✓ Lesson 1: The basics of C++

I am writing this for those people who want to learn how to program in C++, especially those who had trouble. It is for those of you who want a sense of accomplishment every time your program works perfectly. If you want the sense of accomplishment, read on.

C++ is a programming language. It is a programming language of many different dialects, just like each language that is spoken has many dialects. In C though, they are not because the "speakers" live in the North, South, or grew up in some other place, it is because there are so many compilers. There are about four major ones: Borland C++, Microsoft Visual C++, Watcom C/386, and DJGPP. You can download DJGPP http://www.delorie.com/djgpp/ or you may already have another compiler.

Each of these compilers is a little different. The library functions of one will have all of the standard C++ functions, but they will also have other functions or, continuing the analogy, words. At times, this can lead to confusion, as certain programs will only run under certain compilers, though I do not believe this to be the case with the programs in these tutorials.

++
If you don't have a compiler, I strongly suggest you get one. A simple one is good enough for my tutorials, but get one.
++
C++ is a different breed of programming language. It has only a few keywords for DOS, and it has no keywords to use for output. This means that almost everything is stored in a header file. This gives the use of many functions. But lets see a real program...

```
#include <iostream.h>

int main()
{
  cout<<"HEY, you, I'm alive!  Oh, and Hello World!";
  return 0;
}
```

That does not look too hard, right? Lets break down the program and then look at it. The #include is a preprocessor directive which tells the compiler to put code in the header file iostream.h into our program! By including header files, you an gain access to many different functions. For example, the cout function requires iostream.h.

The next thing is int main() what this is saying is that there is a function called main, and that it returns an integer, hence int. Then those little braces ({ and }) are used to signal the beginning and ending of functions, as well as other code blocks. If you have programmed in Pasal, you will know them as BEGIN and END.

The next line of the program may seem strange. If you have programmed in other languages you might think that print would be used to display text. However, in C++ the cout function is used to display text. It uses the << symbols, known as insertion operators. The quotes tell the compiler that you want to output the literal string as-is. The ; is added to the end of all function calls in C++.

The penultimate line of code is ordering main to return 0. When one returns a value to main, it is passed on to the operating system. As a note, declaring int main() or void main() both will generally work. It is accepted practice to some to declare main as a void, but to others it is
extremely upsetting. Previously, these tutorials had used void main, however, this is NO LONGER recommended, as it does not conform to the ANSI standard.

After, the brace closes off the function. You can try out this program if you want, just cut and paste it into the IDE of a compiler such as DJGPP, or save it to a file ending with a .cpp extension, and use a command-line compiler to compile and link it.

Comments are extremely important to understand. When you declare that an area is a comment, the compiler will IGNORE it. To comment it is possible to use either // , which declares that the entire line past that point is a comment, or it is possible to use /* and then */ to block off everything between the two as a comment. Certain compilers will change the color of a ommented area, but some will not. Be certain not accidently declare part of your code a comment. Note that this is what is known as "commenting-out" a section of code, and it is useful when you are debugging.

So far you should be able to write a simple program to display information typed in by you, the programmer. However, it is also possible for your program to accept input. the function you use is known as cin>>.

Wait! Before you can receive input you must have a place to store input! In programming, these locations where input and other forms of data are stored, are called variables. There are a few different types of variables, which must be stated. The basic types are char, int, and float.

Char is used to create variables that store characters, int is used to create variables that store integers (numbers such as 1, 2, 0, -3, 44, -44), and float is used to delare numbers with decimal places. In fact, they are all keywords that are used in front of variable names to tell the compiler that you have created a variable. That is known as "declaring a variable". When you declare a variable, or variables, you must end the line with a semi-colon, the same as if you were to call a function. If you do not declare the variable you are attempting to use, you will receive numerous error messages and the program will not run.

Here are some examples of declaring variables:

```
int x;
int a, b, c, d;
char letter;
float the_float;
```

It is not possible, however, to declare two variables of different types with the same name.

```
#include <iostream.h>
int main()
{
  int thisisanumber;
  cout<<"Please enter a number:";
  cin>>thisisanumber;
  cout<<"You entered: "<<thisisanumber;
  return 0;
}
```

Let's break apart this program and examine it line by line. Int is the keyword that is used when delcaring a variable which is an integer. The cin>> sets the value of thisisanumber to be whatever the user types into the program when prompted. Keep in mind that the variable was declared an integer, which means the output will be in the form of an integer. Try typing in a sequence of charaters, or a decimal when you run the example program to see what you get as a response. Notice that when printing out a variable, there are not any quotation marks. If there were quotation marks, the output would be "You Entered: thisisanumber." Do not be confused by the inclusion of two separate insertion operators on a line. It is allowable, as long as you make certain to have each separate output of variable or string with its own insertion operator. Do not try to put two variables together with only one << because it will give you an error message. Do not forget to end functions and declarations with the semi-colon(;). Otherwise you will get an error message when you try to compile the program.

