FaheemKalwar@Yahoo.Com

Intro to the C++ Language

A C++ program is a collection of commands, which tell the computer to do "something". This collection of commands is usually called **C++ source code**, **source code** or just **code**. Commands are either "functions" or "keywords". Keywords are a basic building block of the language, while functions are, in fact, usually written in terms of simpler functions--you'll see this in our very first program, below. (Confused? Think of it a bit like an outline for a book; the outline might show every chapter in the book; each chapter might have its own outline, composed of sections. Each section might have its own outline, or it might have all of the details written up.) Thankfully, C++ provides a great many common functions and keywords that you can use.

But how does a program actually start? Every program in C++ has one function, always named **main**, that is always called when your program first executes. From main, you can also call other functions whether they are written by us or, as mentioned earlier, provided by the compiler.

So how do you get access to those prewritten functions? To access those standard functions that comes with the compiler, you include a header with the #include directive. What this does is effectively take everything in the header and paste it into your program. Let's look at a working program:

```
#include <iostream.h>

int main()
{
  cout<<"HEY, you, I'm alive! Oh, and Hello World!\n";
  cin.get();
}
```

Let's look at the elements of the program. The #include is a "preprocessor" directive that tells the compiler to put code from the header called iostream into our program before actually creating the executable. By including header files, you an gain access to many different functions. For example, the cout function requires iostream. Following the include is the statement, "using namespace std;". This line tells the compiler to use a group of functions that are part of the standard library (std). By including this line at the top of a file, you allow the program to use functions such as cout. The semicolon is part of the syntax of C and C++. It tells the compiler that you're at the end of a command. You will see later that the semicolon is used to end most commands in C++.

The next imporant line is int main(). This line tells the compiler that there is a function named main, and that the function returns an integer, hence int. The "curly braces" ({ and }) signal the beginning and end of functions and other code blocks. You can think ofknow them as meaning BEGIN and END.

The next line of the program may seem strange. If you have programmed in another language, you might expect that print would be the function used to display text. In C++, however, the cout object is used to display text (pronounced "C out"). It uses the << symbols, known as "insertion operators", to indicate what to output. cout<< results in a function call with the ensuing text as an argument to the function. The quotes tell the compiler that you want to output the literal string as-is. The '\n' sequence is actually treated as a single character that stands for a newline (we'll talk about this later in more detail). It moves the cursor on your screen to the next line. Again, notice the semicolon: it is added onto the end of all, such as function calls, in C++.

The next command is cin.get(). This is another function call: it reads in input and expects the user to hit the return key. Many compiler environments will open a new console window, run the program, and then close the window. This command keeps that window from closing because the program is not done yet because it waits for you to hit enter. Including that line gives you time to see the program run.

Upon reaching the end of main, the closing brace, our program will return the value of 0 (and integer, hence why we told main to return an int) to the operating system. This return value is important as it can be used to tell the OS whether our program succeeded or not. A return value of 0 means success and is returned automatically (but only for main, other functions require you to manually return a value), but if we wanted to return something else, such as 1, we would have to do it with a return statement:

```
#include <iostream>

int main()
{
  cout<<"HEY, you, I'm alive! Oh, and Hello World!\n";
  cin.get();

  return 1;
}
```

The final brace closes off the function. You should try compiling this program and running it. You can cut and paste the code into a file, save it as a .cpp file. Our Code::Blocks tutorial actually takes you through creating a simple program, so check it out if you're confused.

If you are not using Code::Blocks, you should read the compiler instructions for information on how to compile.
Once you've got your first program running, why don't you try playing around with the cout function to get used to writing C++?

An Aside on Commenting Your Programs

As you are learning to program, you should also start to learn how to explain your programs (for yourself, if no one else). You do this by adding comments to code; I'll use them frequently to help explain code examples.

When you tell the compiler a section of text is a comment, it will ignore it when running the code, allowing you to use any text you want to describe the real code. To create a comment use either //, which tells the compiler that the rest of the line is a comment, or /* and then */ to block off everything between as a comment. Certain compiler environments will change the color of a commented area, but some will not. Be certain not to accidentally comment out code (that is, to tell the compiler part of your code is a comment) you need for the program. When you are learning to program, it is useful to be able to comment out sections of code in order to see how the output is affected.

User interaction and Saving Information with Variables

So far you've learned how to write a simple program to display information typed in by you, the programmer, and how to describe your program with comments. That's great, but what about interacting with your user? Fortunately, it is also possible for your program to accept input. The function you use is known as cin, and is followed by the insertion operator >>.

Of course, before you try to receive input, you must have a place to store that input. In programming, input and data are stored in variables. There are several different types of variables which store different kinds of information (e.g. numbers versus letters); when you tell the compiler you are declaring a variable, you must include the data type along with the name of the variable. Several basic types include char , int, and float.

A variable of type char stores a single character, variables of type int store integers (numbers without decimal places), and variables of type float store numbers with decimal places. Each of these variable types - char, int, and float - is each a keyword that you use when you **declare** a variable.

What's with all these variable types?

FaheemKalwar@Yahoo.Com

Sometimes it can be confusing to have multiple variable types when it seems like some variable types are redundant (why have integer numbers when you have floats?). Using the right variable size can be important for making your code readable and for efficiency--some variables require more memory than others. Moreover, because of the way the numbers are actually stored in memory, a float is "inexact", and should not be used when you need to store an "exact" integer value.

Declaring Variables in C++

To declare a variable you use the syntax "type <name>;". Here are some variable declaration examples:

```
int x;
char letter;
float the_float;
```

It is permissible to declare multiple variables of the same type on the same line; each one should be separated by a comma.

```
int a, b, c, d;
```

If you were watching closely, you might have seen that declaration of a variable is always followed by a semicolon (note that this is the same procedure used when you call a function).

Common Errors when Declaring Variables in C++

If you attempt to use a variable that you have not declared, your program will not be compiled or run, and you will receive an error message informing you that you have made a mistake. Usually, this is called an **undeclared variable** Don't forget that variables, just like keywords, are case-sensitive, so it's best to use a consistent capitalization scheme to avoid these errors.

While you can have multiple variables of the same type, you cannot have multiple variables with the same name. Moreover, you cannot have variables and functions with the same name.

Using Variables

Ok, so you now know how to tell the compiler about variables, but what about using them?

