CIS 278 (CS1) Programming Methods: C++

Lesson 2: More C++ Basics

Skip to Main Content

Section 2.1: Expressions and Operators

Let's take a closer look at assignment statements. The general form of an assignment statement is

```
<variable> = <expression>;
```

There are 4 elements here. First is a variable, then the assignment operator, then an expression, then a semi-colon. We have covered three of these 4 elements in sufficient detail. But we have not yet said anything about expressions. What exactly is an expression?

We will limit our discussion of expressions at this point to arithmetic expressions (expressions with operands that are numerical). Arithmetic expressions in C++ have essentially the same definition as they do in algebra. However, for the sake of review, here is a definition of "expression":

An **expression** can be one of three things:

- *a value.* For example, 47 is a valid expression, and so

  ```
  aVariable = 47;
  ```

  is a valid assignment statement.
- *a variable.* For example, `paycheck` is a valid expression, and so

  ```
  aVariable = paycheck;
  ```

  is a valid assignment statement.
- *two expressions connected by an operator* and optionally surrounded by parentheses. For example, `hoursWorked * 12` is a valid expression because it is two expressions connected by an operator (`hoursWorked` is an expression, `12` is an expression, and `*` is an operator). So,

  ```
  aVariable = hoursWorked * 12;
  ```

  is a valid assignment statement.

By using this last rule we can make arithmetic expressions as long and complex as we want. For example,

```
aVariable = (hours * hours) / (14 + paycheck - 3) / 17;
```

is a valid assignment statement, because that mess on the right side of the assignment operator (=) is a valid arithmetic expression.

In order to build up arithmetic expressions we need to have arithmetic operators with which to connect our arithmetic expressions. C++ provides us with the usual arithmetic operators, plus one that might be new to you. Here's what you need to know:

- Addition and subtraction work just like they do in arithmetic.
- Multiplication works just like it does in arithmetic, except that you use an asterisk (`*`) instead of "X" or "•". Notice that you cannot indicate multiplication by just putting the two operands right next to each other like you can in algebra. For example, `4x` means 4 times x in algebra. In C++ we would have to say `4 * x`.

- Division is indicated by a slash (/). You can't use ÷. Also, expressions in C++ must come all on one line, so you can't use a horizontal bar to indicate division.

- The modulus operator (this is the one that might be new to you) is a way to get the remainder when you divide. For example, 7 divided by 3 is 2 with a remainder of 1. So, `7 % 3` is equal to 1. Don't worry, we'll talk about modulus in more detail later on.

The following table summarizes the above comments:

operation	**C++ operator**	**algebraic example**	**C++ example**
addition	`+`	`x + 7`	`x + 7`
subtraction	—	`x — 7`	`x — 7`
multiplication	`*`	`4x`	`4 * x`
division	`/`	`x÷2`	`x / 2`
modulus	`%`	none	`x % 2`

Section 2.2: Precedence

When our expressions begin to get more complex we need to have rules that tell us which operators in our expression we should evaluate first. Since C++ follows the same rules of precedence for arithmetic operators as we follow in arithmetic, this section will be review for many of you. It is included, however, because understanding precedence becomes increasingly important as we continue learning C++. As an example, consider this expression:

```
12 + 9 / 3.
```

When we evaluate this expression do we get 7? Or is it 15? It depends on which operators we evaluate first. If we evaluate the + first, then we have 12+9 = 21, then we divide 21 by 3 to get 7, like this:

12 + 9 / 3	
21/ 3	(do addition first, replace 12 + 9 with 21)
7	(replace 21 / 3 with 7)

The rule, however, is that division comes before addition. So we do 9 / 3 = 3 first, then we add 12 and 3 to get 15, like this:

12 + **9 / 3**	
12 + **3**	(do division first, replace 9/3 with 3)
15	(replace 12 + 3 with 15)

We say that the operators that are evaluated first have a higher **precedence** than those that are evaluated later. So, for example, we would say that multiplication has a higher precedence than addition. The rules are:

1. Any operations that are inside of parentheses come first.
2. Then come multiplication, division, and modulus, done from left to right. Notice that, contrary to popular belief, multiplication **does not** come before division.
3. Addition and subtraction, done from left to right, come last.

