BMT 227

C++

Creating an Executable File with the Linker

- The steps to create an executable file are:

1. Create a source code file, with a .CPP extension.
2. Compile the source code into a file with the .OBJ extension.
3. Link your OBJ file with any needed libraries to produce an executable program.

HELLO.CPP
Your First C++ Program

```
# include <iostream.h>

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

Q&A

- **Q. What is the difference between a text editor and a word processor?**
- **Q. If my compiler has a built-in editor, must I use it?**
- **Q. Can I ignore warning messages from my compiler?**
- **Q. What is compile time?**

Quiz

1. What is the difference between an interpreter and a compiler?
2. How do you compile the source code with your compiler?
3. What does the linker do?
4. What are the steps in the normal development cycle?

Listing 2.2.Using cout.

```
1: // Listing 2.2 using cout
2:
3: #include <iostream.h>
4: int main()
5: {
6:          cout << "Hello there.\n";
7:          cout << "Here is 5: " << 5 << "\n";
8:          cout << "The manipulator endl writes a new line to the screen." << endl;
9:          cout << "Here is a very big number:\t" << 70000 << endl;
10:         cout << "Here is the sum of 8 and 5:\t" << 8+5 << endl;
11:                 cout << "Here's a fraction:\t\t" << (float) 5/8 << endl;
12:                 cout << "And a very very big number:\t" << (double) 7000* 7000 << endl;
13:                 cout << "Don't forget to replace Jesse Liberty with yourname...\n";
14:         cout << "Jesse Liberty is a C++ programmer!\n";
15:         return 0;
16: }
```

Listing 2.2 Output

```
Hello there.
Here is 5: 5
The manipulator endl writes a new line to the screen.
Here is a very big number:  70000
Here is the sum of 8 and 5:         13
Here's a fraction:                  0.625
And a very very big number:         4.9e+07
Don't forget to replace Jesse Liberty with your name...
Jesse Liberty is a C++ programmer!
```

Comments

- **Types of Comments**

C++ comments come in two flavors:
the double-slash (//) comment, and the slash-star (/*) comment.

Using Comments

As a general rule, the overall program should have comments at the beginning, telling you what the program does. Each function should also have comments explaining what the function does and what values it returns. Finally, any statement in your program that is obscure or less than obvious should be commented as well.

- The name of the function or program.
- The name of the file.
- What the function or program does.
- A description of how the program works.
- The author's name.
- A revision history (notes on each change made).
- What compilers, linkers, and other tools were used to make the program.
- Additional notes as needed.

Functions

While main() is a function, it is an unusual one. Typical functions are called, or invoked, during the course of your program. A program is executed line by line in the order it appears in your source code, until a function is reached. Then the program branches off to execute the function. When the function finishes, it returns control to the line of code immediately following the call to the function.

Listing 2.4. Demonstrating a call to a function.

```
1: #include <iostream.h>
2:
3:   // function Demonstration Function
4:   // prints out a useful message
5:   void DemonstrationFunction()
6: {
7:              cout << "In Demonstration Function\n";
8: }
9:
10: // function main - prints out a message, then
11: // calls DemonstrationFunction, then prints out
12: // a second message.
13: int main()
14: {
15:              cout << "In main\n" ;
16:              DemonstrationFunction();
17:              cout << "Back in main\n";
18:              return 0;
19: }
```

Listing 2.4 Output

In main

In Demonstration Function

Back in main

Using Functions

Functions either return a value or they return void, meaning they return nothing. A function that adds two integers might return the sum, and thus would be defined to return an integer value. A function that just prints a message has nothing to return and would be declared to return void.

Using Functions

Functions consist of a header and a body. The header consists, in turn, of the return type, the function name, and the parameters to that function.

The parameters to a function allow values to be passed into the function. Thus, if the function were to add two numbers, the numbers would be the parameters to the function. Here's a typical function header:

```
int Sum(int a, int b)
```

- The body of a function consists of an opening brace, zero or more statements, and a closing brace.
- The statements constitute the work of the function. A function may return a value, using a return statement. This statement will also cause the function to exit. If you don't put a return statement into your function, it will automatically return void at the end of the function. The value returned must be of the type declared in the function header.

Listing 2.5. FUNC.CPP demonstrates a simple function.

```
1: #include <iostream.h>
2:     int Add (int x, int y)
3: {
4:
5:                 cout << "In Add(), received " << x << " and " << y << "\n";
6:                 return (x+y);
7: }
8:
9:     int main()
10: {
11:                cout << "I'm in main()!\n";
12:   int a, b, c;
13:                cout << "Enter two numbers: ";
14:                cin >> a;
15:                cin >> b;
16:                cout << "\nCalling Add()\n";
17:  c=Add(a,b);
18:                cout << "\nBack in main().\n";
19:                cout << "c was set to " << c;
20:                cout << "\nExiting...\n\n";
21:  return 0;
22: }
```

Listing 2.5 Output

I'm in main()!
Enter two numbers: 3 5
Calling Add()
In Add(), received 3 and 5
Back in main().
c was set to 8
Exiting...

Q&A

- **Q. What does #include do?**
- **Q. What is the difference between // comments and /* style comments?**
- **Q. What differentiates a good comment from a bad comment?**

Quiz

1. What is the difference between the compiler and the preprocessor?
2. Why is the function main() special?
3. What are the two types of comments, and how do they differ?
4. Can comments be nested?
5. Can comments be longer than one line?

Variables and Constants

Programs need a way to store the data they use. Variables and constants offer various ways to represent and manipulate that data.

- How to declare and define variables and constants.
- How to assign values to variables and manipulate those values.
- How to write the value of a variable to the screen.

What Is a Variable?

In C++ a variable is a place to store information. A variable is a location in your computer's memory in which you can store a value and from which you can later retrieve that value.

Size of Integers

On any one computer, each variable type takes up a single, unchanging amount of room. That is, an integer might be two bytes on one machine, and four on another, but on either computer it is always the same, day in and day out.

A char variable (used to hold characters) is most often one byte long. A short integer is two bytes on most computers, a long integer is usually four bytes, and an integer (without the keyword short or long) can be two or four bytes.

Listing 3.1. Determining the size of variable types on your computer.