Here is a sample program demonstrating the use of a variable:

```
#include <iostream>

using namespace std;
```

```
int main()
{
  int thisisanumber;

  cout<<"Please enter a number: ";
  cin>> thisisanumber;
  cin.ignore();
  cout<<"You entered: "<< thisisanumber <<"\n";
  cin.get();
}
```

Let's break apart this program and examine it line by line. The keyword int declares thisisanumber to be an integer. The function cin>> reads a value into thisisanumber; the user must press enter before the number is read by the program. cin.ignore() is another function that reads and discards a character. Remember that when you type intput into a program, it takes the enter key too. We don't need this, so we throw it away. Keep in mind that the variable was declared an integer; if the user attempts to type in a decimal number, it will be truncated (that is, the decimal component of the number will be ignored). Try typing in a sequence of characters or a decimal number when you run the example program; the response will vary from input to input, but in no case is it particularly pretty. Notice that when printing out a variable quotation marks are not used. Were there quotation marks, the output would be "You Entered: thisisanumber." The lack of quotation marks informs the compiler that there is a variable, and therefore that the program should check the value of the variable in order to replace the variable name with the variable when executing the output function. Do not be confused by the inclusion of two separate insertion operators on one line. Including multiple insertion operators on one line is perfectly acceptable and all of the output will go to the same place. In fact, you **must** separate string literals (strings enclosed in quotation marks) and variables by giving each its own insertion operators (<<). Trying to put two variables together with only one << will give you an error message, do not try it. Do not forget to end functions and declarations with a semicolon. If you forget the semicolon, the compiler will give you an error message when you attempt to compile the program.

Changing and Comparing Variables

Of course, no matter what type you use, variables are uninteresting without the ability to modify them. Several operators used with variables include the following: *, -, +, /, =, ==, >, <. The * multiplies, the - subtracts, and the + adds. It is of course important to realize that to modify the value of a variable inside the program it is rather important to use the equal sign. In some languages, the equal sign compares the value of the left and right values, but in C++ == is used for that task. The equal sign is still extremely useful. It sets the left input to the equal sign, which must be one, and only one, variable equal to the value on the right side of the equal sign. The

operators that perform mathematical functions should be used on the right side of an equal sign in order to assign the result to a variable on the left side.

Here are a few examples:

```
a = 4 * 6; // (Note use of comments and of semicolon) a is 24
a = a + 5; // a equals the original value of a with five added to it
a == 5     // Does NOT assign five to a. Rather, it checks to see if a equals 5.
```

The other form of equal, ==, is not a way to assign a value to a variable. Rather, it checks to see if the variables are equal. It is useful in other areas of C++; for example, you will often use == in such constructions as conditional statements and loops. You can probably guess how < and > function. They are greater than and less than operators.

For example:

```
a < 5  // Checks to see if a is less than five
a > 5  // Checks to see if a is greater than five
a == 5 // Checks to see if a equals five, for good measure
```

Did you follow that? Quiz yourself
Next: If Statements - Conditionally Changing Program Behavior

Lesson 2: If statements(Printable Version)

The ability to control the flow of your program, letting it make decisions on what code to execute, is valuable to the programmer. The if statement allows you to control if a program enters a section of code or not based on whether a given condition is true or false. One of the important functions of the if statement is that it allows the program to select an action based upon the user's input. For example, by using an if statement to check a user entered password, your program can decide whether a user is allowed access to the program.

Without a conditional statement such as the if statement, programs would run almost the exact same way every time. If statements allow the flow of the program to be changed, and so they allow algorithms and more

interesting code.

Before discussing the actual structure of the if statement, let us examine the meaning of TRUE and FALSE in computer terminology. A true statement is one that evaluates to a nonzero number. A false statement evaluates to zero. When you perform comparison with the relational operators, the operator will return 1 if the comparison is true, or 0 if the comparison is false. For example, the check 0 == 2 evaluates to 0. The check 2 == 2 evaluates to a 1. If this confuses you, try to use a cout statement to output the result of those various comparisons (for example cout<< (2 == 1);)

When programming, the aim of the program will often require the checking of one value stored by a variable against another value to determine whether one is larger, smaller, or equal to the other.

There are a number of operators that allow these checks.

Here are the relational operators, as they are known, along with examples:

```
>     greater than              5 > 4 is TRUE
<     less than                 4 < 5 is TRUE
>=    greater than or equal     4 >= 4 is TRUE
<=    less than or equal        3 <= 4 is TRUE
==    equal to                  5 == 5 is TRUE
!=    not equal to              5 != 4 is TRUE
```

It is highly probable that you have seen these before, probably with slightly different symbols. They should not present any hindrance to understanding. Now that you understand TRUE and FALSE in computer terminology as well as the comparison operators, let us look at the actual structure of if statements.

The structure of an if statement is as follows:

```
if ( TRUE )
  Execute the next statement
```

To have more than one statement execute after an if statement that evaluates to true, use braces, like we did with the body of a function. Anything inside braces is called a compound statement, or a block.

For example:

```
if ( TRUE ) {
  Execute all statements inside the braces
```

```
}
```

There is also the else statement. The code after it (whether a single line or code between brackets) is executed if the if statement is FALSE.

It can look like this:

```
if ( TRUE ) {
  // Execute these statements if TRUE
}
else {
  // Execute these statements if FALSE
}
```

One use for else is if there are two conditional statements that may both evaluate to true, yet you wish only one of the two to have the code block following it to be executed. You can use an else if after the if statement; that way, if the first statement is true, the else if will be ignored, but if the if statement is false, it will then check the condition for the else if statement. If the if statement was true the else statement will not be checked. It is possible to use numerous else if statements.

Let's look at a simple program for you to try out on your own.

```
#include <iostream>

using namespace std;

int main()                                  // Most important
part of the program!
{
  int age;                                  // Need a
variable...

  cout<<"Please input your age: ";     // Asks for age
  cin>> age;                                // The input is put
in age
  cin.ignore();                             // Throw away enter
  if ( age < 100 ) {                        // If the age is
less than 100
     cout<<"You are pretty young!\n"; // Just to show you
it works...
  }
  else if ( age == 100 ) {                  // I use else just
to show an example
     cout<<"You are old\n";                 // Just to show you
it works...
```

```
  }
  else {
    cout<<"You are really old\n";      // Executed if no
other statement is
  }
  cin.get();
}
```

Boolean operators allow you to create more complex conditional statements. For example, if you wish to check if a variable is both greater than five and less than ten, you could use the boolean AND to ensure both var > 5 and var < 10 are true. In the following discussion of boolean operators, I will capitalize the boolean operators in order to distinguish them from normal english. The actual C++ operators of equivalent function will be described further into the tutorial - the C++ symbols are not: OR, AND, NOT, although they are of equivalent function.

