
     {
          return 1.0;
     }
     if (z < -6.0)
     {
          return 0.0;
     }

     double b1 = 0.31938153;
     double b2 = -0.356563782;
     double b3 = 1.781477937;
     double b4 = -1.821255978;
     double b5 = 1.330274429;
     double p = 0.2316419;
     double c2 = 0.3989423;
     double a = fabs( z );
     double t = 1.0 / ( 1.0 + a * p);
     double b = c2 * exp(( -z ) * ( z / 2.0 ));
     double n = (((( b5 * t + b4 ) * t + b3 ) * t + b2 ) * t + b1 ) * t;
     n = 1.0 – b * n;
     if ( z < 0.0 )
     {
          n = 1.0 - n;
     }
     return n;
}
```


9. Random Numbers

This Mersenne Twister algorithm class code is available at **www.benvanvliet.com**.

Main.cpp

```
#include <iostream>
#include "Mersenne.h"  // Available at www.benvanvliet.com
using namespace std;

int main()
{
	Mersenne::seed_generator( 1234 );

	for ( int i = 0; i < 100; i++ )
	{
		cout << Mersenne::rand_uniform() << endl;
	}

	return 0;
}
```

The code for generating random numbers from other continuous and discrete distributions is available at **www.benvanvliet.com**. Also, later we will look at using the **Boost library** random number generation capability.

CHAPTER 8: MATRICES

1. Matrix Math

This matrix template class is available at **www.benvanvliet.com**. The matrix class is adapted from Somnath Kundu's matrix11. As Kundu states, "it can be used to perform common matrix operations in your C++ program like any other built-in data types. The implementation is not optimized for performance and memory usage. So it is not suitable for large and sparse matrices. But for small and medium size matrices these should not be a problem, keeping in mind the present generation of powerful computers."

At the heart of the matrix class is a dynamically allocated, two-dimensional array within a structure called base_mat. The matrix class methods perform operations on this array, iterating through the elements to accomplish the correct mathematical algorithms. The following example gives a brief introduction to the base_mat structure.

```
#include <iostream>
using namespace std;

template< class T >
class matrix
{
      struct base_mat
      {
            T **a;
            size_t row_size;
            size_t col_size;

            base_mat ( size_t r, size_t c )
            {
                row_size = r;
                col_size = c;

                a = new T *[ r ];

                for( size_t i = 0; i < r; i++ )
                {
                      a[ i ] = new T [ c ];
                }
            }
            ~base_mat()
            {
                for( size_t i = 0; i < row_size; i++ )
                {
                      delete [] a[ i ];
                }
                delete [] a;
            }
      };

private:
      base_mat *m;

public:
```



```
      matrix( size_t r, size_t c )
      {
           m = new base_mat( r, c );
      }
      ~matrix()
      {
           delete m;
      }
      T &operator()( size_t r, size_t c )
      {
           return m->a[ r ][ c ];
      }
};

typedef matrix< double > Matrix;

int main()
{
      Matrix m( 5, 5 );

      for ( int i = 0; i < 5; i++ )
      {
            for ( int j = 0; j < 5; j++ )
            {
                  m( i, j ) = i + j;
                  cout << m( i, j ) << '\t';
            }
            cout << endl;
      }
}
```

For our purposes, this matrix class works very well. The following example illustrates the many overloaded operators and methods on the matrix class, including matrix addition, subtraction, multiplication, inversion, and solving.

```
#include <iostream>
#include "matrix.h" // Code available at www.benvanvliet.com
using namespace std;

int main()
{
   try
   {
      Matrix m1( 3, 3 );
      m1( 0, 0 ) = 10;              //    10    2     2
      m1( 0, 1 ) = 2;               //     6    4     3
      m1( 0, 2 ) = 2;               //     2    4     3
      m1( 1, 0 ) = 6;
      m1( 1, 1 ) = 4;
      m1( 1, 2 ) = 3;
      m1( 2, 0 ) = 2;
      m1( 2, 1 ) = 4;
      m1( 2, 2 ) = 3;
```



```
      Matrix m2( 3, 3 );
      m2( 0, 0 ) = 6;                //     6    5     3
      m2( 0, 1 ) = 5;                //     4    4     7
      m2( 0, 2 ) = 3;                //    10    5     6
      m2( 1, 0 ) = 4;
      m2( 1, 1 ) = 4;
      m2( 1, 2 ) = 7;
      m2( 2, 0 ) = 10;
      m2( 2, 1 ) = 5;
      m2( 2, 2 ) = 6;

