LESSON SET

3 Expressions, Input, Output and Data Type Conversions

PURPOSE

1. To learn input and formatted output statements
2. To learn data type conversions (coercion and casting)
3. To work with constants and mathematical functions

PROCEDURE

1. Students should read the Pre-lab Reading Assignment before coming to lab.
2. Students should complete the Pre-lab Writing Assignment before coming to lab.

Contents	Pre-requisites	Approximate completion time	Page number	Check when done
Pre-lab Reading Assignment		20 min.	26	
Pre-lab Writing Assignment	Pre-lab reading	10 min.	32	
LESSON 3A				
Lab 3.1 Working with the `cin` Statement	Confidence in use of data types	15 min.	33	
Lab 3.2 Formatting Output	Basic understanding of `cout` and formatted output	15 min.	35	
Lab 3.3 Arithmetic Operations and Math Functions	Understanding of pre-defined functions `pow` and `sqrt`	20 min.	36	
LESSON 3B				
Lab 3.4 Working with Type Casting	Understanding of type casting (implicit and explicit data conversion)	20 min.	37	

continues

Lab 3.5			
Student Generated Code Assignments	Understanding of all concepts covered in this section.	30 min.	39

PRE-LAB READING ASSIGNMENT

Review of the `cout` Statement

The **`cout`** statement invokes an output stream, which is a sequence of characters to be displayed to the screen.

Example: `cout << "Hi there";`

The **insertion operator** << inserts the string of characters `Hi there` into the output stream that goes to the screen. The `cout` statement can be thought of as an **`ostream`** (output stream) data type.

Input Instructions

Just as the `cout` statement transfers data from the computer to the "outside" world, the **`cin`** statement transfers data into the computer from the keyboard.

Example: `cin >> grade;`

The **extraction operator** >> extracts an item from the input stream. In this case, since `grade` is an integer, this instruction will wait for an integer to be entered at the keyboard and then will place that number in the memory location called `grade`.

Just as `cout` is of type ostream, `cin` is considered to be an **istream** (input stream) data type. In order to use `cin` and `cout` in a C++ program, the `#include <iostream>` directive should be included in the header. The >> extraction operator also serves as a separator between input variables, allowing more than one memory location to be loaded with a single `cin` instruction. The values read must be the same data type as their corresponding variables, although a floating point variable could receive an integer since the conversion will be made automatically. Conversion is discussed later in this lesson.

Example:
```
float rate;
float hours;

cin >> rate >> hours;
```

The `cin` statement will wait for two floating point numbers (separated by at least one blank space) to be input from the keyboard. The first will be stored in `rate` and the second in `hours`.

There is one problem with the example above; it does not indicate to the user for what data the `cin` statement is waiting. Remember that the `cin` statement is expecting data from the user at the keyboard. For this reason, every `cin` statement should be preceded by a `cout` statement that indicates to the user the data to be input. Such a `cout` statement is called a **prompt**.

Example:

```
float rate, hours;
                          // More than one variable can be defined
                          // in a statement. Multiple variables are
                          // separated by a comma.
float grosspay;

cout << "Please input the pay rate per hour"
     << " and then the number of hours worked" << endl;
cin >> rate >> hours;

grosspay = rate * hours;          // finds the grosspay

cout << endl << "The rate is = " << rate << endl;
cout << "The number of hours = " << hours << endl;
cout << "The gross pay =  " << grosspay << endl;
```

When `cin` is reading numeric data, whitespace (blank spaces or unseen control characters) preceding the number are ignored and the read continues until a non-numeric character is encountered.

Strings

It is often useful to store a string, such as a name, in a variable. Since the `char` data type holds only a single character, we must define a variable that is able to hold a whole sequence of characters. One way to do this is through an **array** of characters, often referred to as a **C-string** in C++. When using this method to define a string, the programmer must indicate how many characters it can hold. The last character must be reserved for the end-of-string character `'\0'` which marks the end of the string. In Example 2 below, the variable `name` can hold up to 11 characters even though the size of the array indicates 12. The extra character is reserved for the end-of-string marker. Arrays of characters are discussed in a later chapter. For now we can define a variable to be a **string object**: Example 1 below.