Example 1:
Problem: evaluate the expression `24 / 3 * 4.`
Solution: if multiplication came before division, our answer would be 2:

24 / **3 * 4**	
24 / **12**	(replace 3 * 4 with 12)

2	(replace 24 / 12 with 2)

Rule #2, however, tells us that division and multiplication come at the same time. We simply evaluate operators from left to right. Since the division is left of the multiplication in this example, the division should be done first. Here is the correct solution:

24 / 3 * 4	
8 * 4	(replace 24 / 3 with 8)
32	(replace 8 * 4 with 32)

Example 2:
Problem: evaluate the expression `24 / (3 * 4)`.
Solution: Because Rule #1 tells us to do operations that appear inside parentheses first, we are now forced to start with 3 * 4:

24 / (**3 * 4**)	
24 / (**12**)	(replace 3 * 4 with 12)
2	(replace 24 / (12) with 2)

Example 3:
Problem: evaluate the expression `24 - 3 * 4`.
Solution: since multiplication comes before subtraction, we must start with 3* 4:

24 - **3 * 4**	
24 - **12**	(replace 3 * 4 with 12)
12	(replace 24 - 12 with 12)

Example 4:
Problem: evaluate the expression `(24 - 3) * 4`.
Solution:

(**24 - 3**) * 4	
(**21**) * 4	(replace 24 - 3 with 21)
84	(replace 21 * 4 with 84)

Section 2.3: Modulus and Division

You may have noticed that everything we have said so far in the lessons has applied strictly to integer values. We will discuss non-integer values in the next section of this lesson. We have a little problem, though. If we are going to limit ourselves to just integers, what happens when we do division and it doesn't come out to be an integer? For example, what if we were to include a sequence of statements like this in our program:

```
int hours;
hours = 23 / 4;
```

The answer is that in C++ when you divide two integers, the answer is an integer. You simply chop off (truncate is the technical term) any fraction or decimal part of the number. So 23 / 4 is 5. This is extremely counter-intuitive, so make sure you take note of this! It will come up in your assignments...you'll get the wrong answer, and if you don't remember this, you won't understand why.

C++ provides the modulus operator so that we can get the remainder in integer division. To understand how this works, let's go back to third grade math for a minute (at least it was the third grade at my school). Remember when we first learned how to divide? We were taught to do the problem something like this:

```
    5 r 3
4 | 23
```

$$\frac{20}{3}$$

We could read this as "23 divided by 4 is equal to 5 with a remainder of 3." This is how C++ does division when we are working with just integers. The "is equal to 5" means that 23 / 4 equals 5. The "with a remainder of 3" means that 23 % 4 equals 3.

Although it might be hard to imagine at this point, the modulus operator actually turns out to be quite useful. For example, you can use the modulus operator to see if a one number is divisible by another. To see if 24 is divisible by x, you would evaluate 24 % x. If the answer is 0, then 24 is divisible by x (in other words, when you divide 24 by x the remainder is 0).

Here is another example. Let's say we wanted to write a program to figure out how to make change for 83 cents using quarters, dimes, nickels and pennies. Our first step would be to figure out how many quarters to use. A correct assignment statement to accomplish this would be

```
numQuarters = 83 / 25;
```

83 / 25 is 3, so this gives us the correct number of quarters. Next we need to figure out how many dimes. Before we can do that we need to figure out how much money is left after we have taken care of the quarters. This is where modulus comes in.

```
amountLeft = 83 % 25;
```

83 % 25 is 8, so this gives us the correct amount of money left. The rest of this example is left as an exercise.

Section 2.4: Double Variables and Values

Up to this point we have discussed only integer values and variables. Of course, not every problem we encounter will deal exclusively with integers. In the rest of this lesson we will discuss several other types of variables that we can use. Throughout the discussion it is important to understand the distinction between variables and values. (Warning: our text uses the phrase "literal constant" or "literal value" or just "literal" where I use the word "value".) For each data type, you must know how to distinguish between a variable and a value of that type, and under what circumstances each is used.