```
1: #include <iostream.h>
2:
3: int main()
4: {
5:         cout << "The size of an int is:\t\t" << sizeof(int) << " bytes.\n";
6:         cout << "The size of a short int is:\t" << sizeof(short) << " bytes.\n";
7:         cout << "The size of a long int is:\t" << sizeof(long) << " bytes.\n";
8:         cout << "The size of a char is:\t\t" << sizeof(char) << " bytes.\n";
9:         cout << "The size of a float is:\t\t" << sizeof(float) << " bytes.\n";
10:        cout << "The size of a double is:\t" << sizeof(double) << " bytes.\n";
11:
12:        return 0;
13: }
```

Listing 3.1 Output

```
The size of an int is:          4 bytes.
The size of a short int is:     2 bytes.
The size of a long int is:      4 bytes.
The size of a char is:          1 bytes.
The size of a float is:         4 bytes.
The size of a double is:        8 bytes.
```

signed and unsigned

All integer types come in two varieties: signed and unsigned.

The idea here is that sometimes you need negative numbers, and sometimes you don't. Integers (short and long) without the word "unsigned" are assumed to be signed. Signed integers are either negative or positive. Unsigned integers are always positive.

Because you have the same number of bytes for both signed and unsigned integers, the largest number you can store in an unsigned integer is twice as big as the largest positive number you can store in a signed integer.

An unsigned short integer can handle numbers from 0 to 65,535.

Half the numbers represented by a signed short are negative, thus a signed short can only represent numbers from -32,768 to 32,767.

Fundamental Variable Types

Type	*Size*	*Values*
unsigned short int	2 bytes	0 to 65,535
short int	2 bytes	-32,768 to 32,767
unsigned long int	4 bytes	0 to 4,294,967,295
long int	4 bytes	-2,147,483,648 to 2,147,483,647
int (16 bit)	2 bytes	-32,768 to 32,767
int (32 bit)	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned int (16 bit)	2 bytes	0 to 65,535
unsigned int (32 bit)	2 bytes	0 to 4,294,967,295
char	1 byte	256 character values
float	4 bytes	1.2e-38 to 3.4e38
double	8 bytes	2.2e-308 to 1.8e308

Defining a Variable

You create or define a variable by stating its type, followed by one or more spaces, followed by the variable name and a semicolon. The variable name can be virtually any combination of letters, but cannot contain spaces.

As a general programming practice, avoid such horrific names as J23qrsnf, and restrict single letter variable names (such as x or i) to variables that are used only very briefly.

Try to use expressive names such as myAge or howMany.

Case Sensitivity

C++ is case-sensitive. In other words, uppercase and lowercase letters are considered to be different.

A variable named age is different from Age, which is different from AGE.

Keywords

Some words are reserved by C++, and you may not use them as variable names.

These are keywords used by the compiler to control your program.

Keywords include if, while, for, and main.

Creating More Than One Variable at a Time

You can create more than one variable of the same type in one statement by writing the type and then the variable names, separated by commas.

For example:

```
unsigned int myAge, myWeight; // two unsigned int variables
long area, width, length; // three longs
```

Assigning Values to Your Variables

You assign a value to a variable by using the assignment operator (=). Thus, you would assign 5 to Width by writing:

```
unsigned short Width;
Width = 5;
```

You can combine these steps and initialize Width when you define it by writing

```
unsigned short Width = 5;
```

Just as you can define more than one variable at a time, you can initialize more than one variable at creation. For example:

```
// create two long variables and initialize them
long width = 5, length = 7;
```

This example initializes the long integer variable width to the value 5 and the long integer variable length to the value 7.

You can even mix definitions and initializations:

```
int myAge = 39, yourAge, hisAge = 40;
```

typedef

It can become tedious, repetitious, and, most important, error-prone to keep writing unsigned short int.

C++ enables you to create an alias for this phrase by using the keyword typedef, which stands for type definition.

In effect, you are creating a synonym, and it is important to distinguish this from creating a new type.

typedef is used by writing the keyword typedef, followed by the existing type and then the new name. For example:

```
typedef unsigned short int USHORT
```

Listing 3.3. A demonstration of typedef.

```
1: // *****************
2: // Demonstrates typedef keyword
3: #include <iostream.h>
4:
5: typedef unsigned short int USHORT; //typedef defined
6:
7: void main()
8: {
9: USHORT Width = 5;
10: USHORT Length;
11: Length = 10;
12: USHORT Area = Width * Length;
13: cout << "Width:" << Width << "\n";
14: cout << "Length: " << Length << endl;
15: cout << "Area: " << Area <<endl;
16: }
```

Output:
Width:5
Length: 10
Area: 50

When to Use short and When to Use long

If there is any chance that the value you'll want to put into your variable will be too big for its type, use a larger type.

unsigned short integers, assuming that they are two bytes, can hold a value only up to 65,535.

Signed short integers can hold only half that. Although unsigned long integers can hold an extremely large number (4,294,967,295) that is still quite finite.

If you need a larger number, you'll have to go to float or double.

Wrapping Around an unsigned Integer

The fact that unsigned long integers have a limit to the values they can hold is only rarely a problem, but what happens if you do run out of room?

When an unsigned integer reaches its maximum value, it wraps around and starts over, much as a car odometer might.

Listing 3.4.A demonstration of putting too large a value in an unsigned integer.