Example 1 (using a string object)

```
string name;
cout << "What is your name";
cin  >> name;
cout << "Hi " << name << endl;
```

Example 2 (using a C-string)

```
char name[12]
cout << "What is your name";
cin >> name;
cout << "Hi " << name << endl;
```

Although Example 1 will work, we often do not use `cin >>` to read in strings. This is because of the way it handles **whitespace** (blank spaces, tabs, line breaks, etc.). When `cin >>` is reading numeric data, leading whitespace is ignored and the read continues until a non-numeric character is encountered. As one might expect, `cin >>` is a logical choice for reading numeric data. However, when `cin >>` is reading into a variable defined as a string, it skips leading whitespaces but stops if a blank space is encountered within the string. Names that have a space in it such

as Mary Lou, would not be read into one variable using `cin >>`. We can get around this restriction by using special functions for reading whole lines of input. The `getline` function allows us to input characters into a string object. In Example 1 above we could read a name like Mary Lou into the `name` variable with the statement

```
getline(cin, name);
```

The first word in the parentheses is an indication of "where" the data is coming from. In this case it is coming from the keyboard so we use `cin`. Data could come from other sources such as files (discussed later in this chapter) in which case the name of the file would be used instead of `cin`. The second word in parentheses is the name of the variable that will "receive" the string (`name` in this case).

When using C-strings, we can read whole lines of input using `cin.getline (string_name, length)`, where length specifies the number of characters the C-string can hold. In Example 2 above, we could read a name like Mary Lou into the `name` variable with the statement

```
cin.getline(name,12);
```

This allows a maximum of 11 characters to be read in and stored in `name`, reserving a space for the '\0' end-of-string character.

Summary of storing and inputting strings

`cin >> name;`	Skips leading whitespaces Stops at the first trailing whitespace which is not consumed (ie. the whitespace is not placed in name)
`cin.getline(name,12);`	Does not skip leading whitespaces Stops when either 11 characters are read or when an end-of-line '\n' character is encountered (which is not consumed)

Formatted Output

C++ has special instructions that allow the user to control output attributes such as spacing, decimal point precision, data formatting and other features.

Example:

```
cout << fixed                 // This displays the output in decimal
                              // format rather than scientific notation.

cout << showpoint;            // This forces all floating-point output to
                              // show a decimal point, even if the values
                              // are whole numbers

cout << setprecision(2);      // This rounds all floating-point numbers
                              // to 2 decimal places
```

The order in which these **stream manipulators** appear does not matter. In fact, the above statements could have been written as one instruction:

```
cout << setprecision(2) << fixed << showpoint;
```

Spacing is handled by an indication of the width of the field that the number, character, or string is to be placed. It can be done with the `cout.width(n);` where `n` is the width size. However it is more commonly done by the `setw(n)` within a `cout` statement. The `#include <iomanip>` directive must be included in the header (global section) for features such as `setprecision()` and `setw()`.

Example:
```
float price = 9.5;
float rate = 8.76;
cout << setw(10) << price << setw(7) << rate;
```

The above statements will print the following:

```
    9.5   8.76
```

There are seven blank spaces before 9.5 and three blank spaces between the numbers. The numbers are right justified. The computer memory stores this as follows:

							9	.	5				8	.	7	6

Note: So far we have used `endl` for a new line of output. `'\n'` is an escape sequence which can be used as a character in a string to accomplish the same thing.

Example: Both of the following will do the same thing.

```
cout << "Hi there\n";            cout << "Hi there" << endl;
```

Expressions

Recall from Lesson Set 2 that the assignment statement consists of two parts: a variable on the left and an expression on the right. The expression is converted to one value that is placed in the memory location corresponding to the variable on the left. These expressions can consist of variables, constants and literals combined with various operators. It is important to remember the mathematical precedence rules which are applied when solving these expressions.