Now that you know a little bit about variables, here are some ways to manipulate them. *, -, +, /, =, ==, >, < are all operators used on numbers, these are the simple ones. The * multiplies, the - subtracts, and the + adds. Of course, the most important for changing variables
is the equal sign. In some languages, = checks if one side is equal to the other side, but in C++ == is used for that task. However, the equal sign is still extremely useful. It sets the left side of the equal sign, which must be one AND ONLY one variable, equal to the right side. The right side of the equal sign is where the other operators can be used.

Here are a few examples:

```
a=4*6;  //(Note use of comments and of semi-colon) a is 24
a=a+5;  // a equals the original value of a with five additional units
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++ such as if statements and loops.

You can probably guess what the < and > are for. They are greater than and less than checks.

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
```

Note: My homepage is http://www.cprogramming.com. My email is webmaster@cprogramming.com. Please email me with comments and or suggestions. If you want to use this on your own site please
email me and add a link to http://www.cprogramming.com. Thanks :)

✓ *Lesson 2: If statements*

The if statement is one of the most powerful devices in programming. It allows the program to choose an action based upon input. For example, a program can be written to decide if a user can access the program by using an if statement to check if the password is correct or not.

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

Please note that all Boolean operators or values will be capitalized to differentiate them from normal English. Note, as well, that the actual C++ symbols to the operators are described later, but the C++ symbols are not OR, NOT, and AND. The symbols simply mean the same thing.

There are many things to understand when using if statements. You must understand Boolean operators such as OR, NOT, and AND. You must also understand comparisons between numbers and most especially, the idea of true and false in the same way computers do.

True and false in C++ are denoted by a non-zero number, and zero. Therefore, 1 is TRUE and 8.3 is TRUE but 0 is FALSE. No other number is interpreted as being false.

When programming, the program often will require the checking of one value stored by a variable against another value, to determine which is larger, smaller, or if the two are equal.

To do so, there are a number of operators used.

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

It is highly probable that you have seen these before in some mathematical manifestation, probably with slightly different symbols. They should not present any hindrance understanding.

More interesting are the Boolean operators. They return 0 for FALSE, and a nonzero number for TRUE.

NOT: This just says that the program should reverse the value. For example, NOT (1) would be 0. NOT (0) would be 1. NOT (any number but zero) would be 0. In C and C++ NOT is written as !.

AND: This is another important command, it returns a one if 'this' AND 'this' are true. (1) AND (0) would come out as 0 because both are not true, only 1 is true. (1) AND (1) would come out as 1. (ANYREAL NUMBER BUT ZERO) AND (0) would be 0. (ANY REAL NUMBER BUT ZERO) AND (ANY REAL NUMBER BUTZERO) would be 1. The AND is written as && in C++. Do not be confused and think that it checks equality between numbers. It does not. Keep in mind that AND will be evaluated before OR.

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) would be 1! (0)OR(0) would be0. (ANY REAL NUMBER) OR (ANY REAL NUMBER BUT ZERO) would be 1! 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 it shares its key with \. Keep in mind that OR will be evaluated after AND.

The next thing to learn is to combine them... What is !(1 && 0)? Of course, it would be TRUE (1). This is because 1 && 0 evaluates two 0 and ! 0 equals 1.

Try some of these...they are not hard. If you have questions about them, you can email meat lallain@concentric.net.

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 enjoy this you might want to look more at Boolean Algebra, which is also very helpful to programmers as it is useful for helping program conditional statements.

The structure of an if statement is essentially:

```
if (TRUE)
  Do whatever follows on the next line.
```

To have more than one statement execute after an if statement (when it evaluates to true) use brackets.

For example,

```
if (TRUE)
{
  Do everything between the brackets.
}
```

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(FALSE)
{
  Not executed if its false
}
else
{
  do all of this
}
```

One use for else is if there are two conditional statements that will both evalueate to true with the same value(but only at times. For example, x<30 and x<50 will both return true if x is less than 30 but not otherwise), but you wish only one of them to be checked. You can use else after the if statement. If the if statement was true the else statement will not be checked. It is possible to use numerous else statements followed by if statements.

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

```
#include <iostream.h>

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

  if(age<100)                                   //If the age is less than 100
  {
    cout<<"You are pretty young!";     //Just to show it works
  }
  else if(age==100)             //I use else just to show an example
  {
    cout<<"You are old";                //Just to show you it works...
  }
  else if(age>100)
  {
   cout<<"You are really old";          //Proof that it works for any condition
  }
  return 0;
}
```

Now, this program did not use && || ! or anything in it. This is because it did not need too. Its purpose was to demonstrate if statements.

Note: My homepage is http://www.cprogramming.com. My email is webmaster@cprogramming.com. Please email me with comments and or suggestions. If you want to use this on your own site please email me and add a link to http://www.cprogramming.com. Thanks :)

✓ *Lesson 3: Loops*

Loops are used to repeat a block of code. You should understand the concept of C++'s true and false, because it will be necessary when working with loops.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 layout is for(variable initialization, conditional expression, modification of variable) The variable initialization allows you to either declare a variable and give it a value or give a value to an already declared variable. Second, the conditional expression tells the program that while the conditional expression is true the loop should continue to repeat itself. The variable modification 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. That would be something ridiculous probably.

Ex.