```
1: #include <iostream.h>
2: int main()
3: {
4: unsigned short int smallNumber;
5: smallNumber = 65535;
6: cout << "small number:" << smallNumber << endl;
7: smallNumber++;
8: cout << "small number:" << smallNumber << endl;
9: smallNumber++;
10: cout << "small number:" << smallNumber << endl;
11: return 0;
12: }

Output:
small number:65535
small number:0
small number:1
```

Wrapping Around a signed Integer

A signed integer is different from an unsigned integer, in that half of the values you can represent are negative. Instead of picturing a traditional car odometer, you might picture one that rotates up for positive numbers and down for negative numbers. One mile from 0 is either 1 or -1.

When you run out of positive numbers, you run right into the largest negative numbers and then count back down to 0.

Listing 3.5. A demonstration of adding too large a number to a signed integer.

```
1: #include <iostream.h>
2: int main()
3: {
4: short int smallNumber;
5: smallNumber = 32767;
6: cout << "small number:" << smallNumber << endl;
7: smallNumber++;
8: cout << "small number:" << smallNumber << endl;
9: smallNumber++;
10: cout << "small number:" << smallNumber << endl;
11: return 0;
12: }
```

Output:

```
small number:32767
small number:-32768
small number:-32767
```

Characters

Character variables (type char) are typically 1 byte, enough to hold 256 values.

A char can be interpreted as a small number (0-255) or as a member of the ASCII set (American Standard Code for Information Interchange).

Characters and Numbers

When you put a character, for example, 'a', into a char variable, what is really there is just a number between 0 and 255. The compiler knows, however, how to translate back and forth between characters (represented by a single quotation mark and then a letter, numeral, or punctuation mark, followed by a closing single quotation mark) and one of the ASCII values.

The value/letter relationship is arbitrary; there is no particular reason that the lowercase "a" is

assigned the value 97.

It is important to realize, however, that there is a big difference between the value 5 and the character `5'. The latter is actually valued at 53, much as the letter `a' is valued at 97.

Listing 3.6. Printing characters based on numbers

```
1: #include <iostream.h>
2: int main()
3: {
4: for (int i = 32; i<128; i++)
5: cout << (char) i;
6: return 0;
7: }
```

Output:

```
!"#$%G'()*+,./0123456789:;<>?@ABCDEFGHIJKLMNOP_QRSTUVWXYZ[\]^'
abcdefghijklmnopqrstuvwxyz<|>~s
```

This simple program prints the character values for the integers 32 through 127.

Constants

Like variables, constants are data storage locations. Unlike variables, and as the name implies, constants don't change. You must initialize a constant when you create it, and you cannot assign a new value later.

Constants

Literal Constants

A literal constant is a value typed directly into your program wherever it is needed.

Symbolic Constants

A symbolic constant is a constant that is represented by a name, just as a variable is represented.

Unlike a variable, however, after a constant is initialized, its value can't be changed.

Constants

- **Literal Constants**

  int myAge = 39;

  myAge is a variable of type int; 39 is a literal constant. You can't assign a value to 39, and its value can't be changed.
- **Symbolic Constants**

  students = classes * 15;

  In this example, 15 is a literal constant. Your code would be easier to read, and easier to maintain, if you substituted a symbolic constant for this value:

  students = classes * studentsPerClass

  If you later decided to change the number of students in each class, you could do so where you define the constant studentsPerClass without having to make a change every place you used that value.

Constants

There are two ways to declare a symbolic constant in C++.

The old, traditional, and now obsolete way is with a preprocessor directive:

```
#define studentsPerClass 15
```

Defining Constants with const Although #define works, there is a new, much better way to define constants in C++:

```
const unsigned short int studentsPerClass = 15;
```

This method has several advantages in making your code easier to maintain and in preventing bugs. The biggest difference is that this constant has a type, and the compiler can enforce that it is used according to its type.

Enumerated Constants

Enumerated constants enable you to create new types and then to define variables of those types whose values are restricted to a set of possible values.

For example, you can declare COLOR to be an enumeration, and you can define that there are five values for COLOR: RED, BLUE, GREEN, WHITE, and BLACK.

```
enum COLOR { RED, BLUE, GREEN, WHITE, BLACK };
```

- It makes COLOR the name of an enumeration, that is, a new type.
- **2.** It makes RED a symbolic constant with the value 0, BLUE a symbolic constant with the value 1, GREEN a symbolic constant with the value 2, and so forth.

Every enumerated constant has an integer value. If you don't specify otherwise, the first constant will have the value 0, and the rest will count up from there. Any one of the constants can be initialized with a particular value, however, and those that are not initialized will count upward from the ones before them.

Thus, if you write

enum Color { RED=100, BLUE, GREEN=500, WHITE, BLACK=700 };

then RED will have the value 100; BLUE, the value 101; GREEN, the value 500; WHITE, the value 501; and BLACK, the value 700.

Listing 3.7. A demonstration of enumerated constants.

```
#include <iostream.h>
int main()
{
enum Days { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday };

Days DayOff;
 int x;

cout << "What day would you like off (0-6)? ";
cin >> x;
DayOff = Days(x);

        if (DayOff == Sunday || DayOff == Saturday)
        cout << "\nYou're already off on weekends!\n";
                else
                cout << "\nOkay, I'll put in the vacation day.\n";
return 0;
}
```

Q&A

- **Q. If a short int can run out of room and wrap around, why not always use long integers?**
- **Q. What happens if I assign a number with a decimal point to an integer rather than to a float?**
- **Q. Why not use literal constants; why go to the bother of using symbolic constants?**
- **Q. What happens if I assign a negative number to an unsigned variable?**
- **Q. Can I work with C++ without understanding bit patterns, binary arithmetic, and hexadecimal?**

Quiz

1. What is the difference between an integral variable and a floating-point variable?
2. What are the differences between an unsigned short int and a long int?
3. What are the advantages of using a symbolic constant rather than a literal constant?
4. What are the advantages of using the const keyword rather than #define?
5. What makes for a good or bad variable name?
6. Given this enum, what is the value of BLUE?