In lesson 1 we used variables of type `int` to store integer values. For example, the statement

```
hours = 47;
```

assigns the integer **value** `47` to the integer **variable** `hours`. In the same way we will now use **variables** of type `double` to store `double` **values**. A `double` value is defined as a number that has a decimal point in it. (Note: in some textbooks the type "float" is used instead of "double". You should use "double", not "float". When you see "float" used in the text just think "double". For our purposes they are synonymous.) In mathematical terms we might think of these as real numbers, but the C++ definition of "double value" is not quite the same as the mathematical definition of "real number." For example, in C++ 5 is an integer value, but 5.0 is a double value because it includes a decimal point. This is despite the fact that in algebra 5.0 would be considered an integer.

The declaration statement used to define a `double` variable looks like this:

```
double salary;
```

After using this declaration statement, we could then say something like

```
salary = 1213.81;
```

Here's another example:

```
double gallons;
double quarts;

quarts = 7;
gallons = quarts / 4;
```

If we had declared gallons and quarts to be integers, gallons would have been assigned the value 1. However, since we have declared gallons and quarts to be doubles, this statement assigns to gallons the value 1.75.

Because of the way that computers store doubles, you can never assume that a double value is being stored with exact precision. For this reason, you should never use the equals operator (==) to compare doubles.

You should always use type int whenever possible. Use double only when the situation requires the use of non-integer values.

At times you will need to convert an int into a double. For example, in order to do some sort of calculation that requires decimal points in the answer. If the int that needs to be converted is a literal, this is easy: you can just add a ".0" to the literal. For example, if the int is 47, to force C++ to treat it as a double instead of as an int you can just use 47.0. You should expect to have to use this technique in assignment 2.

If the int that needs to be converted is an int variable, you should use a static-cast. (There are other ways to do this conversion in C++, but the one I'm about to show you is best practice.) If you have an int variable named "numCorrect" and you need to convert it to a double, use this syntax: static_cast <double> (numCorrect). You probably won't need this technique right away, but keep it in mind for future assignments.

All 5 of the arithmetic operators we have discussed require both of their operands to be of the same type. This means, for example, that C++ cannot add an int and a double. However, C++ is able convert back and forth between the two types, so the two types can be mixed in expressions and assignment statements. The assignment statement above, for example, has a double (quarts) and an int (4) in the expression. The results, however, are not always what we would expect. Unlike the situation in algebra, in C++ we cannot perform mathematical operations with operands of two different types. We must have rules for how to convert one type to another so that the types of the operands match.

Rule #1 (expressions): The first rule is that if an expression has one int operand and one double operand, the int operand is temporarily changed to a double, so the result of the expression will be of type double. For example let's look at the expression

```
(2 + 3.5)/5
```

First we do 2 + 3.5. Since 2 is an int value and 3.5 is a double value, 2 becomes 2.0 (a double value), and the result is 5.5.

Now we have 5.5 / 5. Since 5.5 is a double value and 5 is an int value, the 5 gets changed to 5.0 (a double value) and the result is 1.1.

Rule #2 (assignments): Things get a bit more complicated in assignment statements. The rule is that C++ automatically changes the value of the expression on the right side of the assignment operator so that it matches the type of the variable of the left side of the assignment operator.

Example 1:

```
int i;
double x;
double y;

x = 11;
y = 4;
i = x / y;
```

x/y is equal to 2.75. When C++ goes to assign this value to i, it realizes that i is an integer variable, and so before doing the assignment, it converts 2.75 to an integer by dropping the decimal. So i gets the value 2.

Example 2:

```
int i;
int j;
double x;

i = 11;
```

```
j = 4;
x = i / j;
```

Because `i` and `j` are both integers, `i/j` is equal to `2`. When C++ goes to assign this value to `x`, it realizes that `x` is a `double` variable, and so before doing the assignment, it converts `2` to a `double`. So `x` gets the value `2.0`.

Both of these examples illustrate that C++ doesn't always do things the way we expect it to. Unless we are careful, we might not expect `i` to get the value `2` in example 1, because in math we learn to give an exact answer (2.75) or to round off to the nearest integer (3). In example 2, we might expect `x` to get the value `2.75`. After all, `x` is a `double` variable so it could handle the value `2.75`. If we are careful, though, we notice that `i` and `j` are both integers, so `i/j` is equal to `2`. When mixing `double`s and `int`s in expressions and assignment statements, we must be very careful to follow C++'s rules exactly.