Precedence Rules of Arithmetic Operators

1. Anything grouped in parentheses is top priority
2. Unary negation (example: –8)
3. Multiplication, Division and Modulus * / %
4. Addition and Subtraction + –

Example:
```
( 8 * 4/2 + 9 - 4/2 + 6 * (4+3) )
( 8 * 4/2 + 9 - 4/2 + 6 * 7 )
( 32 /2   + 9 - 4/2 + 6 * 7 )
(   16    + 9 - 4/2 + 6 * 7 )
(   16    + 9 -  2  + 6 * 7 )
(   16    + 9 -  2  + 42 )
(      25     -  2  + 42 )
(           23      + 42 )  =    65
```

Converting Algebraic Expressions to C++

One of the challenges of learning a new computer language is the task of changing algebraic expressions to their equivalent computer instructions.

Example: `4y(3-2)y+7`

How would this algebraic expression be implemented in C++?

```
4 * y * (3-2) * y + 7
```

Other expressions are a bit more challenging. Consider the quadratic formula:

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

We need to know how C++ handles functions such as the square root and squaring functions.

There are several predefined math library routines that are contained in the `cmath` library. In order to use these we must have the `#include <cmath>` directive in the header.

Exponents in C++ are handled by the `pow(number,exp)` function, where `number` indicates the base and `exp` is the exponent. For example,

2^3 would be written as `pow(2,3)`

5^9 would be written as `pow(5,9)`

Square roots are handled by `sqrt(n)`. For example,

$\sqrt{9}$ would be written as `sqrt(9)`

Look at the following C++ statements and try to determine what they are doing.

```
formula1 = ( -b + sqrt(pow(b,2) -(4 * a * c))) / (2 * a);
formula2 = ( -b - sqrt(pow(b,2) -(4 * a * c))) / (2 * a);
```

(These are the roots from the quadratic formula in C++ format.)

Data Type Conversions

Recall the discussion of data types from Lesson Set 2. Whenever an integer and a floating point variable or constant are mixed in an operation, the integer is changed temporarily to its equivalent floating point. This automatic conversion is called **implicit type coercion**.

Consider the following:

```
int count;
count = 7.8;
```

We are trying to put a floating point number into an integer memory location. This is like trying to stuff a package into a mailbox that is only large enough to contain letters. Something has to give. In C++ the floating point is **truncated** (the entire fractional component is cut off) and, thus, we have loss of information.

Type conversions can be made explicit (by the programmer) by using the following general format: static_cast<DataType>(Value). This is called **type casting** or **type conversion**.

Example:

```
int count;
float sum;

count = 10.89;                    // Float to integer  This is Type coercion
                                  // 10 is stored in count

count = static_cast<int>(10.89);  // Also float to integer; however this is
                                  // type casting
```

If two integers are divided, the result is an integer that is truncated. This can create unexpected results.

Example:

```
int num_As = 10;
int totalgrade = 50;
float percent_As;

percent_As = num_As / totalgrade;
```

In this problem we would expect `percent_As` to be .20 since 10/50 is .20. However since both `num_As` and `totalgrade` are integers, the result is integer division which gives a truncated number. In this case it is 0. Whenever a smaller integer value is divided by a larger integer value the result will always be 0. We can correct this problem by type casting.

```
percent_As = static_cast<float>(num_As)/totalgrade;
```

Although the variable `num_As` itself remains an integer, the type cast causes the divide operation to use a copy of the `num_As` value which has been converted to a float. A float is thus being divided by the integer `totalGrade` and the result (through type coercion) will be a floating-point number.

PRE-LAB WRITING ASSIGNMENT

Fill-in-the-Blank Questions

1. What is the final value (in C++) of the following expression?

   `(5 - 16 / 2 * 3 + (3 + 2 / 2) - 5)` _____

2. How would the following expression be written in C++?

   $2x + 3^4$

   _____

3. Implicit conversion is also known as data type _____.
4. Explicit type conversion is also known as type _____.
5. List the preprocessor directive that must be included for `cin` and `cout` to be used in a C++ program. _____
6. Blank spaces or unseen control characters in a data file are referred to as _____.
7. The << in a `cout` statement is called the _____operator.
8. The `#include<`_____> is needed for formatted output.
9. The `'\n'` is a special character that _____.