```
 #include <iostream.h>  //We only need one header file
int main()              //We always need this
 {              //The loop goes while x<100, and x increases by one every loop
  for(int x=0;x<100;x++) //Keep in mind that the loop condition checks the x value before it
  {                                    //Actually repeats, or loops, again.  So if x==100 the loop will end.      cout<<x<<endl;
//Outputting x
 }
  return 0;                            }
```

This program is a very simple example of a for loop. x is set to zero, while x is less than 100 it calls cout<<x<<endl; and it adds 1 to x until the loop ends. 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(true) then execute all the code in the loop. 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..

Ex.

```
#include <iostream.h>   //We only need this header file

int main()                   //Of course...
{
  int x=0;                          //Don't forget to declare variables
 while(x<100)                       //While x is less than 100 do
 {
    cout<<x<<endl;//Same output as the above loop
    x++;              //Adds 1 to x every time it repeats, in for loops the                               //loop structure allows this
to be done in the structure
  }
 return 0;
}
```

This was another simple example, but it is longer than the above FOR loop. The easiest way to think of the loop is to think the code executes, when it reaches the brace at the end it goes all the way back up to while, which checks the boolean expression.

DO WHILE - DO WHILE loops are useful for only things that want to loop at least once. The structure is DO {THIS} WHILE (TRUE);

For example:

```
#include <iostream.h>

int main()
{
  int x;
  x=0;
  do
  {
    cout<<"Hello world!";
  }while(x!=0);
  return 0;
}
```

Keep in mind that you must include a trailing semi-colon after while in the above example. Notice that this loop will also execute once, because it automatically executes before checking the truth statement.

✓ *Lesson 4: Loops*

Now that you should have learned about variables, loops, and if statements it is time to learn about functions. You should have an idea of their uses. Cout is an example of a function. In general, functions perform a number of pre-defined commands to accomplish something productive.

Functions that a programmer writes will generally require a prototype. Just like an 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:

```
int a;
a=random(5); //random is sometimes defined by the compiler
                    //Yes, it returns between 0 and the argument minus 1
```

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 arg);
```

There can be more than one argument passed to a function, and it does not have to return a value. 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 bracket (remember, functions require brackets around them) followed by code, just as you would write it for the main function. Finally, end it all with a cherry and a bracket. Okay, maybe not a cherry.

Lets look at an example program:

```
#include <iostream.h>

int mult(int x, int y);

int main()
{
  int x, y;
  cout<<"Please input two numbers to be multiplied: ";
  cin>>x>>y;
  cout<<"The product of your two numbers is "<<mult(x, y);
  return 0;
}

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

This program begins with the only necessary include file. It is followed by the prototye of the function. Notice that it has the final semi-colon! The main function is 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.

Notice how cout actually outputs what appears to be the mult function. What is really happening is that mult acts as a variable. Because it returns a value it is possible for the cout function to output the return value.

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, although the definition is below where it is used.

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. In that case, the prototype would have a return type of void.

The most important functional (Pun semi-intended) question is why. 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.

Another reason for functions is to break down a complex program into something manageable. 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 takes, which could be in their own functions. In this way, a program can be designed that makes sense when read.

Note: My homepage is http://www.cprogramming.com. My email is webmaster@cprogramming.com. Please email me with comments and suggestions. If you want to use this text on your own site, please email me and add a link to http://www.cprogramming.com. Thanks :)

✓ *Lesson 5: switch case*

Switch case statements are a substitute for long if statements. The basic format for using switch case is outlined below.

```
Switch (expression or variable)
{
 case variable equals this:
  do this;
  break;
  case variable equals this:
  do this;
  break;
  case variable equals this:
  do this;
  break;
          ...
  default:
  do this
}
```

The expression or variable has a value. The case says that if it has the value of whatever is after that cases then do whatever follows the colon. The break is used to break out of the case statements. Break is a keyword that breaks out of the code block, usually surrounded by braces, which it is in. In this case, break prevents the program from testing the next case statement also.

Switch case serves as a simple way to write long if statements. Often it can be used to process input from a user.

Below is a sample program, in which not all of the proper functions are actually declared, but which shows how one would use switch case in a program.

