Tutorial: Programming in C
Chapter 2, 6/27/93
The First Program: Compound Interest

The first program we examine will calculate compound interest of a savings account. Certain things in the program won't be explained in great detail the first time around, but just enough to be used; I'll come back to them in greater detail later.

I'll give simple instructions on how to compile and run the program under the Unix environment. Compiling and running under other environments, particularly PC-based ones, is usually simpler, and I refer you to the online help (and the MANUALS!) of your compiler software.

Everything between the two lines consisting of three dashes is part of the text of the program, and you will want to either key it into your editor yourself or clip it out of this message and paste it into the editor (if you're an editing wiz). The three-dash lines themselves lines should *NOT* go into the program. Name the file interest.c. If you're using vi on a Unix system, the command to start editing this file is "vi interest.c", entered at the % prompt without the quotes.

/* interest.c, by Tom Boutell, 6/27/93. Note that this text is preceded by a slash and an asterisk. This means that this text is a comment, intended to be read by you or another programmer looking at your code. The comment is ended by another asterisk and another slash. Note that you cannot put comments inside comments. */

/* This program calculates the value of a savings account after a given number of years, with a given initial investment, rate of interest and frequency with which interest is computed. */

/* Get standard input and output functions */ #include

/* Get standard math functions */ #include

int main() { /* Initial balance (money in account). Since this value can have a fractional part, we declare a float (floating point) variable to store it. */ float initial_balance;

/* Rate of interest, per year (also a floating point value) */ float interest;

/* Number of times interest is compounded each year (interest periods) (thus 1.0 is annually, 365.0 is daily) */ float frequency;

/* Time in years */ float years;

/* Total interest periods. This cannot have a fractional part, so we declare an integer (no fractional part) variable to store it. */ int interest_periods;

/* Current balance. (We store this in a separate place form the initial balance, so we will still be able to tell how much money we started with when the calculation is finished.) */

float balance;

/* Counter of interest periods */ int i;

/* Initial values */ initial_balance = 1000.0; interest = .05; /* Five per cent interest */ frequency = 12.0; /* Compounded monthly: 12 times per year */ years = 4.0; /* For four years */

/* Calculate number of interest periods */ interest_periods = frequency * years;

/* Set working balance to begin at initial balance. */ balance = initial_balance;

/* Loop through interest periods, increasing balance */ for (i=0; (i /* Each period, multiply balance by 1.0 plus the annual rate of interest divided by the number of times per year (frequency) with which interest is compounded. */ balance = balance * (1.0 + (interest/frequency)); } /* Print out result */ printf("Initial balance: %f Final balance: %f\n", initial_balance, balance); /* Everything went fine */ return 0; }

Let's consider the program line by line to learn how it works.
The program begins with several comments. (Anything contained between /* and */ is a comment in a C program.) Comments aren't read by the compiler; they're for your benefit, and for the benefit of other programmers who must read your code.
Next, several #include directives appear.
#include tells the compiler to treat another file as part of your code. Usually this is done to bring in information about common functions that are used in most programs but are not part of the language itself. (Unlike languages such as BASIC, C does not define functions like "print" and "input." Such things are written *in* the language, and have no special powers that your own functions do not have.) There is a standard set of functions to perform input and output, and these are fetched by the #include line. The #include line fetches useful functions for calculations on floating point numbers; we do not actually use any of these special functions in the program as it stands, but it is common practice to include when using floating point numbers (numbers with fractions), as some compilers are likely to mishandle them otherwise.
Following the #include directives, the actual program begins.
FUNCTIONS: Every program in the C language is a collection of functions. The purpose of a function is to perform a particular task, large or small. Some functions are defined in the "standard C library", and we inform the compiler we will be using these functions by using the #include directives described above. Others we define ourselves, breaking down the tasks our program must perform into subtasks and writing additional functions to perform those tasks until the program is complete.
Every C program has ONE function called "main". The "main" function is where the program begins; typically the main function calls other functions to perform various tasks until the program's overall goal has been achieved.
The line:
int main() {
begins the main function.
The "int" indicates what kind of value this function will return to the function that called it.
"But this is main()! You just said main() goes first! Nobody calls it!"
Well, not quite -- you call it from the % prompt (or the C> prompt, or the prompt of another operating system). Technically speaking, the "shell" called the program, and the shell can take advantage of the value you return from main to determine whether the program "worked" or not. This is very useful when writing shell scripts (under Unix) or .BAT files (under MSDOS).
The two parentheses after main represent the list of "arguments" (parameters) the function can be called with. Since this is a simple main() function, it takes no arguments, and so there is nothing between the parentheses. (More sophisticated programs *do* take arguments to main(). This will be discussed later.)
The "{" at the end of the line signals the beginning of the actual code making up the function. "{" marks the beginning of a "compound statement" (series of statements making up one unit, to be performed one after the other) in the C language.
Next, in the main() function (and in any function), come the variable declarations.
("Variables", with which you will be familiar if you have worked in any programming language, are needed to store values in your program, and to provide meaningful names by which to refer to those values.)
Consider the line:

float initial_balance;
In order to compute compound interest, we need to know how much money the account starts out with. initial_balance is the variable in which we will store this quantity.
Money amounts often contain fractions, or (if you're lucky) are very large numbers. For that reason, we need to store them as floating-point numbers.
So the word "float" appears preceding the name of the variable.
"float" is a "type specifier". There must always be a type specifier in a variable declaration, and variables MUST ALWAYS BE DECLARED, UNLIKE IN BASIC, which allows variables to be introduced for the first time when they are first actually used. This may seem frustrating, but in fact it helps to structure code, and clear declarations of what type the variable has help to avoid conclusion later on in the code.
Now, NOTE THE SEMICOLON at the end of the line. The semicolon marks the end of the statement.
This is crucial in C. Spaces, carriage returns (new lines) and the like have much less meaning in C than they do in languages such as BASIC. You MUST place a semicolon at the end of a simple statement. (A series of statements enclosed by { }'s is a compound statement; a single statement followed by a semicolon (";") is a simple statement.)
"So if spaces don't matter much, why is your code indented so carefully?"
Because, otherwise, it becomes illegible soup. C does not attempt to tell you to clean up your room, but it does provide you with plenty of breakable toys to trip over. Keeping your code well-organized is the best way to avoid confusing yourself.
The next three lines of code (not including comments) declare additional floating-point variables to be used in the calculation.
Then we encounter the following line:
int interest_periods;
This line declares a variable of type integer. This is because we know there will be a non-fractional number of interest periods that will pass in their entirety during the time over which we are computing interest. Integers typically require less space to store than floating point numbers, and computations involving them are usually faster as well, so when you are certain that a value can be represented by an integer, it is reasonable to declare it as such.
(NOTE: The size of an integer varies, unfortunately, from one environment to the next. The "size" of an integer is the range of values it can represent. In mathematics, we consider an integer to be any real value without a fractional component, but in C, an integer has a maximum and a minimum value. This stems from the fact that each computer has a size of value which it can most comfortably (quickly and efficiently) operate upon, and this size is the size of an integer on that particular model of computer. There are ways to specify larger integers, which will be discussed later.
On an IBM compatible running MSDOS, the value of an integer is usually limited to between -32768 and 32767. On other systems, integers have a much larger range.)
Returning to the code, we next declare two more variables, one a floating point value and the other an integer.
Next, we begin the actual work of the program by assigning initial values to the variables. The line:
initial_balance = 1000.0;