When using if statements, you will often wish to check multiple different conditions. You must understand the Boolean operators OR, NOT, and AND. The boolean operators function in a similar way to the comparison operators: each returns 0 if evaluates to FALSE or 1 if it evaluates to TRUE.

NOT: The NOT operator accepts one input. If that input is TRUE, it returns FALSE, and if that input is FALSE, it returns TRUE. For example, NOT (1) evalutes to 0, and NOT (0) evalutes to 1. NOT (any number but zero) evaluates to 0. In C and C++ NOT is written as !. NOT is evaluated prior to both AND and OR.

AND: This is another important command. AND returns TRUE if both inputs are TRUE (if 'this' AND 'that' are true). (1) AND (0) would evaluate to zero because one of the inputs is false (both must be TRUE for it to evaluate to TRUE). (1) AND (1) evaluates to 1. (any number but 0) AND (0) evaluates to 0. The AND operator is written && in C++. Do not be confused by thinking it checks equality between numbers: it does not. Keep in mind that the AND operator is evaluated before the OR operator.

OR: Very useful is the OR statement! If either (or both) of the two values it checks are TRUE then it returns TRUE. For example, (1) OR (0) evaluates to 1. (0) OR (0) evaluates to 0. The OR is written as || in C++. Those are the pipe characters. On your keyboard, they may look like a stretched colon. On my computer the pipe shares its key with \. Keep in mind that OR will be evaluated after AND.

It is possible to combine several boolean operators in a single statement; often you will find doing so to be of great value when creating complex expressions for if statements. What is !(1 && 0)? Of course, it would be TRUE. It is true is because 1 && 0 evaluates to 0 and !0 evaluates to TRUE

(ie, 1).

Try some of these - they're not too hard. If you have questions about them, feel free to stop by our forums.

```
A. !( 1 || 0 )          ANSWER: 0
B. !( 1 || 1 && 0 )     ANSWER: 0 (AND is evaluated before
OR)
C. !( ( 1 || 0 ) && 0 )  ANSWER: 1 (Parenthesis are
useful)
```

If you find you enjoyed this section, then you might want to look more at Boolean Algebra.

Quiz yourself
Previous: The Basics
Next: Loops

Lesson 3: Loops

(Printable Version)

Loops are used to repeat a block of code. Being able to have your program repeatedly execute a block of code is one of the most basic but useful tasks in programming -- many programs or websites that produce extremely complex output (such as a message board) are really only executing a single task many times. (They may be executing a small number of tasks, but in principle, to produce a list of messages only requires repeating the operation of reading in some data and displaying it.) Now, think about what this means: a loop lets you write a very simple statement to produce a significantly greater result simply by repetition.

One Caveat: before going further, you should understand the concept of C++'s true and false, because it will be necessary when working with loops (the conditions are the same as with if statements). There are three types of loops: for, while, and do..while. Each of them has their specific uses. They are all outlined below.

FOR - for loops are the most useful type. The syntax for a for loop is

```
for ( variable initialization; condition; variable update
) {
  Code to execute while the condition is true
}
```

The variable initialization allows you to either declare a variable and give it a value or give a value to an already existing variable. Second, the condition tells the program that while the conditional expression is true the loop should continue to repeat itself. The variable update section is the easiest way for a for loop to handle changing of the variable. It is possible to do things like x++, x = x + 10, or even x = random (5), and if you really wanted to, you could call other functions that do nothing to the variable but still have a useful effect on the code. Notice that a semicolon separates each of these sections, that is important. Also note that every single one of the sections may be empty, though the semicolons still have to be there. If the condition is empty, it is evaluated as true and the loop will repeat until something else stops it.

Example:

```
#include <iostream>

using namespace std; // So the program can see cout and
endl

int main()
{
  // The loop goes while x < 10, and x increases by one
every loop
  for ( int x = 0; x < 10; x++ ) {
    // Keep in mind that the loop condition checks
    //  the conditional statement before it loops again.
    //  consequently, when x equals 10 the loop breaks.
    // x is updated before the condition is checked.
    cout<< x <<endl;
  }
  cin.get();
}
```

This program is a very simple example of a for loop. x is set to zero, while x is less than 10 it calls cout<< x <<endl; and it adds 1 to x until the condition is met. Keep in mind also that the variable is incremented after the code in the loop is run for the first time.

WHILE - WHILE loops are very simple. The basic structure is

while (condition) { Code to execute while the condition is true }
The true represents a boolean expression which could be x == 1 or while (x != 7) (x does not equal 7). It can be any combination of boolean statements that are legal. Even, (while x ==5 || v == 7) which says execute the code while x equals five or while v equals 7. Notice that a while loop is the same as a for loop without the initialization and update sections. However, an empty condition is not legal for a while loop as it is with a for loop.

Example:

```
#include <iostream>

using namespace std; // So we can see cout and endl

int main()
{
  int x = 0;  // Don't forget to declare variables

  while ( x < 10 ) { // While x is less than 10
    cout<< x <<endl;
    x++;              // Update x so the condition can be
met eventually
  }
  cin.get();
}
```

This was another simple example, but it is longer than the above FOR loop. The easiest way to think of the loop is that when it reaches the brace at the end it jumps back up to the beginning of the loop, which checks the condition again and decides whether to repeat the block another time, or stop and move to the next statement after the block.

DO..WHILE - DO..WHILE loops are useful for things that want to loop at least once. The structure is

```
do {
} while ( condition );
```

Notice that the condition is tested at the end of the block instead of the beginning, so the block will be executed at least once. If the condition is true, we jump back to the beginning of the block and execute it again. A do..while loop is basically a reversed while loop. A while loop says "Loop while the condition is true, and execute this block of code", a do..while loop says "Execute this block of code, and loop while the condition is true".