      Matrix m3 = m1 + m2;
      cout << "M1 + M2 = \n" << m3 << endl;

      m3 = m1 - m2;
      cout << "M1 - M2 = \n" << m3 << endl;

      m3 = m1 * m2;
      cout << "M1 * M2 = \n" << m3 << endl;

      m3 = !m1;
      cout << "Inverse of M1 = \n" << m3 << endl;

      m3 = m1 ^ 4U;
      cout << "M ^ 4 = \n" << m3 << endl;

      m3 = m1.Adj();
      cout << "Adjoint of M1 = \n" <<  m3 << endl;

      cout << "Cofactor of M1( 0, 0 ) = " << m1.Cofact( 0, 0 ) << endl;
      cout << "Determinant of M1 = " << m1.Det() << endl;
      cout << "Condition No. of M1 = " << m1.Cond() << endl;
      cout << "Norm of M1 = " << m1.Norm() << endl;

      m3 = ( m1 * 4.0 ) + ( m2 ^ 2U ) - ( 5.0 * m1 / 4.0 ) * ( 2.0 / m2 );

      cout << "Solution:\n" << m3 << endl;

      Matrix v( 3, 1 ), s( 3, 1 );

      cout << "Enter the vector as (3x1) matrix to solve m1:\n";
      cin >> v;  // 100, 120, 100

      s = m1.Solve( v );  // Logically equivalent to:  s = m1.Inv() * v;
      cout << "Solution:\n" << s << endl;
   }
   catch ( exception e )
   {
      cout << e.what() << endl;
   }
}
```


2. Reading Data into a Matrix

The following program uses tab separated values in a data.txt file. The program uses prices to calculate a matrix of log returns.

```
10    50    25
11    51    26
9     49    27
12    48    25
13    47    26
11    44    25
12    47    24
11    46    25
10    48    26
```

```
#include <cmath>
#include <fstream>
#include "matrix.h"  // Code available at www.benvanvliet.com
using namespace std;

int main()
{
   ifstream m_InFile( "C:\\data.txt" );

   Matrix prices( 9, 3 );

   m_InFile >> prices;

   Matrix returns( 8, 3 );
   for ( int r = 0; r < 8; r++ )
   {
      for ( int c = 0; c < 3; c++ )
      {
         returns( r, c ) = log( prices( r + 1, c ) / prices( r, c ) );
      }
   }
   cout << returns;
   return 0;
}
```


3. Optmization Using the Simplex Method

Bohan Liu's class code for the simplex algorithm is available at **www.benvanvliet.com**.

```
//  This code was written by Bohan Liu.

#include <iostream>
#include <iomanip>

#include "Simplex.h"  // Code available at www.benvanvliet.com

using namespace std;

int main()
{
    // ----------------------------- Example #1 -----------------------------

    Simplex x1(3,3);

    double targetfunct1[] = {3, 2, 4};

    x1.AddTargetFunct(targetfunct1, false);

    double constr11[] = {1, 1, 2, 4};
    double constr12[] = {2, 0, 3, 5};
    double constr13[] = {2, 1, 3, 7};

    x1.AddConstraints(constr11);
    x1.AddConstraints(constr12);
    x1.AddConstraints(constr13);

    cout << setw(30) << "EXAMPLE ONE\n" << endl;

    Simplex::Outputs op1 = x1.result();

    cout << "Optimizers: " << op1.Optimizers << "\nTarget Value: " <<
                                op1.Value << endl;

    cout << endl;

    ----------------------------- Example #2 -----------------------------

    Simplex x2(3,3);

    double targetfunct2[] = {15, 17, 20};

    x2.AddTargetFunct(targetfunct2, false);

    double constr21[] = {0, 1, -1, 2};
    double constr22[] = {3, 3, 5, 15};
    double constr23[] = {3, 2, 1, 8};

    x2.AddConstraints(constr21);
    x2.AddConstraints(constr22);
    x2.AddConstraints(constr23);
```



```
    cout << setw(30) << "EXAMPLE TWO\n" << endl;

    Simplex::Outputs op2 = x2.result();

    cout << "Optimizers: " << op2.Optimizers << "\nTarget Value: "
                           << op2.Value << endl;

    cout << endl;

    // ----------------------------- Example #3 -----------------------------

    Simplex x3(3,3);

    double targetfunct3[] = {1, 2, -1};

    x3.AddTargetFunct(targetfunct3, false);

    double constr31[] = {-2, -1, -1, -14};
    double constr32[] = {4, 2, 3, 28};
    double constr33[] = {2, 5, 5, 30};

    x3.AddConstraints(constr31, Simplex::greater);
    x3.AddConstraints(constr32, Simplex::less);
    x3.AddConstraints(constr33, Simplex::less);

    cout << setw(30) << "EXAMPLE THREE\n" << endl;

    Simplex::Outputs op3 = x3.result();

    cout << "Optimizers: " << op3.Optimizers << "\nTarget Value: "
                           << op3.Value << endl;

    cout << endl;

    return 0;
}
```

CHAPTER 9: LIBRARIES

1. Creating Static and Dynamic Libraries

You can create static libraries (.lib) that can be used by other applications. When you compile a program that references a static library, the .lib file is linked into the executable. The first thing to do is to create a static library; the second is to create an executable that references the library. To create a static library:

STEP 1 Create a new Win32 Console Application named MyQuant. In the Application Wizard, click on Application Settings and set the Application Type as Static Library and uncheck Precompiled Header. Click finish.

STEP 2 Add the following class to the library.

MyClass.h