```
enum COLOR { WHITE, BLACK = 100, RED, BLUE, GREEN = 300 };
```

7. Which of the following variable names are good, which are bad, and which are invalid?

a. Age
b. !ex
c. R79J
d. TotalIncome
e. __Invalid

Statements

One of the most common statements is the following assignment statement:

```
x = a + b;
```

- Unlike in algebra, this statement does not mean that x equals a+b.
- This is read, "Assign the value of the sum of a and b to x," or "Assign to x, a+b." Even though this statement is doing two things, it is one statement and thus has one semicolon. The assignment operator assigns whatever is on the right side of the equal sign to whatever is on the left side.

Whitespace

Whitespace (tabs, spaces, and newlines) is generally ignored in statements. The assignment statement previously discussed could be written as

```
x=a+b;
```

- or as

```
x          =a
+          b          ;
```

- Although this last variation is perfectly legal, it is also perfectly foolish.
- Whitespace can be used to make your programs more readable and easier to maintain, or it can be used to create horrific and indecipherable code.

Expressions

- Anything that evaluates to a value is an expression in C++. An expression is said to return a value.
- Thus, 3+2; returns the value 5 and so is an expression.
- All expressions are statements.

Listing 4.1. Evaluating complex expressions.

```
#include <iostream.h>
int main()
{
        int a=0, b=0, x=0, y=35;
        cout << "a: " << a << " b: " << b;
        cout << " x: " << x << " y: " << y << endl;
        a = 9;
        b = 7;
        y = x = a+b;
        cout << "a: " << a << " b: " << b;
        cout << " x: " << x << " y: " << y << endl;
                return 0;
}
```

Output:

```
a: 0 b: 0 x: 0 y: 35
a: 9 b: 7 x: 16 y: 16
```

Operators

An operator is a symbol that causes the compiler to take an action. Operators act on operands, and in C++ all operands are expressions. In C++ there are several different categories of operators.

Two of these categories are

- Assignment operators.
- Mathematical operators.

Assignment Operator

The assignment operator (=) causes the operand on the left side of the assignment operator to have its value changed to the value on the right side of the assignment operator. The expression

```
x = a + b;
```

assigns the value that is the result of adding a and b to the operand x.

- An operand that legally can be on the left side of an assignment operator is called an lvalue. That which can be on the right side is called (you guessed it) an rvalue.
- Constants are r-values. They cannot be l-values. Thus, you can write

```
x = 35; // ok
```

- but you can't legally write

```
35 = x; // error, not an lvalue!
```

Mathematical Operators

- There are five mathematical operators: addition (+), subtraction (-), multiplication (*), division (/), and modulus (%).
- Addition and subtraction work as you would expect, although subtraction with unsigned integers can lead to surprising results, if the result is a negative number.

Integer Division and Modulus

- Integer division is somewhat different from everyday division. When you divide 21 by 4, the result is a real number (a number with a fraction). Integers don't have fractions, and so the "remainder" is lopped off.
- The answer is therefore 5. To get the remainder, you take 21 modulus 4 (21 % 4) and the result is 1. The modulus operator tells you the remainder after an integer division.
- Finding the modulus can be very useful. For example, you might want to print a statement on every 10th action. Any number whose value is 0 when you modulus 10 with that number is an exact multiple of 10. Thus 1 % 10 is 1, 2 % 10 is 2, and so forth, until 10 % 10, whose result is 0. 11 % 10 is back to 1, and this pattern continues until the next multiple of 10, which is 20.

WARNING

Many novice C++ programmers inadvertently put a semicolon after their if statements:

```
if(SomeValue < 10);
SomeValue = 10;
```

What was intended here was to test whether SomeValue is less than 10, and if so, to set it to 10, making 10 the minimum value for SomeValue.

Running this code snippet will show that SomeValue is always set to 10! Why?

The if statement terminates with the semicolon (the do-nothing operator). Remember that indentation has no meaning to the compiler.

This snippet could more accurately have been written as:

```
if (SomeValue < 10)        // test
;                          // do nothing
SomeValue = 10;            // assign
```

Removing the semicolon will make the final line part of the if statement and will make this code do what was intended.

Combining the Assignment and Mathematical Operators

It is not uncommon to want to add a value to a variable, and then to assign the result back into the variable. If you have a variable myAge and you want to increase the value by two, you can write

```
int myAge = 5;
int temp;
temp = myAge + 2; // add 5 + 2 and put it in temp
myAge = temp; // put it back in myAge
```

This method, however, is terribly convoluted and wasteful. In C++, you can put the same variable on both sides of the assignment operator, and thus the preceding becomes

```
myAge = myAge + 2;
```

which is much better. In algebra this expression would be meaningless, but in C++ it is read as "add two to the value in myAge and assign the result to myAge."

Even simpler to write, but perhaps a bit harder to read is:

```
myAge += 2;
```

The self-assigned addition operator (+=) adds the rvalue to the lvalue and then reassigns the result into the lvalue. This operator is pronounced "plus-equals."

There are self-assigned subtraction (-=), division (/=), multiplication (*=), and modulus (%=) operators as well.

Increment and Decrement

The most common value to add (or subtract) and then reassign into a variable is 1. In C++, increasing a value by 1 is called incrementing, and decreasing by 1 is called decrementing. There are special operators to perform these actions.

- The increment operator (++) increases the value of the variable by 1, and the decrement operator (--) decreases it by 1. Thus, if you have a variable, C, and you want to increment it, you would use this statement:

```
C++;     // Start with C and increment it.
```

- This statement is equivalent to the more verbose statement

```
C = C + 1;
```

- which you learned is also equivalent to the moderately verbose statement