Example:

```
#include <iostream>

using namespace std;

int main()
{
  int x;

  x = 0;
  do {
    // "Hello, world!" is printed at least one time
    //  even though the condition is false
    cout<<"Hello, world!\n";
  } while ( x != 0 );
  cin.get();
}
```

Keep in mind that you must include a trailing semi-colon after the while in the above example. A common error is to forget that a do..while loop must be terminated with a semicolon (the other loops should not be terminated with a semicolon, adding to the confusion). Notice that this loop will execute once, because it automatically executes before checking the condition.

Quiz yourself
Previous: If Statements

Lesson 4: Functions

(Printable Version)

Now that you should have learned about variables, loops, and conditional statements it is time to learn about functions. You should have an idea of their uses as we have already used them and defined one in the guise of main. cin.get() is an example of a function. In general, functions are blocks of code that perform a number of pre-defined commands to accomplish something productive.

Functions that a programmer writes will generally require a prototype. Just like a blueprint, the prototype tells the compiler what the function will return, what the function will be called, as well as what arguments the function can be passed. When I say that the function returns a value, I mean that the

function can be used in the same manner as a variable would be. For example, a variable can be set equal to a function that returns a value between zero and four.

For example:

```
#include <cstdlib>   // Include rand()

using namespace std; // Make rand() visible

int a = rand(); // rand is a standard function that all compilers have
```

Do not think that 'a' will change at random, it will be set to the value returned when the function is called, but it will not change again.

The general format for a prototype is simple:

```
return-type function_name ( arg_type arg1, ..., arg_type argN );
```

arg_type just means the type for each argument -- for instance, an int, a float, or a char. It's exactly the same thing as what you would put if you were declaring a variable.

There can be more than one argument passed to a function or none at all (where the parentheses are empty), and it does not have to return a value. Functions that do not return values have a return type of void. Lets look at a function prototype:

```
int mult ( int x, int y );
```

This prototype specifies that the function mult will accept two arguments, both integers, and that it will return an integer. Do not forget the trailing semi-colon. Without it, the compiler will probably think that you are trying to write the actual definition of the function.

When the programmer actually defines the function, it will begin with the prototype, minus the semi-colon. Then there should always be a block with the code that the function is to execute, just as you would write it for the main function. Any of the arguments passed to the function can be used as if they were declared in the block. Finally, end it all with a cherry and a closing brace. Okay, maybe not a cherry.

Lets look at an example program:

```
#include <iostream>

using namespace std;

int mult ( int x, int y );

int main()
{
  int x;
  int y;

  cout<<"Please input two numbers to be multiplied: ";
  cin>> x >> y;
  cin.ignore();
  cout<<"The product of your two numbers is "<< mult ( x,
y ) <<"\n";
  cin.get();
}

int mult ( int x, int y )
{
  return x * y;
}
```

This program begins with the only necessary include file and a directive to make the std namespace visible. Everything in the standard headers is inside of the std namespace and not visible to our programs unless we make them so. Next is the prototype of the function. Notice that it has the final semi-colon! The main function returns an integer, which you should always have to conform to the standard. You should not have trouble understanding the input and output functions. It is fine to use cin to input to variables as the program does. But when typing in the numbers, be sure to separate them by a space so that cin can tell them apart and put them in the right variables.

Notice how cout actually outputs what appears to be the mult function. What is really happening is cout is printing the value returned by mult, not mult itself. The result would be the same as if we had use this print instead

```
cout<<"The product of your two numbers is "<< x * y
<<"\n";
```

The mult function is actually defined below main. Due to its prototype being above main, the compiler still recognizes it as being defined, and so the compiler will not give an error about mult being undefined. As long as the prototype is present, a function can be used even if there is no definition. However, the code cannot be run without a definition even though it will

compile. The prototype and definition can be combined into one also. If mult were defined before it is used, we could do away with the prototype because the definition can act as a prototype as well.

Return is the keyword used to force the function to return a value. Note that it is possible to have a function that returns no value. If a function returns void, the retun statement is valid, but only if it does not have an expression. In otherwords, for a function that returns void, the statement "return;" is legal, but redundant.

The most important functional (Pun semi-intended) question is why do we need a function? Functions have many uses. For example, a programmer may have a block of code that he has repeated forty times throughout the program. A function to execute that code would save a great deal of space, and it would also make the program more readable. Also, having only one copy of the code makes it easier to make changes. Would you rather make forty little changes scattered all throughout a potentially large program, or one change to the function body? So would I.

Another reason for functions is to break down a complex program into logical parts. For example, take a menu program that runs complex code when a menu choice is selected. The program would probably best be served by making functions for each of the actual menu choices, and then breaking down the complex tasks into smaller, more manageable tasks, which could be in their own functions. In this way, a program can be designed that makes sense when read. And has a structure that is easier to understand quickly. The worst programs usually only have the required function, main, and fill it with pages of jumbled code.

Quiz yourself
Previous: Loops
Next: Switch/case

Lesson 6: An introduction to pointers

(Printable Version)

Pointers are an extremely powerful programming tool. They can make some things much easier, help improve your program's efficiency, and even allow you to handle unlimited amounts of data. For example, using pointers is one way to have a function modify a variable passed to it. It is also possible to use pointers to dynamically allocate memory, which means that you can write programs that can handle nearly unlimited amounts of data on the fly--you

don't need to know, when you write the program, how much memory you need. Wow, that's kind of cool. Actually, it's very cool, as we'll see in some of the next tutorials. For now, let's just get a basic handle on what pointers are and how you use them.

 What are pointers? Why should you care?

Pointers are aptly named: they "point" to locations in memory. Think of a row of safety deposit boxes of various sizes at a local bank. Each safety deposit box will have a number associated with it so that the teller can quickly look it up. These numbers are like the memory addresses of variables. A pointer in the world of safety deposit box would simply be anything that stored the number of another safety deposit box. Perhaps you have a rich uncle who stored valuables in his safety deposit box, but decided to put the real location in another, smaller, safety deposit box that only stored a card with the number of the large box with the real jewelery. The safety deposit box with the card would be storing the location of another box; it would be equivalent to a pointer. In the computer, pointers are just variables that store memory addresses, usually the addresses of other variables.

The cool thing is that once you can talk about the address of a variable, you'll then be able to go to that address and retrieve the data stored in it. If you happen to have a huge piece of data that you want to pass into a function, it's a lot easier to pass its location to the function than to copy every element of the data! Moreover, if you need more memory for your program, you can request more memory from the system--how do you get "back" that memory? The system tells you where it is located in memory; that is to say, you get a memory address back. And you need pointers to store the memory address.