```
#include <iostream.h>
#include <conio.h>
int main()
{
  char input;
  cout<<"1. Play game";
  cout<<"2. Load game";
  cout<<"3. Play multiplayer";
  cout<<"4. Exit";
  cin>>input;
  switch (input)
  {
   case 1: playgame();
              break;
   case 2:
              loadgame();
        break;
   case 3:      //Note use of : not ;
        playmultiplayer();
              break;
   case 4:
        return 0;
  default:
        cout<<"Error, bad input, quitting";
  }
  return 0;
}
```

This program will not compile yet, but it serves as a model (albeit simple) for processing input.

If you do not understand this then try mentally putting in if statements for the case statements. Note that using return in the middle of main will automatically end the program. Default simply skips out of the switch case construction and allows the program to terminate naturally. If you do not like that, then you can make a loop around the whole thing to have it wait for valid input. I know that some functions were not prototyped. You could easily make a few small functions if you wish to test the code.

Note: My homepage is http://www.cprogramming.com. My email is webmaster@cprogramming.com. Please email me with comments and or suggestions. If you want to use this on your own site please email me and add a link to http://www.cprogramming.com. Thanks :)

✓ *Lesson 6: An introduction to pointers*

Pointers can be confusing, and at times, you may wonder why you would ever want to use them. The truth is, they can make some things much easier. 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 allows certain programming techniques, such as linked lists.
Pointers are what they sound like...pointers. They point to locations in memory. Picture a big jar that holds the location of another jar. In the other jar holds a piece of paper with the number 12 written on it. The jar with the 12 is an integer, and the jar with the memory address of the 12 is a pointer

Pointer syntax can also be confusing, because pointers can both give the memory location and give the actual value stored in that same location. When a pointer is declared, the syntax is this: variable_type *name; Notice the *. This is the key to declaring a pointer, if you use it before the variable name, it will declare the variable to be a pointer.

As I have said, there are two ways to use the pointer to access information about the memory address it points to. It is possible to have it give the actual address to another variable, or to pass it into a function. 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.

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 the variable, put the & sign in front of the variable name. This makes it give its address. This is called the reference operator, because it returns the memory address.

For example:
```
#include <iostream.h>
int main()
{
 int x;          //A normal integer
 int *pointer;   //A pointer to an integer
 pointer=&x;      //Read it, "pointer equals the address of x"
 cin>>x;         //Reads in x
 cout<<*pointer; //Note the use of the * to output the actual number stored in x
 return 0;
}
```

The cout outputs the value in x. Why is that? Well, look at the code. The integer is called x. A pointer to an integer is then defined as "pointer". Then it stores the memory location of x in pointer by using the ampersand (&) symbol. If you wish, you can think of it as if the jar that had the integer had a ampersand in it then it would output its name (in pointers, the memory address) Then the user inputs the value for x. Then the cout uses the * to put the value stored in the memory location of pointer. If the jar with the name of the other jar in it had a * in front of it would give the value stored in the jar with the same name as the one in the jar with the name. It is not too hard, the * gives the value in the location. The unastricked gives the memory location.

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 computer. 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:

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 will 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.

Example:
```
Delete ptr;
```

After deleting a pointer, it is a good idea to reset it to point to NULL. NULL is a standard compiler-defined statement that sets the pointer to point to, literally, nothing. By doing this, you minimize the potential for doing something foolish with the pointer.

The final implication of NULL is that if there is no more free memory, it is possible for the ptr after being "new"-ed to point to NULL. Therefore, it is good programming practice to check to ensure that the pointer points to something before using it. Obviously, the program is unlikely to work without this check.

✓ *Lesson 7: Structures*

Before discussing classes, this lesson will be an introduction to data structures similar to classes. Structures are a way of storing many different variables of different types under the same name.

The format for declaring a structure(in C++) is

```
struct NAME
{
  VARIABLES;
};
```

Where NAME is the name of the entire type of structure. To actually create a single structure the syntax is NAME name_of_single_structure; To access a variable of the structure it goes name_of_single_structure.name_of_variable;

For example,

```
struct example
{
  int x;
};

example an_example;
an_example.x=33;
```

Here is an example program

```
struct database
{
  int id_number;
  int age;
  float salary;
};

int main()
{
  database employee;  //There is now an employee variable that has modifiable
                                  //variables inside it.
  employee.age=22;
  employee.id_number=1;
  employee.salary=12000.21;
  return 0;
}
```

The struct database declares that database has three variables in it, age, id_number, and salary.

You can use database like a variable type like int. You can create an employee with the database type as I did above. Then, to modify it you call everything with the 'employee.' in front of it. You can also return structures from functions by defining their return type as a structure type. Example:

```
struct database fn();
```

I suppose I should explain unions a little bit. They are
like structures except that all the variables share the same memory. When a union is declared the compiler allocates enough memory for the largest data-type in the union. Its like a giant storage chest where you can store one large item, or a bunch of small items, but never the both at the same time.

The '.' operator is used to access different variables inside a union also.

As a final note, if you wish to have a pointer to a structure, to actually access the information stored inside the structure that is pointed to, you use the -> operator in place of the . operator.

A quick example:

```
#include <iostream.h>

struct xampl
{
  int x;
};

int main()
{
  xampl structure;
  xampl *ptr;
  ptr=&structure; //Yes, you need the & when dealing with structures
                  //and using pointers to them
  cout<<ptr->x;  //The -> acts somewhat like the * when used with pointers
                                          //It says, get whatever is at that memory address
                                          //Not "get what that memory address is"
  return 0;
}
```

Note: My homepage is http://www.cprogramming.com. My email is webmaster@cprogramming.com. Please email me with comments and or suggestions. If you want to use this on your own site please email me and add a link to http://www.cprogramming.com. Thanks :)

✓ *Lesson 8: Array basics*

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++ has no built-in datatype for strings, at least in DOS, you can make an array of characters.

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 string.h. There is another lesson on the uses of string.h, 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 11 element. The memory for the array that was allocated for it will only be ten locations in memory, but the twelfth could be anything, which could crash your computer.

You will find lots of useful things to do with arrays, from store 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.h>

int main()
{
  int x, y, anarray[8][8];//declares an array like a chessboard
  for(x=0; x<8; x++)
  {
    for(y=0; y<8; y++)
    {
      anarray[x][y]=0;//sets the element to zero, once the loops finish all elements will be 0
    }
  }
  for(x=0; x<8;x++)
  {
    for(y=0; y<8; y++)
    {
      cout<<"anarray["<<x<<"]["<<y<<"]="<<anarray[x][y]<<" ";//you'll see
    }
  }
 return 0;
}
```

