Tutorial: Programming in C
Chapter 3, 6/29/93
NOTE: Previous chapters are available by anonymous ftp from isis.cshl.org in the directory pub/tutorial. Look there first before you ask me to send copies!
Augmenting the First Program: User Input, Monthly Deposits, New C Keywords
The first program we developed calculated compound interest on a savings account. This is rather useful, but the rate of interest, initial balance, and so on were all coded directly into the program. This meant that altering and recompiling the program was necessary in order to examine a different set of conditions.
I will now present a new, interactive version of the program which accepts input from the user and allows several different financial scenarios to be examined in the same session. This version will also allow a monthly deposit to be specified, so you may find it interesting to examine your own finances. (Of course those on the debt side of life can enter a negative number to represent payments made each month and see how long it takes for the initial balance to be reduced to zero!)
The program follows. See the previous installment for instructions on entering, compiling, and executing it, as well as the manuals for your compiler. (Enter everything between the three-dash lines; do not enter the dash lines themselves.)

/* interest.c, by Tom Boutell, 6/27/93. Updated 6/29/93 to support user input. */
/* This program calculates the balance of a savings or loan account after a number of years specified by the user, with an interest rate, monthly payment, initial balance and rate of compounding specified by the user. */
/* 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;
/* Monthly deposit (negative values are permitted) */ float deposit;
/* Flag: when this is set true (nonzero), the user is finished */
int done;
/* User input: analyze again? */
int again;
/* Initially, of course, we are *not* finished. (C does NOT automatically set variables to zero. Making this assumption is a common mistake among new programmers.) */
done = 0;

/* Loop until done. */
while (!done) { /* Fetch starting values from user */ printf("Initial balance: "); scanf("%f", &initial_balance);
printf("Interest rate (example: .05 for 5 percent): "); scanf("%f", &interest);
printf("Number of compoundings per year (12 = monthly, 365 = daily): "); scanf("%f", &frequency);
printf("Monthly deposit (enter negative value for loan payment): "); scanf("%f", &deposit);
printf("Number of years (examples: 1, 5, .5): "); scanf("%f", &years);
/* Actual logic begins here. */
/* 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); printf("Enter 1 for another analysis, or 0 to quit: "); scanf("%d", &again);
if (again) { done = 0; } else { done = 1; } } /* Everything went fine */ return 0; }

Once again I will proceed line by line, explaining the portions of the program that have changed.
First, I added several new variables, which are reasonably well-explained by their associated comments:
float deposit;
specifies a deposit to be made on a monthly basis (a negative value can be entered in order to make a "loan payment" (reduce the balance) each month).
int done;
will be used to determine whether the program should halt.
int again;
will store the user's decision as to whether the program should be run again.
done = 0;
initializes the done flag to false. This line is VERY important! The C programming language does NOT initialize variables to zero. This means that if you do not initialize or otherwise set the value of a variable, its value is undefined; it can be anything, and it can be different from one run to the next.
Some (faulty) compilers do in fact set all variables to zero. Others have an option to do so. DON'T trust this. If you do, you will be unable to move your program to a different compiler or computer without significant changes. This is one of the most common errors in C programming.
The line:
while (!done) {
begins a while loop.
While loops are similar to the for loops discussed in the previous chapter, in that they continue until a condition ceases to be true. But they are simpler in that they consist only of the while keyword, a conditional expression in parentheses, and the following statement, which is executed until the condition ceases to be true. The condition is tested each time the loop returns to the top. If the condition is false at the very beginning, the loop will never execute.