A note about terms: the word pointer can refer either to a memory address itself, or to a variable that stores a memory address. Usually, the distinction isn't really that important: if you pass a pointer variable into a function, you're passing the value stored in the pointer--the memory address. When I want to talk about a memory address, I'll refer to it as a memory address; when I want a variable that stores a memory address, I'll call it a pointer. When a variable stores the address of another variable, I'll say that it is "pointing to" that variable.

Pointer Syntax

Pointers require a bit of new syntax because when you have a pointer, you need the ability to request both the memory location it stores and the value stored at that memory location. Moreover, since pointers are somewhat special, you need to tell the compiler when you declare your pointer variable

that the variable is a pointer, and tell the compiler what type of memory it points to.

The pointer declaration looks like this:

```
<variable_type> *<name>;
```

For example, you could declare a pointer that stores the address of an integer with the following syntax:

```
int *points_to_integer;
```

Notice the use of the *. This is the key to declaring a pointer; if you add it directly before the variable name, it will declare the variable to be a pointer. Minor gotcha: if you declare multiple pointers on the same line, you must precede each of them with an asterisk:

```
// one pointer, one regular int
int *pointer1, nonpointer1;

// two pointers
int *pointer1, *pointer2;
```

As I mentioned, there are two ways to use the pointer to access information: it is possible to have it give the actual address to another variable. To do so, simply use the name of the pointer without the *. However, to access the actual memory location, use the *. The technical name for this doing this is dereferencing the pointer; in essence, you're taking the reference to some memory address and following it, to retrieve the actual value. It can be tricky to keep track of when you should add the asterisk. Remember that the pointer's natural use is to store a memory address; so when you use the pointer:

```
call_to_function_expecting_memory_address(pointer);
```

then it evaluates to the address. You have to add something extra, the asterisk, in order to retrieve the value stored at the address. You'll probably do that an awful lot. Nevertheless, the pointer itself is supposed to store an address, so when you use the bare pointer, you get that address back.

Pointing to Something: Retrieving an Address

In order to have a pointer actually point to another variable it is necessary to have the memory address of that variable also. To get the memory address of a variable (its location in memory), put the & sign in front of the variable

name. This makes it give its address. This is called the address-of operator, because it returns the memory address. Conveniently, both ampersand and address-of start with a; that's a useful way to remember that you use & to get the address of a variable.

For example:

```
#include <iostream>

using namespace std;

int main()
{
  int x;              // A normal integer
  int *p;             // A pointer to an integer

  p = &x;             // Read it, "assign the address of x
to p"
  cin>> x;            // Put a value in x, we could also
use *p here
  cin.ignore();
  cout<< *p <<"\n"; // Note the use of the * to get the
value
  cin.get();
}
```

The cout outputs the value stored in x. Why is that? Well, let's look at the code. The integer is called x. A pointer to an integer is then defined as p. Then it stores the memory location of x in pointer by using the address-of operator (&) to get the address of the variable. Using the ampersand is a bit like looking at the label on the safety deposit box to see its number rather than looking inside the box, to get what it stores. The user then inputs a number that is stored in the variable x; remember, this is the same location that is pointed to by p.

The next line then passes *p into cout. *p performs the "dereferencing" operation on p; it looks at the address stored in p, and goes to that address and returns the value. This is akin to looking inside a safety deposit box only to find the number of (and, presumably, the key to) another box, which you then open.

Notice that in the above example, pointer is initialized to point to a specific memory address before it is used. If this was not the case, it could be pointing to anything. This can lead to extremely unpleasant consequences to the program. For instance, the operating system will probably prevent you from accessing memory that it knows your program doesn't own: this will cause your program to crash. If it let you use the memory, you could mess

with the memory of any running program--for instance, if you had a document opened in Word, you could change the text! Fortunately, Windows and other modern operating systems will stop you from accessing that memory and cause your program to crash. To avoid crashing your program, you should always initialize pointers before you use them.

It is also possible to initialize pointers using free memory. This allows dynamic allocation of array memory. It is most useful for setting up structures called linked lists. This difficult topic is too complex for this text. An understanding of the keywords new and delete will, however, be tremendously helpful in the future.

The keyword new is used to initialize pointers with memory from free store (a section of memory available to all programs). The syntax looks like the example:

```
int *ptr = new int;
```

It initializes ptr to point to a memory address of size int (because variables have different sizes, number of bytes, this is necessary). The memory that is pointed to becomes unavailable to other programs. This means that the careful coder should free this memory at the end of its usage.

The delete operator frees up the memory allocated through new. To do so, the syntax is as in the example.

```
delete ptr;
```

After deleting a pointer, it is a good idea to reset it to point to 0. When 0 is assigned to a pointer, the pointer becomes a null pointer, in other words, it points to nothing. By doing this, when you do something foolish with the pointer (it happens a lot, even with experienced programmers), you find out immediately instead of later, when you have done considerable damage.

In fact, the concept of the null pointer is frequently used as a way of indicating a problem--for instance, some functions left over from C return 0 if they cannot correctly allocate memory (notably, the malloc function). You want to be sure to handle this correctly if you ever use malloc or other C functions that return a "NULL pointer" on failure.

In C++, if a call to new fails because the system is out of memory, then it will "throw an exception". For the time being, you need not worry too much about this case, but you can read more about what happens when new fails.

Taking Stock of Pointers

FaheemKalwar@Yahoo.Com

Pointers may feel like a very confusing topic at first but I think anyone can come to appreciate and understand them. If you didn't feel like you absorbed everything about them, just take a few deep breaths and re-read the lesson. You shouldn't feel like you've fully grasped every nuance of when and why you need to use pointers, though you should have some idea of some of their basic uses.

Quiz yourself
Previous: Switch/case
Next: Structures

Lesson 8: Array basics (Printable Version)

Arrays are useful critters because they can be used in many ways. For example, a tic-tac-toe board can be held in an array. Arrays are essentially a way to store many values under the same name. You can make an array out of any data-type including structures and classes.

Think about arrays like this:

```
[][][][][][]
```

Each of the bracket pairs is a slot(element) in the array, and you can put information into each one of them. It is almost like having a group of variables side by side.