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
```

Note: My homepage is http://www.cprogramming.com. My email is webmaster@cprogramming.com. Please email me with comments and or suggestions. If you want to use this on your own site please email me and add a link to http://www.cprogramming.com. Thanks :)

✓ *Lesson 9: Strings*

In C++ 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. Static strings are words surrounded by double quotation marks.

"This is a static string"

To declare a string of 50 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, 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:
cin.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, '\n'); without the length, or vice versa, 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.h>

int main()
{
  char string[256];   //A nice long string
  cout<<"Please enter a long string: ";
  cin.getline(string, 256, '\n'); //The user input goes into string
  cout<<"Your long string was:"<<endl<<string;
  return 0;
}
```

Remember that you are actually passing the address of the array when you pass string because arrays do not require a reference 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.

String.h 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.

int strcmpi(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 the s1 and s2 are equal.
Positive if s1 is greater than s2.

Strcmpi is not case sensitive, if the words are capitalized it does not matter.

char *strcat(char *desc, 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.

char *strupr(char *s);

strupr converts a string to uppercase. It also returns a string, which will all be in uppercase. The input string, if it is an array and not a static string, will also all be uppercase.

char *strlwr(char *s);

strlwr converts a string to lowercase. It also returns a string, which will all be in uppercase. The input string, if it is an array, will also all be uppercase.

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, which it is.

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

```
#include <iostream.h>   //For cout
#include <string.h>     //For many of the string functions

int main()
{

  char name[50];            //Declare variables
  char lastname[50];        //This could have been declared on the last line...
  cout<<"Please enter your name: ";   //Tell the user what to do
  cin.getline(name, 50, '\n');      //Use gets to input strings with spaces or
//just to get strings after the user presses enter

  if(!strcmpi("Alexander", name))  //The ! means not, strcmpi returns 0 for
  {                              //equal strings
    cout<<"That's my name too."<<endl; //Tell the user if its my name
  }
  else                            //else is used to keep it from always
  {                                                                //outputting this line
    cout<<"That's not my name."
  }

  cout<<"What is your name in uppercase..."<<endl;
  strupr(name);                  //strupr converts the string to uppercase
  cout<<name<<endl;
  cout<<"And, your name in lowercase..."<<endl;
  strlwr(name);                   //strlwr converts the string to lowercase
  cout<<name<<endl;
  cout<<"Your name is "<<strlen(name)<<" letters long"<<endl;  //strlen returns
//the length of the string
  cout<<"Enter your last name:";
  gets(lastname);                    //lastname is also a string
  strcat(name, " ");                                          //We want to space the two names apart
  strcat(name, lastname);           //Now we put them together, we a space in
//the middle
  cout<<"Your full name is "<<name; //Outputting it all...
}
```

Note: My homepage is http://www.cprogramming.com. My email is webmaster@cprogramming.com. Please email me with comments and or suggestions. If you want to use this on your own site please email me and add a link to http://www.cprogramming.com. Thanks :)

✓ *Lesson 10: C++ File I/O*

This is a slightly more advanced topic than what I have covered so far, but I think that it is useful. File I/O is reading from and writing to files. This lesson will only cover text files, that is, files that are composed only of ASCII text.

C++ has two basic classes to handle files, ifstream and ofstream. To use them, include the header file fstream.h. Ifstream handles file input (reading from files), and ofstream handles file output (writing to files). The way to declare an instance of the ifstream or ofstream class is:

```
ifstream a_file;
//or
ifstream a_file("filename");
```

The constructor for both classes will actually open the file if you pass the name as an argument. As well, both classes have an open command (a_file.open()) and a close command (a_file.close()). It is generally a good idea to clean up after yourself and close files once you are finished.

The beauty of the C++ method of handling files rests in the simplicity of the actual functions used in basic input and output operations. Because C++ supports overloading operators, it is possible to use << and >> in front of the instance of the class as if it were cout or cin.