sets the initial balance. Note the use of the = operator, which means assignment ("*make* the left side equal to the right side") in C. There is a separate == operator which is used to *test* for equality. Confusion of the two is the most common error in C programming.
Note also that in assignment statements, as with declarations and all other simple statements, a semicolon marks the end of the statement.
In the following line,
interest = .05; /* Five per cent interest */
Sets the interest rate. Note that a decimal point can begin a number. Note also that a comment can appear on the same line with actual code; this is reasonable for brief comments like the above.
Skipping down several lines, we come to the computation of the number of interest periods that will fall completely within the time over which we are computing interest:
interest_periods = frequency * years;
Note the use of the * operator. The * operator means "multiply" in this situation. It has a second meaning when used in a different context, which will be discussed later.
Skipping a line, we come to the critical part of the program, that in which interest is actually computed:
for (i=0; (i /* Each period, multiply balance by 1.0 plus the annual rate of interest divided by the number of times per year (frequency) with which interest is compounded. */ balance = balance * (1.0 + (interest/frequency)); }
Our goal here is to compound the balance once for each interest period. In order to do so, we take advantage of the "for" keyword to create a "loop."
The "for" keyword is always followed by a set of three expressions, all enclosed in an outer set of parentheses.
The first of the three is an assignment statement, in this case i=0. This statement is performed once, at the very beginning of the loop, ensuring that our count of interest periods begins at zero.
The second is a "conditional expression," the first one we have seen. In C, a conditional expression is actually no different from any other expression (computation). If the expression yields a NON-ZERO value (any value EXCEPT zero), it is regarded as "true." If it yields precisely zero, it is regarded as "false." In the expression used here:
(iThe "<" operator is used to mean "is less than": if the value of i is less than the value of interest_periods, the "<" operator will yield a nonzero value. If i is NOT less than interest_periods, the expression will yield zero. The ">" operator ("is greater than") and the "==" operator ("is equal to"; NOTE THAT THIS IS NOT THE SAME AS A SINGLE "=") are also available, as are ">=" and "<=" ("greater than or equal to" and "less than or equal to"), and "!" (exclamation point), which means "not" (changing a true value to false, or vice versa).
The conditional expression in a for loop is evaluated at the *start* of every pass through the loop. (Note that this means that if the condition is not true on the first pass through the loop, the loop will never execute at all!)
In this case, the condition is used to ensure that we will compute interest once for every interest period; the counter begins at zero, which satisfies the condition, and the process stops when the counter becomes equal to interest_periods.
Since we begin counting from zero, if interest_periods is equal to 4, then on each pass i will be equal to:
0, 1, 2, and finally 3.

Thus we pass through the loop exactly four times.
Now draw your attention to the third and final expression between the parentheses:
i++
(NOTE: the final parenthesis ends this expression, and so a semicolon is not and should not be provided.)
This expression takes advantage of the ++ operator. It appears strange to those used to BASIC or to other languages that do not have such an operator, but in fact it saves a great deal of typing.
Briefly,
i++
is equivalent to:
i = i + 1
and ++ is referred to as the "increment" operator. Note that there is also a "decrement" operator ("--").
Now we are ready to consider what actually takes place *inside* the for loop.
A for loop consists of the for keyword, the three expressions inside the parentheses, and the next statement *following* the parentheses. This following statement can be either a simple statement (one action ending in a semicolon) or a compound statement (a series of statements within { } 's). To avoid losing track of whether the loop is correctly written, it is best to always use a compound statement, even if you only put one simple statement inside it.
Here again is the statement inside the for loop, which is executed once for each pass through the loop:
balance = balance * (1.0 + (interest/frequency));
Note the use of parentheses to group expressions. Their meaning is identical to their usual meaning in algebra. Note also the use of the "/" operator, which means "divide by".
In fact, C follows the standard algebraic order of expressions, with additional rules to cover the unusual operators of C; see Kernighan & Ritchie for a more detailed discussion. I strongly recommend the liberal use of parentheses when you are in any doubt whatsoever as to the order in which operations will take place.
(A note on my arithmetic: the new balance after a given interest period is equal to the current balance, plus the current balance times the annual rate of interest divided by the number of interest periods which will take place in a year. The 1.0 is present to account for the current balance before the multiplication.)
Note that this expression can be made more succinct through the use of the "*=" operator:
balance *= (1.0 + (interest/frequency));
The "*=" operator is similar to the "++" and "--" operators in that it eliminates typing. It means "multiply the left side by the right side and assign this value to the left side." The "+=", "-=" and "/=" operators are also available, and "+=" and "-=" are used particularly often.
A closing "}" ends the compound statement, and thus the entire for loop. When this loop has completed execution, the final balance at the end of the time period is known.
The next statement prints out what has been calculated:
printf("Initial balance: %f Final balance: %f\n", initial_balance, balance);
(Note that this statement is broken across two lines. This is acceptable, and often necessary. Generally the second and subsequent lines should be indented one step further than the first line of such a statement; otherwise considerable confusion results.)

printf() is *not* a keyword, which is to say it does not have a special significance in C, like "for" does. printf() is a function, like main(), or like other functions you will soon write. It is, however, part of the standard C library, and it is made available for use by the #include directive at the beginning of your program.