Lets look at the syntax for declaring an array.

```
int examplearray[100]; // This declares an array
```

This would make an integer array with 100 slots, or places to store values(also called elements). To access a specific part element of the array, you merely put the array name and, in brackets, an index number. This corresponds to a specific element of the array. The one trick is that the first index number, and thus the first element, is zero, and the last is the number of elements minus one. 0-99 in a 100 element array, for example.

What can you do with this simple knowledge? Lets say you want to store a string, because C had no built-in datatype for strings, it was common to use arrays of characters to simulate strings. (C++ now has a string type as part of the standard library.)

For example:

```
char astring[100];
```

will allow you to declare a char array of 100 elements, or slots. Then you can receive input into it it from the user, and if the user types in a long string, it will go in the array. The neat thing is that it is very easy to work with strings in this way, and there is even a header file called cstring. There is another lesson on the uses of strings, so its not necessary to discuss here.

The most useful aspect of arrays is multidimensional arrays. How I think about multi-dimensional arrays:

```
[][][][][]
[][][][][]
[][][][][]
[][][][][]
[][][][][]
```

This is a graphic of what a two-dimensional array looks like when I visualize it.

For example:

```
int twodimensionalarray[8][8];
```

declares an array that has two dimensions. Think of it as a chessboard. You can easily use this to store information about some kind of game or to write something like tic-tac-toe. To access it, all you need are two variables, one that goes in the first slot and one that goes in the second slot. You can even make a three dimensional array, though you probably won't need to. In fact, you could make a four-hundred dimensional array. It would be confusing to visualize, however. Arrays are treated like any other variable in most ways. You can modify one value in it by putting:

```
arrayname[arrayindexnumber] = whatever;
```

or, for two dimensional arrays

```
arrayname[arrayindexnumber1][arrayindexnumber2] =
whatever;
```

However, you should never attempt to write data past the last element of the array, such as when you have a 10 element array, and you try to write to the

[10] element. The memory for the array that was allocated for it will only be ten locations in memory, but the next location could be anything, which could crash your computer.

You will find lots of useful things to do with arrays, from storing information about certain things under one name, to making games like tic-tac-toe. One suggestion I have is to use for loops when access arrays.

```
#include <iostream>

using namespace std;

int main()
{
  int x;
  int y;
  int array[8][8]; // Declares an array like a chessboard

  for ( x = 0; x < 8; x++ ) {
    for ( y = 0; y < 8; y++ )
      array[x][y] = x * y; // Set each element to a value
  }
  cout<<"Array Indices:\n";
  for ( x = 0; x < 8;x++ ) {
    for ( y = 0; y < 8; y++ )
      cout<<"["<<x<<"]["<<y<<"]="<< array[x][y] <<" ";
    cout<<"\n";
  }
  cin.get();
}
```

Here you see that the loops work well because they increment the variable for you, and you only need to increment by one. Its the easiest loop to read, and you access the entire array.

One thing that arrays don't require that other variables do, is a reference operator when you want to have a pointer to the string. For example:

```
char *ptr;
char str[40];
ptr = str;  // Gives the memory address without a
reference operator(&)
```

As opposed to

```
int *ptr;
int num;
```

```
ptr = &num; // Requires & to give the memory address to
the ptr
```

The reason for this is that when an array name is used as an expression, it refers to a pointer to the first element, not the entire array. This rule causes a great deal of confusion, for more information please see our Frequently Asked Questions.

Quiz yourself
Previous: Structures
Next: Strings

Lesson 9: C Strings

(Printable Version)

In C++ there are two types of strings, C-style strings, and C++-style strings. This lesson will discuss C-style strings. C-style strings are really arrays, but there are some different functions that are used for strings, like adding to strings, finding the length of strings, and also of checking to see if strings match. The definition of a string would be anything that contains more than one character strung together. For example, "This" is a string. However, single characters will not be strings, though they can be used as strings.

Strings are arrays of chars. String literals are words surrounded by double quotation marks.

```
"This is a static string"
```

To declare a string of 49 letters, you would want to say:

```
char string[50];
```

This would declare a string with a length of 50 characters. Do not forget that arrays begin at zero, not 1 for the index number. In addition, a string ends with a null character, literally a '\0' character. However, just remember that there will be an extra character on the end on a string. It is like a period at the end of a sentence, it is not counted as a letter, but it still takes up a space. Technically, in a fifty char array you could only hold 49 letters and one null character at the end to terminate the string.

TAKE NOTE: char *arry; Can also be used as a string. If you have read the tutorial on pointers, you can do something such as:

```
arry = new char[256];
```

which allows you to access arry just as if it were an array. Keep in mind that to use delete you must put [] between delete and arry to tell it to free all 256 bytes of memory allocated.

For example:

```
delete [] arry.
```

Strings are useful for holding all types of long input. If you want the user to input his or her name, you must use a string. Using cin>> to input a string works, but it will terminate the string after it reads the first space. The best way to handle this situation is to use the function cin.getline. Technically cin is a class (a beast similar to a structure), and you are calling one of its member functions. The most important thing is to understand how to use the function however.

The prototype for that function is:

```
istream& getline(char *buffer, int length, char
terminal_char);
```

The char *buffer is a pointer to the first element of the character array, so that it can actually be used to access the array. The int length is simply how long the string to be input can be at its maximum (how big the array is). The char terminal_char means that the string will terminate if the user inputs whatever that character is. Keep in mind that it will discard whatever the terminal character is.

It is possible to make a function call of cin.getline(arry, 50); without the terminal character. Note that '\n' is the way of actually telling the compiler you mean a new line, i.e. someone hitting the enter key.