```
C += 1;
```

Prefix and Postfix

- Both the increment operator (++) and the decrement operator(--) come in two varieties: prefix and postfix.
- The prefix variety is written before the variable name (++myAge); the postfix variety is written after (myAge++).
- In a simple statement, it doesn't much matter which you use, but in a complex statement, when you are incrementing (or decrementing) a variable and then assigning the result to another variable, it matters very much.
- The prefix operator is evaluated before the assignment, the postfix is evaluated after.
- The semantics of prefix is this: Increment the value and then fetch it. The semantics of postfix is different: Fetch the value and then increment the original.

This can be confusing at first, but if x is an integer whose value is 5 and you write

```
int a = ++x;
```

you have told the compiler to increment x (making it 6) and then fetch that value and assign it to a.

Thus, a is now 6 and x is now 6.

If, after doing this, you write

```
int b = x++;
```

you have now told the compiler to fetch the value in x (6) and assign it to b, and then go back and increment x. Thus, b is now 6, but x is now 7.

Listing 4.3. A demonstration of prefix and postfix operators.

```
// Listing 4.3 - demonstrates use of
// prefix and postfix increment and
// decrement operators
#include <iostream.h>
int main()
{
    int myAge = 39; // initialize two integers
     int yourAge = 39;
    cout << "I am: " << myAge << " years old.\n";
    cout << "You are: " << yourAge << " years old\n";
    myAge++; // postfix increment
    ++yourAge; // prefix increment
    cout << "One year passes...\n";
    cout << "I am: " << myAge << " years old.\n";
    cout << "You are: " << yourAge << " years old\n";
     cout << "Another year passes\n";
    cout << "I am: " << myAge++ << " years old.\n";
    cout << "You are: " << ++yourAge << " years old\n";
    cout << "Let's print it again.\n";
    cout << "I am: " << myAge << " years old.\n";
    cout << "You are: " << yourAge << " years old\n";
            return 0;
}
```

Listing 4.3 Output

I am 39 years old
You are 39 years old
One year passes
I am 40 years old
You are 40 years old
Another year passes
I am 40 years old
You are 41 years old
Let's print it again
I am 41 years old
You are 41 years old

Precedence

- In the complex statement:

  x = 5 + 3 * 8;
- which is performed first, the addition or the multiplication?
- If the addition is performed first, the answer is 8 * 8, or 64. If the multiplication is performed first, the answer is 5 + 24, or 29.
- Every operator has a precedence value, Multiplication has higher precedence than addition, and thus the value of the expression is 29.
- When two mathematical operators have the same precedence, they are performed in left-to-right order.
- Thus

  x = 5 + 3 + 8 * 9 + 6 * 4;
- is evaluated multiplication first, left to right. Thus, 8*9 = 72, and 6*4 = 24. Now the expression is essentially

  x = 5 + 3 + 72 + 24;
- Now the addition, left to right, is 5 + 3 = 8; 8 + 72 = 80; 80 + 24 = 104.

Precedence

TotalSeconds = NumMinutesToThink + NumMinutesToType * 60

TotalSeconds = (NumMinutesToThink + NumMinutesToType) * 60

Nesting Parentheses

For complex expressions, you might need to nest parentheses one within another. For example, you might need to compute the total seconds and then compute the total number of people who are involved before multiplying seconds times people:

```
TotalPersonSeconds = ( ( (NumMinutesToThink + NumMinutesToType) * 60)
* (PeopleInTheOffice + PeopleOnVacation) )
```

```
TotalMinutes = NumMinutesToThink + NumMinutesToType;
TotalSeconds = TotalMinutes * 60;
TotalPeople = PeopleInTheOffice + PeopleOnVacation;
TotalPersonSeconds = TotalPeople * TotalSeconds;
```

The Nature of Truth

In C++, zero is considered false, and all other values are considered true, although true is usually represented by 1. Thus, if an expression is false, it is equal to zero, and if an expression is equal to zero, it is false. If a statement is true, all you know is that it is nonzero, and any nonzero statement is true.

Relational Operators

- The relational operators are used to determine whether two numbers are equal, or if one is greater or less than the other. Every relational statement evaluates to either 1 (TRUE) or 0 (FALSE).
- If the integer variable myAge has the value 39, and the integer variable yourAge has the value 40,
- you can determine whether they are equal by using the relational "equals" operator:

```
myAge == yourAge;
// is the value in myAge the same as in yourAge?
```

- This expression evaluates to 0, or false, because the variables are not equal.
- The expression

```
myAge > yourAge; // is myAge greater than yourAge?
```

evaluates to 0 or false.

There are six relational operators: equals (==), less than (<), greater than (>), less than or equal to (<=), greater than or equal to (>=), and not equals (!=).

Name	*Operator*	*Sample*	*Evaluates*
Equals	==	100 == 50;	false
		50 == 50;	true
Not Equals	!=	100 != 50;	true
		50 != 50;	false
Greater Than	>	100 > 50;	true
		50 > 50;	false
Greater Than or Equals	>=	100 >= 50;	true
		50 >= 50;	true
Less Than	<	100 < 50;	false
		50 < 50;	false
Less Than or Equals	<=	100 <= 50;	false
		50 <= 50;	true

The if Statement

- Normally, your program flows along line by line in the order in which it appears in your source code.
- The if statement enables you to test for a condition (such as whether two variables are equal) and branch to different parts of your code, depending on the result.
- The simplest form of an if statement is this:

```
if (expression)
        statement;
```

- The expression in the parentheses can be any expression at all, but it usually contains one of the relational expressions. If the expression has the value o, it is considered false, and the statement is skipped. If it has any nonzero value, it is considered true, and the statement is executed.
- Consider the following example:

```
if (bigNumber > smallNumber)
        bigNumber = smallNumber;
```

- Because a block of statements surrounded by braces is exactly equivalent to a single statement, the following type of branch can be quite large and powerful:

```
if (expression)
{
        statement1;
        statement2;
        statement3;
}
```

- Here's a simple example of this usage:

```
if (bigNumber > smallNumber)
{
        bigNumber = smallNumber;
        cout << "bigNumber: " << bigNumber << "\n";
        cout << "smallNumber: " << smallNumber << "\n";
}
```

Listing 4.4. A demonstration of branching based on relational operators.