Section 2.5: Char Variables and Values

Sometimes it is desirable to store a character in a variable. In the same way that we use int **variables** to store integer **values** and double **variables** to store double **values**, we use char **variables** to store character **values**.

Characters are things like letters ('a', 'Z'), digits ('1','9','0'), and other symbols that might appear on a screen ('$','+','^'). There are also some characters that are invisible, but we won't concern ourselves with them just yet.

A declaration statement to declare a character variable looks like this:

```
char ch;
```

Then we could say something like this:

```
ch = '?';
```

We must be very careful to distinguish between char **values** and **variables**. With integers this is easy: `hours` is obviously a variable, not an integer value. `47` is obviously an integer value, not a variable. The same is true for double values and variables. With characters things get more difficult. We distinguish between variables and character values by placing single quotes around character values. This is why the question mark in the example above has single quotes around it. This means that if we see the letter `x` in a program (no quotation marks), it is a variable, not a character value. But if we see `'x'` in a program it is a character value, not a variable.

A common mistake made by novice computer programmers is omitting the quotation marks when using a character value. When this happens, C++ will think that we are trying to use a variable that has not been declared.

Two other important points about character values and variables:

- When a character value is typed in by the user, the user should **not** enclose it in quotes.
- Character variables store exactly one character. It doesn't make sense to say `ch = 'hi';`

Be careful as you work through the next three examples. Note that despite their many similarities there are important differences. You might even want to start by making sure that you see the differences in the code given for the three examples.

Example 1
What would be the output of the following program segment?

```
char m;
char x;
m = 'x';
x = 'm';
cout << 'x';
```

The answer is that the letter "x" would appear on the output screen. Why? Let's trace through this code. First two variables are declared. We picture that situation like this:

The question marks in the boxes indicate that the values stored in the variables m and x are unknown. This is always the case immediately after we declare a variable. The value could be anything, we just don't know.

Next we have an assignment statement placing the character value 'x' into the variable m. Our picture now looks like this:

The next statement is `x = 'm';` Our picture now looks like this:

It turns out that all of this is irrelevant, because when we hit the cout statment, the character value 'x' is printed on the screen without regard to what is stored in the variables.

Example 2
What would be the output of the following program segment?

```
char m;
char x;
m = 'x';
x = 'm';
cout << x;
```

The answer is that the letter "m" would appear on the output screen. Why? Let's trace through this code. First two variables are declared. We picture that situation like this:

Next we have an assignment statement placing the character value 'x' into the variable m. Our picture now looks like this:

The next statement is `x = 'm';` Our picture now looks like this:

When we hit the cout statement, we look up the value stored in the variable "x". This is where example 2 differs from example 1. In example 1, the x in the cout statement had single quotes around it, making it a value that was simply printed. In example 2 the "x" does not have single quotes around it, so it is the **variable** x. The character value 'm' is stored in the variable x, and that is what gets printed on the screen.

Example 3 What would be the output of the following program segment?

```
char m;
char x;
m = 'x';
x = m;
cout << x;
```

The answer is that the letter "x" would appear on the output screen. Why? Let's trace through this code. First two variables are declared. We picture that situation like this:

The question marks in the boxes indicate that the values stored in the variables m and x are unknown. This is always the case immediately after we declare a variable. The value could be anything, we just don't know.

Next we have an assignment statement placing the character value 'x' into the variable m. Our picture now looks like this:

The next statement is `x = m;` Notice that the thing on the right of the assignment operator is not a character value this time. It is a variable. We can tell this because if it was a character value it would be surrounded by single quotes. Execution of this assignment statement involves looking up the value stored in the variable m and placing that value in the variable x. As we can see from the previous picture, the value stored in the variable m is the character value 'x'. So we place that character into the variable x. our picture now looks like this:

So when we hit the cout statment, the character value 'x' is stored in the variable x, and that is what gets printed on the screen.

Section 2.6: String Variables and Values

Well, you ought to be getting the picture by now. In the same way that we use int **variables** to store integer **values** and double **variables** to store double **values** and char **variables** to store character **values**, we use **string** variables to store **string** values. String values are sequences of characters. When they appear in a C++ program, they come enclosed in double quotes (just like characters come enclosed in single quotes).