```
namespace MyQuant
{
	class MyClass
	{
	public:
		static double Add( double, double );
	};
}
```

MyClass.cpp

```
#include "MyClass.h"

namespace MyQuant
{
	double MyClass::Add( double a, double b )
	{
		return a + b;
	}
}
```

Be sure the configuration type of the property of the library is static library (.lib) by right clicking the MyQuant in the Solution Explorer, and find the configuration type from the properties.

STEP 3 Compile the library.

Now, to create the executable that will reference the static library:

STEP 1 Create a new Win32 Console Application named Lib_Test.

STEP 2 Add the following code.

```
#include "stdafx.h"
#include <iostream>

#include "MyClass.h"
```



```
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
	cout << MyQuant::MyClass::Add(3.0, 5.0)<< endl;

	return 0;
}
```

STEP 3 Open the Properties Pages of the Lib_Test project. In the Solution Explorer, right click on the project name and select Properties.

STEP 4 From the Configuration Properties, select C/C++. Under C/C++, select General. In the right hand window, in Additional Include Directories, type the full path to MyClass.h. For example, C:\MyQuant\MyQuant.

STEP 5 From the Configuration Properties, select Linker. Under Linker, select Input. In the right hand window, in Additional Dependencies, type the full path and file name of the .lib file. For example, C:\MyQuant\debug\MyQuant.lib

STEP 6 Compile and run Lib_Test.exe.

To create and use a dynamic link library (.dll), the instructions are exactly the same. As said, in static linking the linker appends the library code to the executable. In dynamic linking, however, no code is appended to the executable from the library. Rather, the executable keeps the name of the required .dll, and at runtime, it loads the need function code from it. This minimizes the size of the executable, but impacts speed. Dynamic libraries make upgrading easier. If you update the .dll file, you will not need to recompile the .exe project to incorporate the changes. In the case of a static library, changes to the .lib file will only be reflected when the .exe project is recompiled.

2. Installing Boost

Boost is a collection of free, open source libraries that extend the standard C++ language by providing a wide variety of classes and functionalities. To install the Boost libraries, follow these steps:

.

STEP 1	Download the latest Boost code file (.zip) at: http://sourceforge.net/projects/boost/files/boost/1.45.0/
STEP 2	Unzip this file to the directory: C:\Program Files\boost\
STEP 3	From the command window, change the directory path to: C:\Program Files\boost\boost_1_45_0
STEP 4	At the command prompt, type: bootstrap
STEP 5	After Step 4 has completed, at the command prompt type: .\bjam
STEP 6	To use Boost libraries in your Visual Studio project, be sure to add **C:\Program Files\boost\boost_1_45_0** in the compiler C/C++ | General | Additional Include Directories. And, add **C:\Program Files\boost\boost_1_45_0\stage\lib** to Additional Dependencies under Linker | Input. Also, under C/C++ | Precompiled Headers, toggle Precompiled Header to Not Using Precompiled Headers.

The Boost website contains a wealth of documentation at: http://www.boost.org/doc/libs/

4. Installing QuantLib

Download Quantlib-0.9.7.zip at http://quantlib.org/download.shtml and click on Download Page. This will connect you to the SourceForge repository. Extract the zip file to your C:\ drive. In the extracted folder, open Quantlib_vc9.sln in Visual Studio 2008. Be sure to select Release static runtime as solution configuration before building QuantLib. Build QuantLib. This will take several minutes.

In order to use QuantLib in your C++ projects, you will need to add "C:\QuantLib-0.9.7" to "Included files" and "C:\QuantLib-0.9.7\lib" to "Library files". To use Quantlib, create a new C++ program and add **#include <ql/quantlib.hpp>** and **using namespace QuantLib.** Be aware also that the **Release (static runtime)** configuration is required when you use QuantLib.

CHAPTER 10: BOOST

The following list of library names and descriptions has been excerpted from the Boost website:

Boost Library Name	Description
Accumulators	Framework for incremental calculation, and collection of statistical accumulators.
Any	Safe, generic container for single values of different value types.
Array	STL compliant container wrapper for arrays of constant size.
Asio	Portable networking, including sockets, timers, hostname resolution and socket iostreams.
Assign	Filling containers with constant or generated data has never been easier.
Bimap	Bidirectional maps library for C++. With Boost::Bitmap you can create associative containers in which both types can be used as key.
Bind	Boost::bind is a generalization of the standard functions std::bind1st and std::bind2nd. It supports arbitrary function objects, functions, function pointers, and member function pointers, and is able to bind any argument to a specific value or route input arguments into arbitrary positions.
Call Traits	Defines types for passing parameters.
Circular Buffer	A STL compliant container also known as ring or cyclic buffer.
Compatibility	Help for non-conforming standard libraries.