```
// Listing 4.4 - demonstrates if statement
// used with relational operators
#include <iostream.h>
int main()
{
int RedSoxScore, YankeesScore;
cout << "Enter the score for the Red Sox: ";
cin >> RedSoxScore;

cout << "\nEnter the score for the Yankees: ";
cin >> YankeesScore;

cout << "\n";

if (RedSoxScore > YankeesScore)
cout << "Go Sox!\n";

if (RedSoxScore < YankeesScore)
{
cout << "Go Yankees!\n";
cout << "Happy days in New York!\n";
}

if (RedSoxScore == YankeesScore)
{
cout << "A tie? Naah, can't be.\n";
cout << "Give me the real score for the Yanks: ";
cin >> YankeesScore;

if (RedSoxScore > YankeesScore)
cout << "Knew it! Go Sox!";

if (YankeesScore > RedSoxScore)
cout << "Knew it! Go Yanks!";

if (YankeesScore == RedSoxScore)
cout << "Wow, it really was a tie!";
}

cout << "\nThanks for telling me.\n";
return 0;
}
```

else

- Often your program will want to take one branch if y our condition is true, another if it is false.
- The method shown so far, testing first one condition and then the other, works fine but is a bit cumbersome. The keyword **else** can make for far more readable code:

```
if (expression)
        statement;
else
        statement;
```

Listing 4.5. Demonstrating the else keyword.

```
// Listing 4.5 - demonstrates if statement
 // with else clause
#include <iostream.h>
int main()
{
int firstNumber, secondNumber;
 cout << "Please enter a big number: ";
 cin >> firstNumber;
 cout << "\nPlease enter a smaller number: ";
 cin >> secondNumber;
 if (firstNumber > secondNumber)
          cout << "\nThanks!\n";
 else
          cout << "\nOops. The second is bigger!";

 return 0;
 }
```

Advanced if Statements

It is worth noting that any statement can be used in an if or else clause, even another if or else statement. Thus, you might see complex if statements in the following form:

```
if (expression1)
{
        if (expression2)
                statement1;
        else
        {
                if (expression3)
                        statement2;
                else
                        statement3;
        }
}
else
        statement4;
```

Listing 4.6. A complex, nested if statement.

```
// Listing 4.5 - a complex nested
// if statement
#include <iostream.h>
int main()
{
// Ask for two numbers
// Assign the numbers to bigNumber and littleNumber
// If bigNumber is bigger than littleNumber,
// see if they are evenly divisible
// If they are, see if they are the same number

int firstNumber, secondNumber;
cout << "Enter two numbers.\nFirst: ";
cin >> firstNumber;
cout << "\nSecond: ";
cin >> secondNumber;
cout << "\n\n";

if (firstNumber >= secondNumber)
                {
                if ( (firstNumber % secondNumber) == 0) // evenly divisible?
                                {
                                                if (firstNumber == secondNumber)
                                                cout << "They are the same!\n";
                                                else
                                                cout << "They are evenly divisible!\n";
                                }
                 else
                                cout << "They are not evenly divisible!\n";
                }
else
cout << "Hey! The second one is larger!\n";
return 0;
}
```

Listing 4.7. A demonstration of why braces help clarify which else statement goes with which if statement.

```
// Listing 4.7 - demonstrates why braces
// are important in nested if statements
#include <iostream.h>
int main()
{
int x;
cout << "Enter a number less than 10 or greater than 100: ";
cin >> x;
cout << "\n";

                if (x > 10)
                                if (x > 100)
                                                cout << "More than 100, Thanks!\n";
                else // not the else intended!
                                cout << "Less than 10, Thanks!\n";

return 0;
}
```

Output:

```
Enter a number less than 10 or greater than 100: 20
Less than 10, Thanks!
```

Listing 4.8. A demonstration of the proper use of braces with an if statement

```
// Listing 4.8 - demonstrates proper use of braces
// in nested if statements
#include <iostream.h>
int main()
{
int x;
cout << "Enter a number less than 10 or greater than 100: ";
cin >> x;
cout << "\n";

        if (x > 10)
        {               if (x > 100)
                                cout << "More than 100, Thanks!\n";
        }
        else // not the else intended!
                        cout << "Less than 10, Thanks!\n";

return 0;
}

Output:
Enter a number less than 10 or greater than 100: 20
```

- Write a program that takes hours, minutes, seconds as an input and print it out in 24 hours & standard format.

Logical Operators

Often you want to ask more than one relational question at a time. "Is it true that x is greater than y, and also true that y is greater than z?" A program might need to determine that both of these conditions are true, or that some other condition is true, in order to take an action.

Operator	*Symbol*	*Example*
AND	&&	expression1 && expression2
OR	||	expression1 || expression2
NOT	!	!expression

Logical AND

A logical AND statement evaluates two expressions, and if both expressions are true, the logical AND statement is true as well. If it is true that you are hungry, AND it is true that you have money, THEN it is true that you can buy lunch. Thus,

```
if ( (x == 5) && (y == 5) )
```

would evaluate TRUE if both x and y are equal to 5, and it would evaluate FALSE if either one is not equal to 5. Note that both sides must be true for the entire expression to be true.

Logical OR

A logical OR statement evaluates two expressions. If either one is true, the expression is true. If you have money OR you have a credit card, you can pay the bill. You don't need both money and a credit card; you need only one, although having both would be fine as well. Thus,