You have been using string values since section 1 of lesson 1. String values are enclosed in double quotes. You haven't used them in assignment statements yet because I haven't told you about string variables yet.

Unlike other types in C++, when you use string variables you must first #include <string> at the top of your file.

Here is some code that declares and uses a string variable:

```
string str1;
str1 = "my dear aunt sally";
cout << str1;
```

This code causes the string "my dear aunt sally" to be printed on the screen.

To be a bit more precise, string values consist of a sequence of 0 or more characters. String values are a bit more complicated than other values. Here are a couple of rules you need to know about string values:

In a C++ program a string value must appear all on one line.

The string that consists of 0 characters has a special name. It is called the "null" string or "empty" string. In a C++ program it would appear like this:

```
""
```

Notice there is nothing between the quotes, not even a space. Also notice that while you can have a null string value, you can't have a null character value, because all character values must contain exactly one character.

Like other types of variables, you can use the extraction operator to read a string. When you do, C++ uses whitespace to mark off where strings start and end. So, for example, if you have this code:

```
cin >> str1;
```

and the user enters "My name is Dave", str1 will get the string value "My".

You can add two strings together, like this:

```
string str1, str2, str3;
str1 = "hello ";
str2 = "there!";
str3 = str1 + str2;
cout << str3;
```

This code would print the sentence "hello there!" on the screen. When used with string operands, the addition (+) operator returns the string that is created by appending the right operand onto the end of the first operand (but neither operand is actually changed in the process). This process is called "concatenation" of strings.

Using the + operator has one limitation that can be kind of tricky, so be careful. The rule is that at least one of the operands must be a string variable. The other operand could be a string value or a string variable or even a character value or character variable, but at least one of the operands must be a string variable.

Unlike the other types we have talked about, with string variables you can call functions to perform operations on them. You do this by putting a dot (".") and then the function you want to call after the dot. For example, I can find the number of characters in a string like this:

```
string str1;
str1 = "monkey see monkey do";
cout << "The string has " << str1.length() << " characters." << endl;
```

This will print the number "20" on the screen because there are 20 characters in that string. Don't forget that spaces count as characters too!

Another example is the substr() function. When you call the substr() function, you provide two arguments. Here's an example:

```
string str1;
string str2;
str1 = "monkey see monkey do";
cout << "The string has " << str1.length() << " characters." << endl;
str2 = str1.substr(3, 7);
cout << "The new string is " << str2 << "." << endl;
```

In this example, the substr() function returns the string formed by starting at the character in position 3 of str1 and including a total of 7 characters. When counting positions in the string, we start with 0, not 1. So str2 is set to "key see". Be careful: many students assume that the second argument indicates the position of the last character to be included in the result. It's not the position of the last character to be included, it's the number of characters to be included.

Section 2.7: Programming With Style

In this class you will not merely be learning how to write computer programs that work. Just as importantly, you will be learning to write good computer programs that work. Students who are just beginning to learn the art of computer programming are often taken aback, confused, and even frustrated by this fact. "Why didn't my program get a perfect grade!?" they ask. "It worked perfectly!" In the real world of computer programming, a program is seldom written and forgotten. It is revisited often to improve it, keep it up-to-date with current circumstances, or reuse parts of it in a new program. In addition, several people may be working on the same program. For these reasons it is important to write computer programs that are clear, easy to read, and well organized.

As we strive to write good computer programs, we have four primary goals in mind. We want to make it easy to:

1. read and understand our program,
2. modify our program,
3. debug our program,
4. reuse parts of our program.

Here are some tips on how to achieve these goals. As we move on with this course considerations of style will become more and more relevant. Get in the habit of paying close attention to them now.

Good Names
Using a variable name that describes accurately and precisely the role that the variable plays in your program is an excellent strategy for making your program easy to understand. Don't use an abbreviation that someone not familiar with your program will have to think about. Never use just one letter for a variable name when you could be more precise.

Blank Lines
Using blank lines in your program to separate the different parts of your program makes it easy for someone to quickly locate these different parts. Don't double space your whole program -- this would defeat the purpose of using blank lines. They aren't just to spread things out, but rather to delineate different parts of your program.

