The *for* Statement

Structure:
for(initial value; terminating condition; update expression) statement;
The test for termination is done prior to execution of the loop body (statement).
Examples:
for(i=1; i<=10; i++)
    cout << i << endl;

When is the *for* statement used? In general, it is used when you want to repeat a loop a predetermined number of times; although, it is not restricted to this case. You don’t have to use literal values:

for (int j=K; j<=M; j=j+N){ body }
In this example, K, M, and N are assumed to have already been computed when the loop is reached. j ranges over the interval [K,M] and increments in jumps of N. What happens if K > M?

Special Notes and Warning

- The initialization of the controlling variable is done once, and this occurs before anything else in the loop.
- The test for whether or not to do the body of the loop is prior to the loop (pre-test loop).
- The update of the controlling variable occurs after each iteration of the loop.
- A dangling semicolon results in a serious logical error; e.g., `for(i=1;i<5;i++);{body}` results in the body being executed exactly once, regardless of the conditions inside the *for* statement.

*break and continue* statements

These statements can be used to alter the normal flow of execution in any of the three loop structures, *for*, *while*, and *do-while*.

*break*: The execution of *break* inside a loop results in leaving the loop and resuming execution with the next statement following the loop structure.

*continue*: The execution of *continue* inside a loop causes all statements following the continue (and inside the loop) to be skipped and starting the loop over.

In a *while* or *do-while* structure *continue* causes control to be passed to the test condition of the loop. In the *for* statement it causes control to be passed to the update statement.

*break* and *continue* can cause the readability of the program to become more difficult; however, they sometimes have naturally good uses.

Question: Given a logical structure resulting from use of a *break* statement inside a loop, could you create an equivalent logical structure by changing the test conditions in the original loop structure and eliminating the *break*?
A Perhaps Unusual Example

What happens with the following:

```cpp
for ( ; ; ){
    series of statements
    if(condition) break;
}
```

The statement

```cpp
for( ; ; ){ body }
```

has no initialization, test for termination, or update statement. As a result it sets up a loop-forever condition. A way out is with a `break`; executed when some condition is true.

Do you think there would ever be a situation for which this is the natural way to set up a loop; i.e., a loop that loops until one or more of several conditions inside the loop become true?

The Pascal language has a REPEAT-UNTIL structure.

```pascal
REPEAT
    body of loop
UNTIL(condition);
```

The loop is done until the condition becomes true, and it is a post-test loop. What C++ structure could be used to simulate this naturally?

Problem:

```cpp
for(i=1; i<=10; i++){
    for(j=1; j<=i; j++)
        cout << '*';
    cout << endl;
}
```

Show what the output will be. How many total *'s will be printed?

Functions

Terminology
- function name
- function return type
- arguments
- formal parameters
- return value
- function call (or invocation)
- function prototype

Examples

```cpp
y = sqrt(x);
```

sqrt is the function name. In this context, `sqrt(x)` creates a call to the function `sqrt`. The variable `x` in this case is an argument and represents the value to be passed to the `sqrt` function. `sqrt(x)` is computed and a single value is returned. That value is then becomes the value of the variable `y`. 
cout.setf(ios::fixed);
cout.setf(ios::showpoint);
cout.precision(2);
These you will use in future assignments. They are functions associated with classes in iostream. They perform actions. To see if a value is returned, we would need to see the function definition. In the context here, no return value is assumed.

A function is a subprogram. It normally is designed to accomplish a single task each time it is called, (normally a function should not perform side tasks), but will likely be called more than once to perform that task multiple times. Different values may be supplied to a function via the argument list in the function call. A function may or may not return a value, but if it does, only one value is returned. The type of the returned value, must be specified in the function definition.

Predefined Functions

The header file, cmath, contains a number of arithmetic and trigonometric functions.
The header file, cstdlib, contains useful functions as well.
A partial list of these functions is found on page 96 in your text.
Remember, that in order to use a predefined function, you must include the header file containing the function information.

rand() and srand(arg)

rand() is in cstdlib. It has no arguments (but remember the ( ) must still be included in a call to the function), and when it is called it returns a “random” integer in the range from 0 to RAND_MAX. RAND_MAX is defined in cstdlib and is the largest random number that can be returned. How would you determine the value of RAND_MAX on the system you are using?

Each time rand( ) is called, it returns the next computer “random” number; however, each time you execute a program, it starts at the same place giving the same sequence of random numbers. To force it to start at a different place the srand(seed) function is provided. The starting point depends on the value of the integer argument, seed. This will be discussed. It is often best not to include srand until your program is debugged and working correctly.

Execute srand( ) only once in order to determine a new starting point for your sequence of pseudo-random numbers. Don’t place it inside a loop, which would restart the sequence of values based on the seed value.
Structure of a Function

return type function_name(arguments){

    body of function; i.e., the statements defining what the function does.

}

Example – What’s this doing?

double pay_amount(double P, double R, int N) {
    R=R/1200.0;
    return P*R/(1.0-exp(-N * log(1.0+R)));
}