```
if ( (x == 5) || (y == 5) )
```

evaluates TRUE if either x or y is equal to 5, or if both are.

Logical NOT

A logical NOT statement evaluates true if the expression being tested is false. Again, if the expression being tested is false, the value of the test is TRUE! Thus

```
if ( !(x == 5) )
```

is true only if x is not equal to 5. This is exactly the same as writing

```
if (x != 5)
```

Relational Precedence

Relational operators and logical operators, being C++ expressions, each return a value: 1 (TRUE) or 0 (FALSE). Like all expressions, they have a precedence order (see Appendix A) that determines which relations are evaluated first. This fact is important when determining the value of the statement

```
if ( x > 5 && y > 5 || z > 5)
```

If x is 3, and y and z are both 10,

```
if ( (x > 5) && (y > 5 || z > 5) )
```

More About Truth and Falsehood

- In C++, zero is false, and any other value is true. Because an expression always has a value, many C++ programmers take advantage of this feature in their if statements. A statement such as

```
if (x) // if x is true (nonzero)
x = 0;
```

can be read as "If x has a nonzero value, set it to 0." This is a bit of a cheat; it would be clearer if written

```
if (x != 0) // if x is nonzero
x = 0;
```

- Both statements are legal, but the latter is clearer.

- These two statements also are equivalent:

```
if (!x) // if x is false (zero)
if (x == 0) // if x is zero
```

Conditional (Ternary) Operator

- The conditional operator (?:) is C++'s only ternary operator; that is, it is the only operator to take three terms.
- The conditional operator takes three expressions and returns a value:

  (expression1) ? (expression2) : (expression3)
- This line is read as "If expression1 is true, return the value of expression2; otherwise, return the value of expression3.

```
// Listing 4.9 - demonstrates the conditional operator
#include <iostream.h>
int main()
{
        int x, y, z;
        cout << "Enter two numbers.\n";
        cout << "First: ";
        cin >> x;
        cout << "\nSecond: ";
        cin >> y;
        cout << "\n";

        if (x > y)
        z = x;
                else
                z = y;

        cout << "z: " << z;
        cout << "\n";

        z = (x > y) ? x : y;

        cout << "z: " << z;
        cout << "\n";
return 0;
}
```

Functions

- A function is, in effect, a subprogram that can act on data and return a value. Every C++ program has at least one function, main(). When your program starts, main() is called automatically. main() might call other functions, some of which might call still others.
- Each function has its own name, and when that name is encountered, the execution of the program branches to the body of that function. When the function returns, execution resumes on the next line of the calling function.
- Functions come in two varieties: user-defined and built-in. Built-in functions are part of your compiler package--they are supplied by the manufacturer for your use.

Declaring and Defining Functions

- Using functions in your program requires that you first declare the function and that you then define the function.
- The declaration tells the compiler the name, return type, and parameters of the function.
- The definition tells the compiler how the function works.
- No function can be called from any other function that hasn't first been declared. The declaration of a function is called its prototype.

Declaring the Function

There are three ways to declare a function:

- Write your prototype into a file, and then use the #include directive to include it in your program.
- Write the prototype into the file in which your function is used.
- Define the function before it is called by any other function. When you do this, the definition acts as its own declaration.

Defining the Function

- The definition of a function consists of the function header and its body.
- The header is exactly like the function prototype, except that the parameters must be named, and there is no terminating semicolon.
- The body of the function is a set of statements enclosed in braces.

Functions

- Function Prototype Syntax

  return_type function_name ([type [parameterName]]...);

- Function Definition Syntax

  return_type function_name ([type parameterName]...)
  {
  statements;
  }

Listing 5.1. A function declaration and the definition and use of that function.

```
// Listing 5.1 - demonstrates the use of function prototypes
typedef unsigned short USHORT;
#include <iostream.h>
USHORT FindArea(USHORT length, USHORT width); //function prototype
int main()
{
        USHORT lengthOfYard;
        USHORT widthOfYard;
        USHORT areaOfYard;
        cout << "\nHow wide is your yard? ";
        cin >> widthOfYard;
        cout << "\nHow long is your yard? ";
        cin >> lengthOfYard;
        areaOfYard= FindArea(lengthOfYard,widthOfYard);
        cout << "\nYour yard is ";
        cout << areaOfYard;
        cout << " square feet\n\n";
return 0;
}
USHORT FindArea(USHORT l, USHORT w)
{
return l * w;
}
```

- Write C++ program to find greatest number between three numbers using if-els.

Home Work

- Wright a C++ program to find the area and perimeter of a right triangle.

```
c = a*a + b*b;
z = sqrt(c);
```

Function Prototype Examples

- long FindArea(long length, long width); // returns long, has two parameters
- void PrintMessage(int messageNumber); // returns void, has one parameter
- int GetChoice(); // returns int, has no parameters
- BadFunction(); // returns int, has no parameters

- Every function has a return type. If one is not explicitly designated, the return type will be int. Be sure to give every function an explicit return type. If a function does not return a value, its return type will be void.

Function Definition Examples

```
long Area(long l, long w)
{
        return l * w;
}
void PrintMessage(int whichMsg)
{
        if (whichMsg == 0)
        cout << "Hello.\n";
        if (whichMsg == 1)
        cout << "Goodbye.\n";
        if (whichMsg > 1)
        cout << "I'm confused.\n";
}
```

Listing 5.2. The use of local variables and parameters.