Compressed Pair	Empty member optimization.
Concept Check	Tools for generic programming.
Config	Helps Boost library developers adapt to compiler idiosyncrasies; not intended for library users.
Conversion	Polymorphic and lexical casts.
CRC	The Boost CRC Library provides two implementations of CRC (cyclic redundancy code) computation objects and two implementations of CRC computation functions. The implementations are template-based.
Date Time	A set of date-time libraries based on generic programming concepts.
Disjoint Sets	Boost.DisjointSets provides disjoint sets operations with union by rank and path compression.
Dynamic Bitset	The dynamic_bitset class represents a set of bits. It provides accesses to the value of individual bits via an operator[] and provides all of the bitwise operators that one can apply to builtin integers, such as operator& and operator<<. The number of bits in the set is specified at runtime via a parameter to the constructor of the dynamic_bitset.
Enable If	Selective inclusion of function template overloads.
Exception	The Boost Exception library supports transporting of arbitrary data in exception objects, and transporting of exceptions between threads.
Filesystem	The Boost Filesystem Library provides portable facilities to query and manipulate paths, files, and directories.
Flyweight	Design pattern to manage large quantities of highly redundant objects.
Foreach	Automates iteration over a sequence.
Format	The format library provides a class for formatting arguments according to a format-string, as does printf, but with two major differences: format sends the arguments to an internal stream, and so is entirely type-safe and naturally supports all user-defined types; the ellipsis (...) can not be used correctly in the strongly typed context of format, and thus the function call with arbitrary arguments is replaced by successive calls to an argument feeding operator%.
Function	Function object wrappers for deferred calls or callbacks.
Function Types	Boost.FunctionTypes provides functionality to classify, decompose and synthesize function, function pointer, function reference and pointer to member

	types.
Functional	The Boost.Function library contains a family of class templates that are function object wrappers.
Functional/Factory	Function object templates for dynamic and static object creation
Functional/Forward	Adapters to allow generic function objects to accept arbitrary arguments
Functional/Hash	A TR1 hash function object that can be extended to hash user defined types.
Fusion	Library for working with tuples, including various containers, algorithms, etc.
GIL	Generic Image Library
Graph	The BGL graph interface and graph components are generic, in the same sense as the Standard Template Library (STL).
In Place Factory, Typed In Place Factory	Generic in-place construction of contained objects with a variadic argument-list.
Integer	The organization of boost integer headers and classes is designed to take advantage of <stdint.h> types from the 1999 C standard without resorting to undefined behavior in terms of the 1998 C++ standard. The header <boost/cstdint.hpp> makes the standard integer types safely available in namespace boost without placing any names in namespace std.
Interprocess	Shared memory, memory mapped files, process-shared mutexes, condition variables, containers and allocators.
Interval	Extends the usual arithmetic functions to mathematical intervals.
Intrusive	Intrusive containers and algorithms.
IO State Savers	The I/O sub-library of Boost helps segregate the large number of Boost headers. This sub-library should contain various items to use with/for the standard I/O library.
Iostreams	Boost.IOStreams provides a framework for defining streams, stream buffers and i/o filters.
Iterators	The Boost Iterator Library contains two parts. The first is a system of concepts which extend the C++ standard iterator requirements. The second is a framework of components for building iterators based on these extended concepts and includes several useful iterator adaptors.
Lambda	Define small unnamed function objects at the actual call site, and more.
Lexical Cast	General literal text conversions, such as an int represented a string, or vice-versa, from Kevlin Henney.
Math	Boost.Math includes several contributions in the domain of mathematics.
Math Common Factor	Greatest common divisor and least common multiple.
Math Octonion	Octonions.
Math Quaternion	Quaternions.
Math/Special Functions	A wide selection of mathematical special functions.
Math/Statistical Distributions	A wide selection of univariate statistical distributions and functions that operate on them.
Member Function	Generalized binders for function/object/pointers and member functions.
Meta State Machine	A very high-performance library for expressive UML2 finite state machines.
Min-Max	Standard library extensions for simultaneous min/max and min/max element computations.
MPI	Message Passing Interface library, for use in distributed-memory parallel application programming.