For a example:

```
#include <iostream>

using namespace std;

int main()
{
```

```
  char string[256];                                    // A
nice long string

  cout<<"Please enter a long string: ";
  cin.getline ( string, 256, '\n' );                   //
Input goes into string
  cout<<"Your long string was: "<< string <<endl;
  cin.get();
}
```

Remember that you are actually passing the address of the array when you pass string because arrays do not require an address operator (&) to be used to pass their address. Other than that, you could make '\n' any character you want (make sure to enclose it with single quotes to inform the compiler of its character status) to have the getline terminate on that character.

cstring is a header file that contains many functions for manipulating strings. One of these is the string comparison function.

```
int strcmp ( const char *s1, const char *s2 );
```

strcmp will accept two strings. It will return an integer. This integer will either be:

```
Negative if s1 is less than s2.
Zero if s1 and s2 are equal.
Positive if s1 is greater than s2.
```

Strcmp is case sensitive. Strcmp also passes the address of the character array to the function to allow it to be accessed.

```
char *strcat ( char *dest, const char *src );
```

strcat is short for string concatenate, which means to add to the end, or append. It adds the second string to the first string. It returns a pointer to the concatenated string. Beware this function, it assumes that dest is large enough to hold the entire contents of src as well as its own contents.

```
char *strcpy ( char *dest, const char *src );
```

strcpy is short for string copy, which means it copies the entire contents of src into dest. The contents of dest after strcpy will be exactly the same as src such that strcmp (dest, src) will return 0.

```
size_t strlen ( const char *s );
```

strlen will return the length of a string, minus the termating character ('\0'). The size_t is nothing to worry about. Just treat it as an integer that cannot be negative, which it is.

Here is a small program using many of the previously described functions:

```
#include <iostream> //For cout
#include <cstring>  //For the string functions

using namespace std;

int main()
{
  char name[50];
  char lastname[50];
  char fullname[100]; // Big enough to hold both name and
lastname

  cout<<"Please enter your name: ";
  cin.getline ( name, 50 );
  if ( strcmp ( name, "Julienne" ) == 0 ) // Equal
strings
    cout<<"That's my name too.\n";
  else                                    // Not equal
    cout<<"That's not my name.\n";
  // Find the length of your name
  cout<<"Your name is "<< strlen ( name ) <<" letters
long\n";
  cout<<"Enter your last name: ";
  cin.getline ( lastname, 50 );
  fullname[0] = '\0';             // strcat searches for
'\0' to cat after
  strcat ( fullname, name );      // Copy name into full
name
```

```
  strcat ( fullname, " " );       // We want to separate
the names by a space
  strcat ( fullname, lastname ); // Copy lastname onto
the end of fullname
  cout<<"Your full name is "<< fullname <<"\n";
  cin.get();
}
```

Safe Programming

The above string functions all rely on the existence of a null terminator at the end of a string. This isn't always a safe bet. Moreover, some of them,

noticeably strcat, rely on the fact that the destination string can hold the entire string being appended onto the end. Although it might seem like you'll never make that sort of mistake, historically, problems based on accidentally writing off the end of an array in a function like strcat, have been a major problem.

Fortunately, in their infinite wisdom, the designers of C have included functions designed to help you avoid these issues. Similar to the way that fgets takes the maximum number of characters that fit into the buffer, there are string functions that take an additional argument to indicate the length of the destination buffer. For instance, the strcpy function has an analogous strncpy function

```
char *strncpy ( char *dest, const char *src, size_t len
);
```

which will only copy len bytes from src to dest (len should be less than the size of dest or the write could still go beyond the bounds of the array). Unfortunately, strncpy can lead to one niggling issue: it doesn't guarantee that dest will have a null terminator attached to it (this might happen if the string src is longer than dest). You can avoid this problem by using strlen to get the length of src and make sure it will fit in dest. Of course, if you were going to do that, then you probably don't need strncpy in the first place, right? Wrong. Now it forces you to pay attention to this issue, which is a big part of the battle.

Quiz yourself
Previous: Arrays
Next: File I/O

Lesson 12: Introduction to Classes (Printable Version)

C++ is a bunch of small additions to C, with a few major additions. One major addition is the object-oriented approach (the other addition is support for generic programming, which we'll cover later). As the name object-oriented programming suggests, this approach deals with objects. Of course, these are not real-life objects themselves. Instead, these objects are the essential definitions of real world objects. Classes are collections of data related to a single object type. Classes not only include information regarding the real world object, but also functions to access the data, and classes possess the ability to inherit from other classes. (Inheritance is covered in a later lesson.)

FaheemKalwar@Yahoo.Com

If a class is a house, then the functions will be the doors and the variables will be the items inside the house. The functions usually will be the only way to modify the variables in this structure, and they are usually the only way even to access the variables in this structure. This might seem silly at first, but the idea to make programs more modular - the principle itself is called "encapsulation". The key idea is that the outside world doesn't need to know exactly what data is stored inside the class--it just needs to know which functions it can use to access that data. This allows the implementation to change more easily because nobody should have to rely on it except the class itself.

The syntax for these classes is simple. First, you put the keyword 'class' then the name of the class. Our example will use the name Computer. Then you put an open bracket. Before putting down the different variables, it is necessary to put the degree of restriction on the variable. There are three levels of restriction. The first is public, the second protected, and the third private. For now, all you need to know is that the public restriction allows any part of the program, including parts outside the class, to access the functions and variables specified as public. The protected restriction prevents functions outside the class to access the variable. The private restriction is similar to protected (we'll see the difference later when we look at inheritance. The syntax for declaring these access restrictions is merely the restriction keyword (public, private, protected) and then a colon. Finally, you put the different variables and functions (You usually will only put the function prototype[s]) you want to be part of the class. Then you put a closing bracket and semicolon. Keep in mind that you still must end the function prototype(s) with a semi-colon.

Let's look at these different access restrictions for a moment. Why would you want to declare something private instead of public? The idea is that some parts of the class are intended to be internal to the class--only for the purpose of implementing features. On the other hand, some parts of the class are supposed to be available to anyone using the class--these are the public class functions. Think of a class as though it were an appliance like a microwave: the public parts of the class correspond to the parts of the microwave that you can use on an everyday basis--the keypad, the start button, and so forth. On the other hand, some parts of the microwave are not easily accessible, but they are no less important--it would be hard to get at the microwave generator. These would correspond to the protected or private parts of the class--the things that are necessary for the class to function, but that nobody who uses the class should need to know about. The great thing about this separation is that it makes the class easier to use (who would want to use a microwave where you had to know exactly how it works in order to use it?) The key idea is to separate the interface you use from the

way the interface is supported and implemented.

Classes must always contain two functions: a constructor and a destructor. The syntax for them is simple: the class name denotes a constructor, a ~ before the class name is a destructor. The basic idea is to have the constructor initialize variables, and to have the destructor clean up after the class, which includes freeing any memory allocated. If it turns out that you don't need to actually perform any initialization, then you can allow the compiler to create a "default constructor" for you. Similarly, if you don't need to do anything special in the destructor, the compiler can write it for you too!