For example:

```
#include <fstream.h>
#include <iostream.h>

int main()
{
  char str[10];
                                  //Used later
  ofstream a_file("example.txt");
                                  //Creates an instance of ofstream, and opens example.txt
  a_file<<"This text will now be inside of example.txt";
                                  //Outputs to example.txt through a_file
  a_file.close();
                                  //Closes up the file

  ifstream b_file("example.txt");
                                  //Opens for reading the file
  b_file>>str;
            //Reads one string from the file
  cout<<str;
            //Should output 'this'
  b_file.close();
          //Do not forget this!
}
```

The default mode for opening a file with ofstream's constructor is to create it if it does not exist, or delete everything in it if something does exist in it. If necessary, you can give a second argument that specifies how the file should be handled. They are listed below:

ios::app -- Opens the file, and allows additions at the end
ios::ate -- Opens the file, but allows additions anywhere
ios::trunc -- Deletes everything in the file
ios::nocreate -- Does not open if the file must be created
ios::noreplace -- Does not open if the file already exists

For example:
```
ifstream a_file("test.txt", ios::nocreate);
```

The above code will only open the file test.txt if that file does not already exist.

Note: My homepage is http://www.cprogramming.com. My email is webmaster@cprogramming.com. Please email me with comments and or suggestions. If you want to use this on your own site please email me and add a link to http://www.cprogramming.com. Thanks :)

✓ *Lesson 11: Typecasting*

Typecasting is making a variable of one type, such as an int, act like another type, a char, for one single application.

To typecast something, simply put the type of variable you want the actual variable to act as inside parentheses in front of the actual variable. (char)a will make 'a' function as a char.

For example:

```
#include <iostream.h>

int main()
{
  cout<<(char)65;
   //The (char) is a typecast, telling the computer to interpret the 65 as a
  //character, not as a number.  It is going to give the ASCII output of the
  //equivalent of the number 65(It should be the letter A).
  return 0;
}
```

One use for typecasting for is when you want to use the ASCII characters. For example, what if you want to create your own chart of all 256 ASCII characters. To do this, you will need to use to typecast to allow you to print out the integer as its character equivalent.

```
#include <iostream.h>

int main()
{
  for(int x=0; x<256; x++)
  {                          //The ASCII character set is from 0 to 255

    cout<<x<<". "<<(char)x<<" ";
                                                         //Note the use of the int version of x to
                             //output a number and the use of (char) to
                                                         // typecast the x into a character
                                   //which outputs the ASCII character that
                                                         //corresponds to the current number
  }
  return 0;
}
```

Note: My homepage is http://www.cprogramming.com. My email is webmaster@cprogramming.com. Please email me with comments and or suggestions. If you want to use this on your own site please email me and add a link to http://www.cprogramming.com. Thanks :)

✓ *Lesson 12: Introduction to Classes*

C++ is a bunch of small additions to C, and one major addition. This one addition is the object-oriented approach. As its name suggests, this deals with objects. Of course, these are not real-life objects. Instead, these objects are the essential definitions of real world objects. Structures are one step away from these objects, they do not possess one element of them: functions. The definitions of these objects are called classes. The easiest way to think about a class is to imagine a structure that has functions.

What is this mysterious structure (not the programming type)? Well, it is not only a collection of variables under one heading, but it is a collection of functions under that same heading. If the structure is a house, then the functions will be the doors and the variables will be the items inside the house. They usually will be the only way to modify the variables in this structure, and they are usually the only to access the variables in this structure.

From now on, we shall call these structures with functions classes (I guess Marx would not like C++). 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 that which is not part of the class, access the variables specified as public. The protected restriction prevents functions outside the class to access the variable. The syntax for that 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.

Classes should always contain two functions: the constructor and 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. The only time the constructor is called is when the programmer declares an instance of the class, which will automatically call the constructor. The only time the destructor is called is when the instance of the class is no longer needed. When the program ends, or when its memory is deallocated (if you do not understand the deallocation part, do not worry). Keeps in mind this: NEITHER constructors NOR destructors RETURN AN ARGUMENT! This means you do not want to try to return a value in them.

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:
```
void Aclass::aFunction()
{
  cout<<"Whatever code";
}
```

```
#include <iostream.h>

class Computer //Standard way of defining the class
{

 public:
            //This means that all of the functions below this(and variables, if there were any)
            //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();
                        //These functions will be defined outside of the class
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;
                            //Initializes it to zero
}

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

//The destructor does not need to do anything.

void Computer::setspeed(int p)
{       //To define a function outside put the name of the function
              //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.
  return 0;
}
```

As you can see, this is a rather simple concept. However, it is very powerful. It makes it easy to prevent variables that are contained (or owned) by the class being overwritten accidentally. It also allows a totally different way of thinking about programming. I want to end this tutorial as an introduction, however.

Note: My homepage is http://www.cprogramming.com. My email is webmaster@cprogramming.com. Please email me with comments and or suggestions. If you want to use this on your own site please email me and add a link to http://www.cprogramming.com. Thanks :)

✓ *Lesson 13: More on Functions*

In lesson 4 you were given the basic information on functions. However, I left out two items of interest. First, when you declare a function you do not have to prototype it! You must give the function definition physically before you call the function. You simply type in the entire definition of the function where you would normally put the prototype.

```
 For example:
#include <iostream.h>
void function(void)
{      //Normally this would be the prototype. Do not include a semicolon
              //Only prototypes have semicolons
  cout<<"HA!  NO PROTOTYPE!";
}

int main()
{
  function();
 //It works like a normal function now.
 return 0;
}
```