MPL	The Boost.MPL library is a general-purpose, high-level C++ template metaprogramming framework of compile-time algorithms, sequences and metafunctions. It provides a conceptual foundation and an extensive set of powerful and coherent tools that make doing explict metaprogramming in C++ as easy and enjoyable as possible within the current language.
Multi-Array	Boost.MultiArray provides a generic N-dimensional array concept definition and common implementations of that interface.
Multi-Index	The Boost Multi-index Containers Library provides a class template named

	multi_index_container which enables the construction of containers maintaining one or more indices with different sorting and access semantics.
Numeric Conversion	Optimized Policy-based Numeric Conversions.
Operators	Templates ease arithmetic classes and iterators.
Optional	Discriminated-union wrapper for optional values.
Parameter	Boost.Parameter Library - Write functions that accept arguments by name.
Pointer Container	Containers for storing heap-allocated polymorphic objects to ease OO-programming.
Polygon	Booleans/clipping, resizing/offsetting and more for planar polygons with integral coordinates.
Pool	Memory pool management.
Preprocessor	Preprocessor metaprogramming tools including repetition and recursion.
Program Options	The program_options library allows program developers to obtain program options, that is (name, value) pairs from the user, via conventional methods such as command line and config file.
Property Map	Concepts defining interfaces which map key objects to value objects.
Property Tree	A tree data structure especially suited to storing configuration data.
Proto	Expression template library and compiler construction toolkit for domain-specific embedded languages.
Python	The Boost Python Library is a framework for interfacing Python and C++. It allows you to quickly and seamlessly expose C++ classes functions and objects to Python, and vice-versa, using no special tools -- just your C++ compiler.
Random	A complete system for random number generation.
Range	A new infrastructure for generic algorithms that builds on top of the new iterator concepts.
Rational	A rational number class.
Ref	A utility library for passing references to generic functions.
Regex	Regular expression library.
Result Of	Determines the type of a function call expression.
Scope Exit	Execute arbitrary code at scope exit.
Serialization	Serialization for persistence and marshalling.
Signals	Managed signals & slots callback implementation.
Signals2	Managed signals & slots callback implementation (thread-safe version 2).
Smart Ptr	Smart pointer class templates.
Spirit	LL parser framework represents parsers directly as EBNF grammars in inlined C++.
Statechart	Boost.Statechart - Arbitrarily complex finite state machines can be implemented in easily readable and maintainable C++ code.
Static Assert	Static assertions (compile time assertions).
String Algo	String algorithms library.
Swap	Enhanced generic swap function.
System	Operating system support, including the diagnostics support that will be part of the C++0x standard library.
Test	Support for simple program testing, full unit testing, and for program execution monitoring.
Thread	Portable C++ multi-threading.
Timer	Event timer, progress timer, and progress display classes.
Tokenizer	Break of a string or other character sequence into a series of tokens.
TR1	The TR1 library provides an implementation of the C++ Technical Report on Standard Library Extensions. This library does not itself implement the TR1 components, rather it's a thin wrapper that will include your standard library's TR1 implementation (if it has one), otherwise it will include the Boost Library equivalents, and import them into namespace std::tr1.
Tribool	3-state boolean type library.

Tuple	Ease definition of functions returning multiple values, and more.
Type Traits	Templates for fundamental properties of types.
Typeof	Typeof operator emulation.
uBLAS	uBLAS provides matrix and vector classes as well as basic linear algebra routines. Several dense, packed and sparse storage schemes are supported.
Units	Zero-overhead dimensional analysis and unit/quantity manipulation and conversion.
Unordered	Unordered associative containers.
Utility	Class noncopyable plus checked_delete(), checked_array_delete(), next(), prior() function templates, plus base-from-member idiom.
Uuid	A universally unique identifier.
Value Initialized	Wrapper for uniform-syntax value initialization, based on the original idea of David Abrahams.
Variant	Safe, generic, stack-based discriminated union container.
Wave	The Boost.Wave library is a Standards conformant, and highly configurable implementation of the mandated C99/C++ preprocessor functionality packed behind an easy to use iterator interface.
Xpressive	Regular expressions that can be written as strings or as expression templates, and which can refer to each other and themselves recursively with the power of context-free grammars.

1. Boost Sockets: UDP

A socket represents a network connection. A socket is parameterized by three things:

- The local IP address and port number.
- The remote socket address (Only for established TCP sockets.)
- The transport protocol (e.g., TCP, UDP, or others.)

This example uses UDP and the synchronous programming model.

Multicast Server

PriceSimulator.h