A function call consists of the name of the function, followed by a set of arguments within parentheses. These arguments will correspond to the arguments that have been declared for that function. (The printf() function is one of a special group of functions which can take a variable number of arguments; users can write such functions also, and this advanced topic will be covered later in the tutorial.)

The first argument is a string. Constant strings are represented in C by a series of characters between double quotes:

"Bob!" "The answer is %d" "Hello, Dolly"

are all legitimate strings. If you need to place a double quote *inside* a string, you can do so by using the special \ character, which is used to "escape" the character following it into a string:

"\"Bah, Humbug,\" he said."

(Here two " characters are escaped into the string.)

In the statement under consideration, the escape sequence \n is used. This sequence places a carriage return (a new line) in the string; otherwise subsequent output would print on the same line. So the string:

"Initial balance: %f Final balance: %f\n"

contains a carriage return at the end.

The two %f sequences in the string are used to tell printf() where to print the floating-point values we will be passing to it.

So the values of initial_balance and balance will be substituted for %f at those points.

(The number of % sequences must match the number of arguments actually passed! To output an integer value, the %d sequence would be used instead of %f.)

Now consider the final statement of the program:

return 0;

This statement returns a value of 0 to the calling function. Since this happens to be the main() function, a value of 0, which is generally accepted to mean the program ran successfully and without incident, is returned to the operating system.

A final } matches the { at the beginning of the main() function, and marks the end of the function.

COMPILING THE PROGRAM

We are now ready to compile and test the program.

Enter the program into your editor if you have not already done so, and save it.

Now instruct your system to compile the program.

If you are using an MSDOS-based compiler with a mouse-driven environment, you will probably want to pull down the "Run" menu and select "Run". The program will compile and, if there are no errors, it will execute. (If the program runs and immediately returns, not allowing you to see the results, enter the appropriate command to look at the output screen. For Borland C products, this command is ALT+F5. I believe the command for Microsoft C products is F9.)

If you are running in a Unix environment, you will need to compile your program by entering the following command at the % prompt:

cc interest.c -o interest

(This instructs the compiler to compile the file interest.c and create an executable program called interest. If you have gcc and not cc, then use the gcc command instead.)
If there are no errors, type:
interest
at the % prompt to run the program.
IF THERE ARE ERRORS:
If the program does not compile successfully, look at the error message that has appeared. Use the editor to look at the line number to which the first error message refers, and search for a typo. (I have tested this program, so it is unlikely to produce errors if it has been entered correctly, though I may be wrong!)
Common mistakes:
-- A missing semicolon at the end of a statement.
-- Forgetting to type the { at the end of the "for" statement's first line.
-- Forgetting the { at the beginning or the } at the end of main().
If the compiler complains that it cannot find include files, then you need to set your include paths properly. Consult your manuals; in the case of a Unix system ask your administrator or a knowledgeable programmer at your site for assistance.
When you believe you have fixed the problem, try again to compile it; when all goes well, run the program to see the result. Correct output should look as follows, with some slight deviation depending on your model of computer and compiler possible:
Initial balance: 1000.000000 Final balance: 1220.895386
(These results came from Borland Turbo C++ version 1.01.)
FURTHER EXPLORATION
Change the interest rate, the number of years, the initial balance, and the frequency with which interest is compounded, compile the program, and run it again. Examine the results. (You may wish to try values that correspond to a savings account of your own.)
Try inserting the following line inside the for loop, on the line just before the line which computes the new balance:
printf("Balance before period %d: %f\n", i, balance);
and compiling and running the program again.
IN THE NEXT CHAPTER
The next chapter will deal with more of the important basic keywords of C, such as the if and while keywords.
IF YOU NEED HELP
For now, contact me directly, after making a serious effort to solve the problem on your own. If I am swamped with requests, I will begin putting together a network of folks willing to provide assistance. If you would be interested in participating in such a network on the *helping* side, by all means contact me so I can begin a Good Samaritan list.