Indentation
You may have noticed in the examples given in these notes that all of the statements in our programs are indented. This is not an accident. It is done on purpose to make our program more readable. You should do the same. As we move on, correct indentation will play a more and more important role.

Comments
C++ gives you the ability to put comments in your program to help out someone who is trying to understand your program. You should use this ability liberally. As your programs become more complex we will expand on your use of this tool. For now you should put a comment at the top of each program that you write, explaining in a few sentences what the program does. You should also include your name, the date, and any other relevant information. Please read the Style Conventions section of our course syllabus for more information about commenting your programs.

There are two ways to make a comment in a C++ program. If the comment appears on a single line, you can simply precede it with two slash characters: //. If the comment goes onto more than one line, you must begin it with /* and end it with */. The following example illustrates both of these uses. Because the purpose here is only to illustrate the syntax of comments, the comments given here are minimal.

```
/*
Steve Old
    12/12/94
    Exercise 1.5 Problem 1

    This program asks the user for the pay rate
    and the number of hours worked and calculates
    the amount of the paycheck.
```

```
*/

    #include <iostream>
    using namespace std;

    int main()
    {
        int hours;              // the hours worked
        int payRate;            // the rate of pay
        int paycheck;           // the amount of the paycheck

        cout << "enter hours worked: ";
        cin >> hours;
        cout << "enter rate of pay: ";
        cin >> payRate;
        paycheck = hours * payRate;
        cout << "the amount of the paycheck is "
             << paycheck << " dollars." << endl;
    }
```

Notes about comments:

- If you start a comment with /* and forget to end it with */ C++ might think that your whole program is one big comment! Very strange things can happen, so carefully end each of your comments.
- Your comments should be written so that they would be helpful to an expert computer programmer. Here's an example of a bad comment:

  ```
  tax = 0.22 * salary; /* assign .22 * salary to the variable tax */
  ```

  As computer programmers, we already know what the statement does. There is no need to repeat it.
- It is possible to place comments in such a way that they will actually make your program more difficult to read. Be careful that your comments do not clutter up your code. It should normally not be necessary to place comments in the body of a program unless there is a special need to explain a particular detail. If you find yourself needing to explain your code, perhaps you should think of a clearer way of writing it. Some programmers prefer to place comments in the body of the program, so if you choose to do this I won't penalize you. However, if you do this, you must take great care to make sure that the comments don't interfere with our program's readability. One exception to this general rule is that it is good practice to place a comment near variable declarations that need explanation, as illustrated in the example above.

Section 2.8: Named Constants

It is usually best not to use int or double literals in our code. Instead we define named constants at the top of the file, and use those constants in the place of the actual literals. As an example, let's take a look at one version of our paycheck program from lesson 1:

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

int main()
{
    int hoursWorked;
    int paycheckAmount;

    cout << "Enter hours worked: ";
    cin >> hoursWorked;
    paycheckAmount = hoursWorked * 12;
    cout << "The amount of the paycheck is $"
         << paycheckAmount << endl;
}
```

```
Enter hours worked: 23
The amount of the paycheck is $276
```

In this example, we are using the int literal 12 directly in the code. But the 12 represents the hourly payrate. So it would be better to define a named constant instead:

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

    const int HOURLY_PAYRATE = 12;

int main()
{
    int hoursWorked;
    int paycheckAmount;

    cout << "Enter hours worked: ";
    cin >> hoursWorked;
    paycheckAmount = hoursWorked * HOURLY_PAYRATE;
    cout << "The amount of the paycheck is $"
         << paycheckAmount << endl;
}
```

```
Enter hours worked: 23
The amount of the paycheck is $276
```

Why Use Named Constants? There are two reasons why it is good practice to use named constants.

1. **Modifying our code is easier** Suppose we have a 10,000,000 line program and the payrate of 12 occurs 4,000 times in the program. Now suppose we need to change the payrate from $12 to $13. It's going to take us a long time to find every occurrence of the number 12 and change it to 13! And there's a pretty decent chance that we will miss one. But if we have defined a constant to represent the payrate, then we simply change the 12 at the top of the program to 13 and we are done! (Note that we can't simply use "find and replace all" because some of the 12's in our program might represent other things that need to remain 12.)