```
#include <cmath>
#include <boost/thread.hpp>
#include <boost/date_time/posix_time/posix_time_types.hpp>

using namespace boost;

class PriceSimulator
{
private:
	double m_Price;
	int m_Pause;
public:
	PriceSimulator( double starting_price ) : m_Price( starting_price )
		{}
	double get_price()
	{
		m_Pause = -1000.0 * log((double)(rand() % 5000 + 1)/5000.0 );
		this_thread::sleep( posix_time::milliseconds( m_Pause ) );

		if ( rand() % 2 + 1 > 1 )
		{
			m_Price += 0.25;
		}
		else
		{
			m_Price -= 0.25;
		}
		return m_Price;
	}
};
```

MyServer.h

```
#include <string>
#include <boost/asio.hpp>
#include <boost/lexical_cast.hpp>
#include "PriceSimulator.h"

using namespace std;
using namespace boost;
```



```
const char *multicast_address = "239.255.0.1";
const short m_Port = 30002;
const double starting_price = 1320.00;

class MyServer
{
private:
      PriceSimulator m_Simulator;
      asio::ip::udp::endpoint m_Endpoint;
      asio::ip::udp::socket m_Socket;

      string message;

public:
      MyServer( asio::io_service &m_Service ) :
          m_Simulator( starting_price ),
          m_Endpoint( asio::ip::address::from_string( multicast_address ),
                    m_Port ),
          m_Socket( m_Service, m_Endpoint.protocol() )
      {
      }
      void start()
      {
         while ( true )
         {
            message = lexical_cast< string >( m_Simulator.get_price() );
            cout << "Server: " << message << endl;
            m_Socket.send_to( asio::buffer( message ), m_Endpoint );
         }
      }
};
```

Main.cpp

```
#include <iostream>
#include <boost/asio.hpp>
#include "MyServer.h"

using namespace boost;
using namespace std;

int main()
{
      cout << "Starting server..." << endl;

      asio::io_service io_service;

      MyServer s( io_service );
      s.start();

      return 0;
}
```


Multicast Client

MyClient.h

```
#include <iostream>
#include <boost/asio.hpp>

using namespace boost;
using namespace std;

const char *multicast_address = "239.255.0.1";
const char *listen_address = "0.0.0.0";
const short m_Port = 30002;

class MyClient
{
private:
   asio::ip::udp::socket m_Socket;
   asio::ip::udp::endpoint m_Endpoint;
public:
   MyClient( asio::io_service &io_service ) :
             m_Socket( io_service ),
             m_Endpoint( asio::ip::address::from_string( listen_address ),
                         m_Port )
   {}

   void start()
   {
      m_Socket.open( m_Endpoint.protocol() );
      m_Socket.set_option( asio::ip::udp::socket::reuse_address( true ) );
      m_Socket.bind( m_Endpoint );

      // Join the multicast group.
      m_Socket.set_option( asio::ip::multicast::join_group(
                           asio::ip::address::from_string(
                           multicast_address ) ) );
      char message[ 16 ] = "0";
      asio::ip::udp::endpoint sender_ep;

      while( true )
      {
         m_Socket.receive_from( asio::buffer( message, 16 ), sender_ep );
         cout << "Client: " << message << endl;
         memset( &message[ 0 ], 0, 16 );
      }
   }
};
```

Main.cpp


```
#include <iostream>
#include <boost/asio.hpp>
#include "MyClient.h"

using namespace boost;
using namespace std;

int main()
{
	cout << "Starting client..." << endl;

	boost::asio::io_service io_service;

	MyClient c( io_service );
	c.start();

	return 0;
}
```


2. Boost Sockets: TCP

This example uses TCP and the synchronous coding model in the C++ server and a C# client.

```
#include <string>
#include <iostream>
#include <boost/asio.hpp>
#include <boost/thread.hpp>
#include <boost/lexical_cast.hpp>
#include <boost/date_time/posix_time/posix_time_types.hpp>

using namespace boost;
using namespace std;

const short m_Port = 30002;
const long m_Pause = 1000;

class MyServer
{
private:
	asio::ip::tcp::endpoint m_Endpoint;
	asio::ip::tcp::socket m_Socket;
	asio::ip::tcp::acceptor m_Acceptor;

	int count;
	string message;

public:
	MyServer( asio::io_service &m_Service ) :
			m_Endpoint( asio::ip::tcp::v4(), m_Port ),
			m_Socket( m_Service ),
			m_Acceptor( m_Service, m_Endpoint ),
			count( 0 )
	{
	}
	void start()
	{
		cout << "Waiting for incoming connection request..." << endl;
		m_Acceptor.accept( m_Socket );
		try
		{
			cout << "Connection request received..." << endl;
			cout << "Transmitting data..." << endl;

			while( true )
			{
				message = lexical_cast< string >( count++ );
				m_Socket.send( asio::buffer( message ) );
				this_thread::sleep( posix_time::milliseconds(
							m_Pause ) );
			}
		}
		catch( std::exception &e )
		{
			cout << "Exception: " << e.what() << "\n";
```



```
            }
            cout << "Disconnected." << endl;
            m_Socket.close();
      }
};
```

Main.cpp

```
#include <iostream>
#include <boost/asio.hpp>
#include "Server.h"

int main(int argc, char* argv[])
{
      cout << "Starting server..." << endl;

      boost::asio::io_service m_Service;