```
#include <iostream.h>

float Convert(float);
int main()
{
float TempFer;
float TempCel;
cout << "Please enter the temperature in Fahrenheit: ";
cin >> TempFer;
TempCel = Convert(TempFer);
cout << "\nHere's the temperature in Celsius: ";
cout << TempCel << endl;
return 0;
}
float Convert(float TempFer)
{
float TempCel;
TempCel = ((TempFer - 32) * 5) / 9;
return TempCel;
}
```

Local and Global Variables

- Not only can you pass in variables to the function, but you also can declare variables within the body of the function. This is done using local variables, so named because they exist only locally within the function itself. When the function returns, the local variables are no longer available.
- Variables defined outside of any function have global scope and thus are available from any function in the program, including main().

Using Functions as Parameters to Functions

- Although it is legal for one function to take as a parameter a second function that returns a value, it can make for code that is hard to read and hard to debug.
- As an example, say you have the functions double(), triple(), square(), and cube(), each of which returns a value. You could write

```
Answer = (double(triple(square(cube(myValue)))));
```

- This statement takes a variable, myValue, and passes it as an argument to the function cube(), whose return value is passed as an argument to the function square(), whose return value is in turn passed to triple(), and that return value is passed to double(). The return value of this doubled, tripled, squared, and cubed number is now passed to Answer.

- It is difficult to be certain what this code does (was the value tripled before or after it was squared?), and if the answer is wrong it will be hard to figure out which function failed.
- An alternative is to assign each step to its own intermediate variable:

```
unsigned long myValue = 2;
unsigned long cubed = cube(myValue); // cubed = 8
unsigned long squared = square(cubed); // squared = 64
unsigned long tripled = triple(squared); // tripled = 196
unsigned long Answer = double(tripled); // Answer = 392
```

- Now each intermediate result can be examined, and the order of execution is explicit.

Return Values

- When the return keyword is encountered, the expression following return is returned as the value of the function. Program execution returns immediately to the calling function, and any statements following the return are not executed.
- It is legal to have more than one return statement in a single function. Listing 5.6 illustrates this idea.

Listing 5.6. A demonstration of multiple return statements.

```
#include <iostream.h>
int Doubler(int AmountToDouble);
int main()
{
        int result = 0;
        int input;

        cout << "Enter a number between 0 and 10,000 to double: ";
        cin >> input;

        cout << "\nBefore doubler is called... ";
        cout << "\ninput: " << input << " doubled: " << result<< "\n";

        result = Doubler(input);

        cout << "\nBack from Doubler...\n";
        cout << "\ninput: " << input << " doubled: " << result << "\n";
return 0;
}
```

```
int Doubler(int original)
{
if (original <= 10000)
return original * 2;
else
return -1;
cout << "You can't get here!\n";
}
```

Output

```
Enter a number between 0 and 10,000 to double: 9000
Before doubler is called...
input: 9000 doubled: 0
Back from doubler...
input: 9000 doubled: 18000
Enter a number between 0 and 10,000 to double: 11000
Before doubler is called...
input: 11000 doubled: 0
Back from doubler...
input: 11000 doubled: -1
```

Overloading Functions

- C++ enables you to create more than one function with the same name. This is called function overloading. The functions must differ in their parameter list, with a different type of parameter, a different number of parameters, or both. Here's an example:

```
int myFunction (int, int);
int myFunction (long, long);
int myFunction (long);
```

- The return types can be the same or different on overloaded functions. You should note that two functions with the same name and parameter list, but different return types, generate a compiler error.

- Function *overloading* i s also called function *polymorphism*. Poly means many, and morph means form: a polymorphic function is many-formed.
- Function polymorphism refers to the ability to "overload" a function with more than one meaning. By changing the number or type of the parameters, you can give two or more functions the same function name, and the right one will be called by matching the parameters used.
- This allows you to create a function that can average integers, doubles, and other values without having to create individual names for each function, such as AverageInts(), AverageDoubles(), and so on.

- Suppose you write a function that doubles whatever input you give it. You would like to be able to pass in an int, a long, a float, or a double. Without function overloading, you would have to create four function names:

```
int DoubleInt(int);
long DoubleLong(long);
float DoubleFloat(float);
double DoubleDouble(double);
```

- With function overloading, you make this declaration:

```
int Double(int);
long Double(long);
float Double(float);
double Double(double);
```

- This is easier to read and easier to use. You don't have to worry about which one to call; you just pass in a variable, and the right function is called automatically.

Listing 5.8. A demonstration of function polymorphism.

```
#include <iostream.h>

int Double(int);
long Double(long);
float Double(float);
double Double(double);

int main()
{
        int myInt = 6500;
        long myLong = 65000;
        float myFloat = 6.5F;
        double myDouble = 6.5e20;

        int doubledInt;
        long doubledLong;
        float doubledFloat;
        double doubledDouble;
```

```
        cout << "myInt: " << myInt << "\n";
        cout << "myLong: " << myLong << "\n";
        cout << "myFloat: " << myFloat << "\n";
        cout << "myDouble: " << myDouble << "\n";

        doubledInt = Double(myInt);
        doubledLong = Double(myLong);
        doubledFloat = Double(myFloat);
        doubledDouble = Double(myDouble);

        cout << "doubledInt: " << doubledInt << "\n";
        cout << "doubledLong: " << doubledLong << "\n";
        cout << "doubledFloat: " << doubledFloat << "\n";
        cout << "doubledDouble: " << doubledDouble << "\n";

return 0;
}
```

```
int Double(int original)
{
        cout << "In Double(int)\n";
        return 2 * original;
}

long Double(long original)
{
        cout << "In Double(long)\n";
        return 2 * original;
}

float Double(float original)
{
        cout << "In Double(float)\n";
        return 2 * original;
}

double Double(double original)
{
        cout << "In Double(double)\n";
        return 2 * original;
}
```

Output

myInt: 6500
myLong: 65000
myFloat: 6.5
myDouble: 6.5e+20
In Double(int)
In Double(long)
In Double(float)
In Double(double)
DoubledInt: 13000
DoubledLong: 130000
DoubledFloat: 13
DoubledDouble: 1.3e+21