The other programming concept 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. 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.h>
inline void hello(void)
{                         //Just use the inline keyword before the function      cout<<"hello";
}
int main()
{
 hello();
   //Call it like a normal function...
  return 0;
}
```

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.

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

Note: My homepage is http://www.cprogramming.com. My email is webmaster@cprogramming.com. Please email me with comments and or suggestions. If you want to use this on your own site please email me and add a link to http://www.cprogramming.com. Thanks :)

✓ Lesson 14: Accepting command line arguments

In C++ it is possible to accept command line arguments. To do so, you must first understand the full definition of int main(). It actually accepts two arguments, one is number of command line arguments, the other is a listing of the command line arguments.

It looks like this:

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

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

The array of character pointers is the listing of all the arguments. argv[0] is entire path to the program including its name. 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.

How could this be used? Almost any program that wants it 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.h> //Needed to manipulate files
#include <iostream.h>
#include <io.h>

int main(int argc, char * argv[])
{
  if(argc!=2)
  {
    cout<<"Sorry, invalid input";
    return 0;
  }
  if(access(argv[1], 00)) //access returns 0 if the file can be accessed
  {                       //under the specified method (00 is passed in
    cout<<"File does not exist";  //because it checks file existence
    return 0;
  }
  ifstream the_file;  //ifstream is used for file input
  the_file.open(argv[1]); //argv[1] is the second argument passed in
                                                    //presumable the file name
  char x;
  the_file.get(x);
  while(x!=EOF)  //EOF is defined as the end of the file
  {
    cout<<x;
    the_file.get(x);//Notice we always let the loop check x for the end of
  }            //file to avoid bad output when it is reached
  the_file.close();  //Always clean up
}
```

This program is fairly simle. It first checks to ensure the user added the second argument, theoretically a file name. It checks this, using the access function, which accepts a file name and an access type, with 00 being a check for existence. This is not an standard C++ function. It may not work on your compiler<Then it creates an instance of the file input class,and it opens the second argument passed into main. If you have not seen get before, it is a standard function used in input and output that is used to input a single character into the character passed to it, or by returning the character. EOF is simply the end of file marker, and x is checked to make sure that the next output is not bad.

✓ *Lesson 15: Singly linked lists*

Linked lists are a way to store data with structures so that the programmer can automatically create a new place to store data whenever necessary. Specifically, the programmer writes a struct or class definition that contains variables holding information about something, and then has a pointer to a struct of its type. Each of these individual struct or classes in the list is known as a node.

Think of it like a train. The programmer always stores the first node of the list. This would be the engine of the train. The pointer is the connector between cars of the train. Every time the train ads a car, it uses the connectors to add a new car. This is like a programmer using the keyword new to create a pointer to a new struct or class.

In memory it is often described as looking like this:

```
----------            ----------
- Data   -        >- Data   -
----------    -  ----------
- Pointer- - -    - Pointer-
----------        ----------
```

Each of the big blocks is a struct (or class) that has a pointer to another one. Remember that the pointer only stores the memory location of something, it is not that thing, so the arrow goes to the next one. At the end, there is nothing for the pointer to point to, so it does not point to anything, it should be set to "NULL" to prevent it from accidentally pointing to a totally arbitrary and random location in memory (which is very bad).

So far we know what the node struct should look like:

```
struct node
{
  int x;
  node *next;
};

int main()
{
  node *root;  //This will be the unchanging first node
  root=new node; //Now root points to a node struct
  root->next=NULL; //The node root points to has its next pointer
                                    //set equal to NULL
  root->x=5;   //By using the -> operator, you can modify the node
  return 0;              //a struct (root in this case) points to.
}
```

This so far is not very useful for doing anything. It is necessary to understand how to traverse (go through) the linked list before going further.

Think back to the train. Lets imagine a conductor who can only enter the train through the engine, and can walk through the train down the line as long as the connector connects to another car. This is how the program will traverse the linked list. The conductor will be a pointer to node, and it will first point to root, and then, if the root's pointer to the next node is pointing to something, the "conductor" (not a technical term) will be set to point to the next node. In this fashion, the list can be traversed. Now, as long as there is a pointer to something, the traversal will continue. Once it reaches a NULL pointer, meaning there are no more nodes (train cars) then it will be at the end of the list, and a new node can subsequently be added if so desired.

Here's what that looks like:

```
struct node
{
  int x;
  node *next;
};

int main()
{
  node *root; //This won't change, or we would lose the list in memory
  node *conductor; //This will point to each node as it traverses
                                    //the list
  root=new node; //Sets it to actually point to something
  root->next=NULL; //Otherwise it would not work well
  root->x=12;
  conductor=root; //The conductor points to the first node
  if(conductor!=NULL)
  {
    while(conductor->next!=NULL)
    {
      conductor=conductor->next;
    }
  }
  conductor->next=new node; //Creates a node at the end of the list
  conductor=conductor->next; //Points to that node
  conductor->next=NULL; //Prevents it from going any further
  conductor->x=42;
}
```

That is the basic code for traversing a list. The if statement ensures that there is something to begin with (a first node). In the example it will always be so, but if it was changed, it might not be true. If the if statement is true, then it is okay to try and access the node pointed to by conductor. The while loop will continue as long as there is another pointer in the next. The conductor simply moves along. It changes what it points to by getting the address of conductor->next.