2. **Self-Documenting Code** If someone is trying to understand our code and sees the number 12, it may not be obvious what the role of the number 12 is in the code. But if someone sees "HOURLY_PAYRATE" in the code, it's clear what it represents.

Notice that I used all uppercase for the constant. This is a very strong convention in C++, and you should follow it. Use all uppercase for constants, and use the underscore to separate the words if there are multiple words.

When you are first learning programming it can be hard to figure out when is a good time to use a named constant and when it is better to simply use the literal. For the first few assignments I try to tell you explicitly whether you should use constants or not. The rule of thumb is that if the number represents something to which we can give a clear name (such as HOURLY_PAYRATE), then use a named constant. An important example of where constants aren't useful is in formulas where the number doesn't represent anything in particular. For example, the formula to convert from celsius to fahrenheit is F = (9/5)C + 32. The 9/5 and the 32 don't represent anything in particular; they are just numbers that happen to make the formula work. Don't use named constants in this case.

In particular, The numbers 0 and 1 typically don't represent anything in particular, so usually you won't need to create a constant for these two numbers.

Sometimes students use a named constant but include the number in the constant's name. For example,

```
const int SEVEN = 7;
```

This does absolutely no good. It doesn't make it easier for us to modify our code -- if we need to change the number to, say, 13, then we would have SEVEN = 13, which would clearly be confusing. It also doesn't help us understand the role of the number in the code. In this case we should either think of a name for the constant that represents that number's role in the program, or decide that it doesn't represent anything in particular and just use the literal.

Although there are exceptions to this, in general (and always for our class) constants should be declared at the top of the file just above int main(). (This is the opposite of variables, which should be declared INSIDE int main().) This makes them "global", which means that they can be accessed from anywhere in the file. That way it is easy to find them when we need them, and we can use the same constant name throughout our code. Don't worry if this talk about "global" is a little hazy right now. We'll study it more later. What you need to know is that named constants go above int main() and variables go inside int main().

Section 2.9: More About the Stream Extraction Operator

Let me say before you begin reading the rest of lesson 2 that I don't think it's very well written. I think I can make it simpler and easier to understand. I almost deleted it, but thought I would leave it here just in case it helps someone.

So far in lecture we have only used the extraction operator in a very simple way: we put the word cin, followed by the stream extraction operator, followed by a single variable name, followed by a semi-colon. Each cin in our programs has corresponded to the user typing a single value and hitting the enter or return button. We will now discuss some ways in which the stream extraction operator is actually much more complex (and powerful) than this.

First we should note that the stream extraction operator can be used to read in more than one value. For example, the statements

```
cin >> hours;
cin >> payrate;
```

could be combined into the statement

```
cin >> hours >> payrate;
```

Normally we would not want to do this because it is better programming practice to prompt the user separately for each piece of information she is entering. However, there will be a few situations in which we will need to do something like this, particularly when we are getting our input from a file instead of from the user.

Notice that while the items in a cout statement can be any expression (for example a string value, endl, or a complex integer expression), the items in a cin statement must be variables.

Streams
First we need to understand the concept of a stream. I implied previously that when you use the stream extraction operator data goes directly from the keyboard to a variable. The situation is actually a bit more complex. When the user types in some data and then hits the return button, what actually happens is the data is placed into the input stream. Here is a more detailed description of what the extraction operator does: First, any data that is sitting in the input stream is extracted from the stream and placed into the variable. If there is no data in the input stream (as is typically - but not always - the case), the computer will stop and allow the user to enter more data. When the user hits the return button, the data goes into the stream and is then processed by the extraction operator.

This probably won't seem real clear the first time you read it. Let me do an example, and then you should come back and read the previous paragraph again to see if it is clear. But first, there are three other pieces of preliminary information:

1. When the user hits the return button, two things happen. First, as I stated above, hitting the return button causes the data that the user just typed to be placed into the input stream. Secondly, it places a special character called the newline character into the input stream. We will speak more of this character later. For now just understand that it exists and that in a C++ program it can be represented with a backslash followed by the letter "n" (\n). So, for example, if ch is a char variable you could store the newline character into ch like this: ch = '\n'.
2. spaces, newlines, and tabs are grouped into a category called whitespace. Don't worry too much about tabs yet. For now, when you think whitespace think spaces and newlines.