      MyServer s( m_Service );
      s.start();

      return 0;
}
```

C#: MyClient.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Net;
using System.Net.Sockets;

namespace Clent_Example
{
    class MyClient
    {
        private Socket m_Socket;
        private IPEndPoint m_EP;

        public MyClient( string IPAdd, int m_Port )
        {
            m_EP = new IPEndPoint( IPAddress.Parse( IPAdd ), m_Port );
            m_Socket = new Socket( AddressFamily.InterNetwork,
                                   SocketType.Stream,
                                   ProtocolType.Tcp );
        }

        public void Connect()
        {
            m_Socket.Connect( m_EP );
```



```
            Console.WriteLine("Connected to server...");

            Byte[] m_Buffer = new Byte[ 4 ];

            while( true )
            {
                 try
                 {
                     m_Socket.Receive( m_Buffer );
                     Console.WriteLine( System.Text.Encoding.ASCII.GetString(
                                        m_Buffer ));
                 }
                 catch( Exception ex )
                 {
                     Console.WriteLine( ex.Message );
                     break;
                 }
             }
             m_Socket.Shutdown( SocketShutdown.Both );
             m_Socket.Close();
        }
    }
}
```

C#: Main.cpp

```
using System;

namespace Clent_Example
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Attempting to connect to server...");
            MyClient c = new MyClient( "127.0.0.1", 30002 );
            c.Connect();
        }
     }
}
```

See the Boost website at http://www.boost.org/doc/libs/ for more information on sockets.

3. Boost Threads

Boost threads can be started with static methods or callable objects. A callable object is one that overloads the () operator. That is, a callable object is a function object.

MyClass.h

```
#include <iostream>
#include <boost/thread.hpp>
#include <boost/date_time/posix_time/posix_time_types.hpp>

using namespace std;
using namespace boost;

class MyClass
{
public:
	static void Go()
	{
		this_thread::sleep( posix_time::milliseconds( 2000 ) );
		cout << "Hello from thread 1.\n";
	}

	void operator()()
	{
		this_thread::sleep( posix_time::milliseconds( 2000 ) );
		cout << "Hello from thread 2.\n";
	}
};
```

Main.cpp

```
#include <iostream>
#include <boost/thread.hpp>
#include <boost/date_time/posix_time/posix_time_types.hpp>

using namespace std;
using namespace boost;

int main()
{
    // Starting a thread with a static method.
    thread m_Thread_1( &MyClass::Go );

    // Starting a thread with a callable object.
    MyClass c;
    thread m_Thread_2( c );

    cout << "The main thread is going to sleep.\n";
    this_thread::sleep( posix_time::milliseconds( 3000 ) );
    cout << "The main thread woke up.\n";
```



```
    return 0;
}
```

See the Boost website at http://www.boost.org/doc/libs/ for more information on threads.

4. Boost Shared Memory

The fastest way to communicate data between processes (or programs) is though **shared memory**. Shared memory works through the operating system, which maps a chunk of memory in space accessible to the different programs. Thus, the different processes can read and write data in that memory space.

We can use shared memory in Windows, but it works differently from UNIX. In Windows, shared memory is automatically destroyed when the last process attached to it is done running. In Boost, there is a special class for called windows_shared_memory.

Process_1.cpp

```
#include <iostream>
#include <boost/interprocess/windows_shared_memory.hpp>
#include <boost/interprocess/mapped_region.hpp>

using namespace std;
using namespace boost::interprocess;

int main ()
{
      windows_shared_memory m_Shared( create_only, "shared memory",
                                      read_write, 1 );
      mapped_region m_Region( m_Shared, read_write );

      // Write 7 to the shared memory location
      std::memset( m_Region.get_address(), 7, 1 );

      // Wait for the other process to pick up the data.
      while ( 1 ) {}
      return 0;
}
```

Process_2.cpp

```
#include <iostream>
#include <boost/interprocess/windows_shared_memory.hpp>
#include <boost/interprocess/mapped_region.hpp>

using namespace std;
using namespace boost::interprocess;

int main ()
{
  windows_shared_memory m_Shared( open_only, "shared_memory", read_only );
  mapped_region m_Region( m_Shared, read_only );

  // Retrieve the data.
  int *mem = static_cast< int * >( m_Region.get_address() );
  cout << *mem << endl;
```



```
  return 0;
}
```

See the Boost website at http://www.boost.org/doc/libs/ for more information on shared memory.

5. Boost Random Numbers

The Boost random number capability uses the Mersenne Twister algorithm to generate uniform variates over the range 0 to 1.

```
#include <iostream>
#include <boost/random/mersenne_twister.hpp>
#include <boost/random/variate_generator.hpp>

using namespace std;
using namespace boost;

int main()
{
   // mt19937 is a typedef of the class containing the Mersenne twister
   // algorithm.
   boost::mt19937 m_Algorithm;

   // Distirubtion is uniform from 0 to 1.
   boost::uniform_01<> m_Distribution;

   // Create a random number generator object with the algorithm and
   // distribution.
   boost::variate_generator< mt19937, uniform_01<> > m_Generator(
                                          m_Algorithm, m_Distribution );

   for ( int i = 0; i < 100; i++ )
   {
      cout << m_Generator() << endl;
   }
   return 0;
}
```