When the programmer declares an instance of the class, the constructor will be automatically called. The only time the destructor is called is when the instance of the class is no longer needed--either when the program ends, the class reaches the end of scope, or when its memory is deallocated using delete (if you don't understand all of that, don't worry; the key idea is that destructors are always called when the class is no longer usable). Keep in mind that neither constructors nor destructors return arguments! This means you do not want to (and cannot) return a value in them.

Note that you generally want your constructor and destructor to be made public so that your class can be created! The constructor is called when an object is created, but if the constructor is private, it cannot be called so the object cannot be constructed. This will cause the compiler to complain.

The syntax for defining a function that is a member of a class outside of the actual class definition is to put the return type, then put the class name, two colons, and then the function name. This tells the compiler that the function is a member of that class.

For example:

```
#include <iostream>

using namespace std;

class Computer // Standard way of defining the class
{
public:
  // This means that all of the functions below this(and
any variables)
  //  are accessible to the rest of the program.
  //  NOTE: That is a colon, NOT a semicolon...
  Computer();
  // Constructor
```

```
  ~Computer();
  // Destructor
  void setspeed ( int p );
  int readspeed();
protected:
  // This means that all the variables under this, until
a new type of
  //  restriction is placed, will only be accessible to
other functions in the
  //  class.  NOTE: That is a colon, NOT a semicolon...
  int processorspeed;
};
// Do Not forget the trailing semi-colon

Computer::Computer()
{
  //Constructors can accept arguments, but this one does
not
  processorspeed = 0;
}

Computer::~Computer()
{
  //Destructors do not accept arguments
}

void Computer::setspeed ( int p )
{
  // To define a function outside put the name of the
class
  //  after the return type and then two colons, and then
the name
  //  of the function.
  processorspeed = p;
}
int Computer::readspeed()
{
  // The two colons simply tell the compiler that the
function is part
  //  of the class
  return processorspeed;
}

int main()
{
  Computer compute;
  // To create an 'instance' of the class, simply treat
it like you would
```

```
  //  a structure.  (An instance is simply when you
create an actual object
  //  from the class, as opposed to having the definition
of the class)
  compute.setspeed ( 100 );
  // To call functions in the class, you put the name of
the instance,
  //  a period, and then the function name.
  cout<< compute.readspeed();
  // See above note.
}
```

This introduction is far from exhaustive and, for the sake of simplicity, recommends practices that are not always the best option. For more detail, I suggest asking questions on our forums and getting a book recommended by our book reviews.

Quiz yourself
Previous: Typecasting
Next: Functions Continued

Lesson 13: More on Functions (Printable Version)

In lesson 4 you were given the basic information on functions. However, I left out one item of interest. That item is the inline function. Inline functions are not very important, but it is good to understand them. The basic idea is to save time at a cost in space. Inline functions are a lot like a placeholder. Once you define an inline function, using the 'inline' keyword, whenever you call that function the compiler will replace the function call with the actual code from the function.

How does this make the program go faster? Simple, function calls are simply more time consuming than writing all of the code without functions. To go through your program and replace a function you have used 100 times with the code from the function would be time consuming not too bright. Of course, by using the inline function to replace the function calls with code you will also greatly increase the size of your program.

Using the inline keyword is simple, just put it before the name of a function. Then, when you use that function, pretend it is a non-inline function.

For example:

```
#include <iostream>

using namespace std;

inline void hello()
{
  cout<<"hello";
}
int main()
{
  hello(); //Call it like a normal function...
  cin.get();
}
```

However, once the program is compiled, the call to hello(); will be replaced by the code making up the function.

A WORD OF WARNING: Inline functions are very good for saving time, but if you use them too often or with large functions you will have a tremendously large program. Sometimes large programs are actually less efficient, and therefore they will run more slowly than before. Inline functions are best for small functions that are called often.

Finally, note that the compiler may choose, in its infinite wisdom, to ignore your attempt to inline a function. So if you do make a mistake and inline a moster fifty-line function that gets called thousands of times, the compiler may ignore you.

In the future, we will discuss inline functions in terms of C++ classes. Now that you understand the concept I will feel more comfortable using inline functions in later tutorials.

Quiz yourself
Previous: Classes
Next: Reading command-line arguments

Lesson 14: Accepting command line arguments (Printable Version)

In C++ it is possible to accept command line arguments. Command-line arguments are given after the name of a program in command-line operating systems like DOS or Linux, and are passed in to the program from the operating system. To use command line arguments in your program, you

must first understand the full declaration of the main function, which previously has accepted no arguments. In fact, main can actually accept two arguments: one argument is number of command line arguments, and the other argument is a full list of all of the command line arguments.

The full declaration of main looks like this:

```
int main ( int argc, char *argv[] )
```

The integer, argc is the ARGument Count (hence argc). It is the number of arguments passed into the program from the command line, including the name of the program.

The array of character pointers is the listing of all the arguments. argv[0] is the name of the program, or an empty string if the name is not available. After that, every element number less than argc are command line arguments. You can use each argv element just like a string, or use argv as a two dimensional array. argv[argc] is a null pointer.

How could this be used? Almost any program that wants its parameters to be set when it is executed would use this. One common use is to write a function that takes the name of a file and outputs the entire text of it onto the screen.

```
#include <fstream>
#include <iostream>

using namespace std;

int main ( int argc, char *argv[] )
{
  if ( argc != 2 ) // argc should be 2 for correct
execution
    // We print argv[0] assuming it is the program name
    cout<<"usage: "<< argv[0] <<" <filename>\n";
  else {
    // We assume argv[1] is a filename to open
    ifstream the_file ( argv[1] );
    // Always check to see if file opening succeeded
    if ( !the_file.is_open() )
      cout<<"Could not open file\n";
    else {
      char x;
```

```
      // the_file.get ( x ) returns false if the end of
the file
      //  is reached or an error occurs
      while ( the_file.get ( x ) )
        cout<< x;
    }
    // the_file is closed implicitly here
  }
}
```

This program is fairly simple. It incorporates the full version of main. Then it first checks to ensure the user added the second argument, theoretically a file name. The program then checks to see if the file is valid by trying to open it. This is a standard operation that is effective and easy. If the file is valid, it gets opened in the process. The code is self-explanatory, but is littered with comments, you should have no trouble understanding its operation this far into the tutorial. :-)

Quiz yourself
Previous: Functions Continued
Next: Linked Lists