3. The stream extraction operator ignores whitespace. It only considers non-whitespace characters when taking values from the input stream and placing them into variables. Whitespace characters are used only to separate the values in the input stream.

Example 1:

Suppose the statement

```
cin >> hours >> payrate;
```

is to be executed (where hours and payrate are int variables), and the user types

```
~~~78~~~~49~~~\n
```

(where "~" represents a space). Remember that the \n means that the user typed the return button at that point. The stream extraction operator would begin by looking for an int in the input stream (since "hours" is an int variable). It would skip over the three leading spaces, and then start reading an int value at the first non-whitespace character. In this case that would be the "7". It would continue reading characters until it reached a character that could not possibly be part of an int value. In this case that would be the next space. So the value 78 would be placed into the variable hours. Similarly, the next extraction operator would begin by skipping over the four spaces in the middle of our input and then placing the value 49 into the variable payrate.

Notice that at this point there are still 4 characters remaining in the input stream. This is not a problem, since next time the stream extraction operator is used, it will begin by simply skipping over these 4 characters.

Notice that this example would work exactly the same way if the statement had been divided into two separate cin statements, like this:

```
cin >> hours;
cin >> payrate;
```

In addition, the example would work exactly the same way if the user had typed the first value, then typed the return button, and then typed the second value. In that case, the input stream might look something like this:

```
~~~78~~\n~49~~~\n
```

This would not affect our example at all, since the \n is whitespace and the stream extraction operator would simply skip over it.

Section 2.10: Extracting Different Types of Data

The stream extraction operator has different ways of knowing when to stop extracting a value from the input stream, depending on the data-type of the variable in which the value will be placed. In the example above, the data-type of the variable was int. The stream extraction operator stopped extracting the value when it reached a character that could not possibly have been part of an int value - in this case, a space. It would also have stopped if it had reached any non-digit character, such as a letter or a period or a punctuation mark.

This rule also applies to the double data-type: The stream extraction operator stops extracting data when it reaches something that could not possibly be part of a double value. Unlike the situation with int, the stream extraction operator would not stop if it reached a period, because a period is a valid part of a double value.

The rule for character values is easier: the stream extraction operator simply extracts a single character. That's it. Be careful to remember, however, that the stream extraction operator will still, as always, ignore whitespace when extracting character values. As a result, the stream extraction operator can never be used to place a space or a newline character into a variable.

The rule for string values is that the stream extraction operator keeps reading characters and including them as part of the string until whitespace is reached. A result of this is that the stream extraction operator can

never be used to read a string that includes whitespace.

Example 2:

Suppose the user types

```
\n\n~~~13.72W~~\n~\n
```

Recall that each time you see a \n in the input stream it means that the user hit the return button at that point.

Suppose that i is an int variable, ch a char variable, and x a double variable. What would be assigned to i, x, and ch if the following statement were executed?

```
cin >> x >> ch >> i;
```

First the stream extraction operator would skip over all of the leading whitespace. It would then extract the double value 13.72 to place into the variable x. It would stop reading the value for x when it reached the character "w" because "w" cannot be part of a double value. It would then read the single character "w" and place it into the variable ch. It would then begin looking for an int variable to place into the variable i. It would skip over all of the whitespace which appears after the "w", but would not find an int value for i. So it would the stop and wait for the user to enter more data into the input stream.

In summary: x = 13.72, ch = "w", i remains unchanged.

Now suppose that we have the same input and the same variables but the statement to be executed is:

```
cin >> i >> ch >> ch;
```

Now the stream extraction operator would (after skipping over the leading whitespace) extract the int value 13 to place in the int variable i. It would stop reading the value for i when it reached the period (".") because a period cannot be part of an int value. It would then read the period and place it into the variable ch. It would then continue on, looking for a second char value. It would extract the single character value "7" and place it into the variable ch (destroying the period that was previously stored there). At this point, there would still be a "2W" followed by some whitespace in the input stream. C++ would continue executing statement in the program, but when the next stream extraction operator was applied, "2" would be the first value in the input stream.

In summary: i = 13, ch = "7", x remains unchanged.