Finally, the code at the end can be used to add a new node to the end. Once the while loop as finished, the conductor will point to the last node in the array. (Remember the conductor of the train will move on until there is nothing to move on to? It works the same way in the while loop.) Therefore, conductor->next is set to null, so it is okay to allocate a new area of memory for it to point to. Then the conductor traverses one more element(like a train conductor moving on the the newly added car) and makes sure that it has its pointer to next set to NULL so that the list has an end. The NULL functions like a period, it means there is no more beyond. Finally, the new node has its x value set. (It can be set through user input. I simply wrote in the '=42' as an example.)

To print a linked list, the traversal function is almost the same. It is necessary to ensure that the last element is printed after the while loop terminates.

For example:

```
conductor=root;
if(conductor!=NULL) //Makes sure there is a place to start
{
  while(conductor->next!=NULL)
  {
    cout<<conductor->x;
    conductor=conductor->next;
  }
  cout<<conductor->x;
}
```

The final output is necessary because the while loop will not run once it reaches the last node, but it will still be necessary to output the contents of the next node. Consequently, the last output deals with this.

✓ *Lesson 16: Recursion*

Recursion is defined as a fuction calling itself. It is in some ways similar to a loop because it repeats the same code, but it requires passing in the looping variable and being more careful. Many programming languages allow it because it can simplify some tasks, an it is often more elegant than a loop.

A simple example of recursion would be:

```
void recurse()
{
  recurse(); //Function calls itself
}

int main()
{
  recurse(); //Sets off the recursion
  return 0;  //Rather pitiful, it will never be reached
}
```

This program will not continue forever, however. The computer keeps function calls on a stack and once too many are called within ending, the program will terminate. Why not write a program to see how many times the function is called before the program terminates?

```
\
#include <iostream.h>
void recurse(int count) //The count variable will be initalized by each function call
{
  cout<<count;
  recurse(count+1);    //It is not necessary to increment count, each function's variables
}                                //are separate (so each count will be initialized one greater)

int main()
{
  recurse(1);        //First function call, so it starts at one
  return 0;
}
```

This simple program will show the number of times the recurse function has been called by initializing each individual function call's count variable one greater than it was previous by passing in count+1. Keep in mind, it is not a function restarting itself, it is hundreds of functions that are each unfinished with the last one calling a new recurse function.

It can be thought of like those little chinese dolls that always have a smaller doll inside. Each doll calls another doll, and you can think of the size being a counter variable that is being decremented by one.

Think of a really tiny doll, the size of a few atoms. You can't get any smaller than that, so there are no more dolls. Normally, a recursive function will have a variable that performs a similar action; one that controls when the function will finally exit. This is often called the 'end condition' of the function. Basically, it is an if-statement that checks some variable for a condition (such as a number being less than zero, or greater than some other number) and if that condition is true, it will not allow the function to call itself again. (Or, it could check if a certain condition is true, normally the contrary of its end condition, and only then allow the function to call itself).

A quick example:

```
void doll(int size)
{
  if(size==0)  //No doll can be smaller than 1 atom (10^0==1) so doesn't call itself again
    return;    //Return does not have to return something, it can be used to exit a function
  doll(size-1);  //Decrements the size variable so the next doll will be smaller.
}

int main()
{
  doll(10);   //Starts off with a large doll (its a logarithmic scale)
  return 0;   //Finally, it will be used
}
```

This program ends when size equals one. This is a good end condition, but if it is not properly set up, it is possible to have an end condition that is always true (or always false, which is not so bad).

Once a function has called itself, it will be ready to go to the next line after the call. It can still perform operations. One function you could write could print out the numbers 123456789987654321. How can you use recursion to write a function to do this? Simply have it keep incrementing a variable passed in, and then output the variable...twice, once before the function recurses, and once after...

```
void printnum(int begin)
{
  cout<<begin;
  if(begin<9)  //The end condition is when begin is greater than 9
    printnum(begin+1);    //for it will not recurse after the if statement.
  cout<<begin;  //Outputs the second begin, after the program has gone through and output
}                          //the numbers from begin to 9.
```

This function works because it will go through and print the numbers begin to 9, and then as each printnum function terminates it will continue printing the value of begin in each function from 9 to begin.

This is just the beginning of the usefulness of recursion. Heres a little challenge, use recursion to write a program that returns the factorial of any number greater than 0. (Factorial is number*number-1*number-2...*1).

Hint: Recursively find the factorial of the smaller numbers first, ie, it takes a number, finds the factorial of the previous number, and multiplies the number times that factorial...have fun, email me at webmaster@cprogramming.com if you get it.

Note: My homepage is http://www.cprogramming.com. My email is webmaster@cprogramming.com. Please email me with comments and or suggestions. If you want to use this on your own site please email me and add a link to http://www.cprogramming.com. Thanks :)