Many other distributions are supported in Boost, including the normal, lognormal, binomial and exponential, among others. Here is a normal example.

```
#include <iostream>
#include <boost/random/normal_distribution.hpp>
#include <boost/random/mersenne_twister.hpp>
#include <boost/random/variate_generator.hpp>

using namespace std;
using namespace boost;

int main()
{
   // Distirubtion is normal with mu = 0 and sigma = 1.
   boost::normal_distribution<> m_Distribution( 0, 1 );
   boost::mt19937 m_Algorithm;

   boost::variate_generator< mt19937, boost::normal_distribution<> >
                     m_Generator( m_Algorithm, m_Distribution );
```



```
   for ( int i = 0; i < 100; i++ )
   {
      cout << m_Generator() << endl;
   }
   return 0;
}
```

See the Boost website at http://www.boost.org/doc/libs/ for more information on generating random numbers.

6. Boost Math

```
#include <iostream>
#include <boost/math/distributions/normal.hpp>

using namespace std;

int main()
{
      boost::math::normal s;  // (Default mu = 0 and sigma = 1)
      cout << s.mean() << endl;
      cout << s.standard_deviation() << endl;
      cout << setprecision( 10 ) << boost::math::pdf( s, 1.0 ) << endl;
      cout << quantile(s, 0.95) << endl;

      return 0;
}
```

See the Boost website at http://www.boost.org/doc/libs/ for more information on the math toolkit.

7. Boost Date Time

The **ptime** class can hold a date and time point. The local time can be retrieved as in this example with sub-second resolution. Windows systems usually do not enable microsecond resolution.

```
#include <iostream>
#include <boost/date_time/posix_time/posix_time.hpp>

using namespace std;
using namespace boost::posix_time;

int main()
{
      ptime t( microsec_clock::local_time() );
      cout << t.date() << endl;
      cout << t.time_of_day() << endl;

      return 0;
}
```

See the Boost website at http://www.boost.org/doc/libs/ for more information on dates and times.

8. Boost ForEach

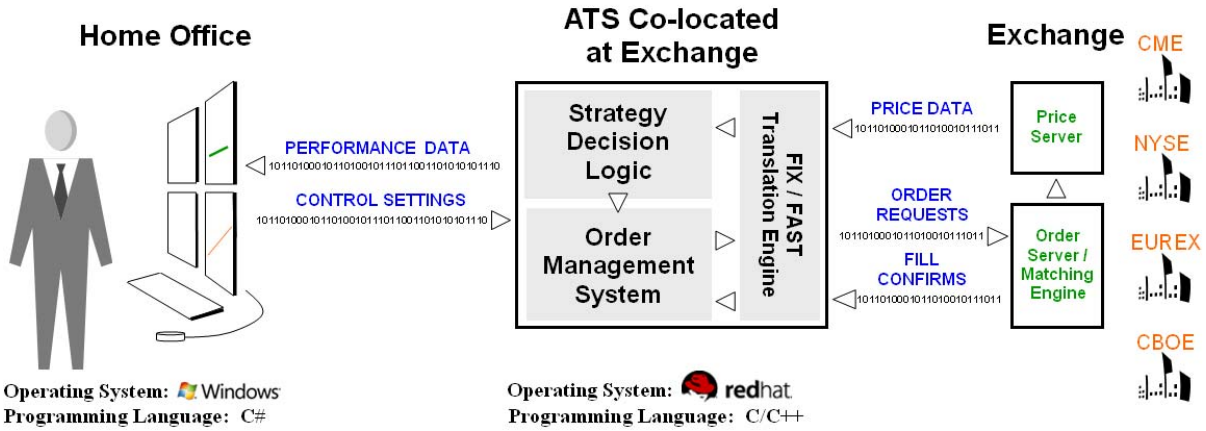
Boost for each makes things a lot easier. Notice the use of the #define in this example.

```
#include <iostream>
#include <list>
#include <boost/foreach.hpp>

using namespace std;
using namespace boost;

int main()
{
      #define foreach BOOST_FOREACH

      list< int > x;
      x.push_back( 5 );
      x.push_back( 7 );
      x.push_back( 3 );
      x.push_back( 9 );

      foreach( int i, x )
      {
            cout << i << endl;
      }

      return 0;
}
```

See the Boost website at http://www.boost.org/doc/libs/ for more information on foreach.

CHAPTER 11: AUTOMATED TRADING SYSTEMS

For automated trading systems, the reference architecture I will use is shown in the following graphic:



In order to implement this architecture, we need several network connections:

- C++ UDP client to accept incoming FAST data from exchange server.
- C++ TCP server to send data to home office. (2-way)
- C# TCP client to accept incoming data from co-located server. (2-way)
- C++ TCP client to transmit FIX to exchange server (2-way)