Recall that "!" is the "NOT" operator in C. So "while (!done)" means "while done is not true." Since done is initially zero, which is false in C, "NOT false" yields true, and the statement after the condition executes.
(Recall also that a statement can be either a simple statement ending in a semicolon, such as "i = 0;" , or a compound statement, which is a series of simple statements inside "{" and "}" characters. while loops and for loops themselves are also statements, and can appear inside each other. A compound statement follows the condition in this while loop, and happens to consist of the majority of the program.)
Now consider the following lines, which gather input from the user:
printf("Initial balance: "); scanf("%f", &initial_balance);
The first line is simple; it prints a message to the user, prompting for an initial balance.
The second line calls scanf(), a function which can be used to gather user input. scanf is analogous to printf(), and is used in a similar fashion: a series of % sequences between double-quotes specify the types of values to be input (%f for floating-point values here, just as for printing such values), and the variables to be input follow the quoted string.
"Fine. But what's that & in front of initial_balance for?"
When you call a function in C, the function receives the *values* of the variables you pass. This may seem obvious, but in fact it's an important issue.
In some languages, such as Microsoft QuickBASIC and QBASIC, a function can change the variables that are passed to it, and those values are *changed in the function that called it.*
In C, this isn't the case. When you pass a variable to a function, the function receives the value, but stored in a different place. So while the function is permitted to change the value, that change won't be reflected in the calling function.
"I still don't see what all this has to do with ampersands. "&"'s, even."
Since a function can't change the values passed to it, in order for scanf() to let the user set the variables you pass, it must be given the *location* of the variables, instead of the *value* stored at them. The "&" operator returns the *location* of a variable.
scanf(), in turn, uses another special operator to access the values stored at that location. This operator, and how to use it in your own functions, will be discussed later.
If all this isn't entirely clear just yet, relax; it'll be explained in more detail in a future chapter. But for now, keep the following simplified explanation in mind:
"A function cannot normally change the variables passed to it. In order to 'grant permission' for variables to be changed, they must be passed with the "&" operator preceding them. Functions must be specifically designed to work this way, and scanf() is one of those functions."
Now consider the subsequent lines, which also use printf()/scanf() pairs to prompt the user and gather input.
Note that the sequence "\n" does not appear in the printf() calls. This is in order to keep the user's input on the same line with the prompt.
Now, turn your attention to the following line, placed inside the for loop just before the line that calculates interest (covered in the previous chapter):
balance += deposit * ((1.0/frequency) * 12.0);
As the comment preceding this line explains, the purpose of this statement is to add the monthly deposit to the balance. As the comment also notes, it is not done perfectly (because if, for instance, compounding is daily, then the deposit is actually added in small daily pieces, and so the entire deposit does not begin earning interest from the beginning of the month, as it would in

reality). Feel free to attempt to improve on this. (Hint: you'll want to keep track of how many interest periods have passed and whether enough time has now passed for a deposit to be made.)
Note the use of parentheses to make the order of operations clear. Also note the use of the += operator to simplify the statement.
Now consider the following lines, placed after the final "}" of the for loop:
printf("Enter 1 for another analysis, or 0 to quit: "); scanf("%d", &again);
The first statement prompts the user; the second stores the user's input into the variable "again" by calling the scanf() function. Note the use of the "%d" sequence to read an integer, just as "%d" is used to print an integer in printf(). It is crucial to use the correct sequence for the correct type.
Finally, consider the following lines:
if (again) { done = 0; } else { done = 1; }
Here we use the "if" keyword for the first time.
An "if" statement consists of "if", followed by a conditional expression in parentheses, followed by a statement which is executed only if the conditional expression is true (not zero).
In *addition,* the "else" keyword may appear next, in which case the statement following "else" is executed only if the condition is *not* true (zero).
In this case, we test the variable "again"; if it is not zero, then the user has opted to run another analysis, so we set the value of "done" to zero. If it *is* zero, the user wants to halt the program, so we set the value of "done" to 1.
Immediately after the if statement, the closing "}" of the while loop appears. (Note that the closing "}" is indented by the same number of spaces as the word "while". This helps to prevent confusion as a function grows in length. Your editor may have keys which assist in moving blocks of code left and right; these can be of great help in indenting.) When this closing "}" is encountered, the running program jumps back to the top of the while loop and tests the condition. If "done" is now true (nonzero, 1 in this case), the while loop exits. Once it does exit, the statement "return 0;" returns from the main() function, ending the program.
Now, if you have not already done so, compile and run the program. (See chapter 2, and the manuals of your compiler, for assistance in doing so.) Note that you must press RETURN (or ENTER, depending on your keyboard) after typing each value the program prompts for. Also note that negative values can be entered by prefacing them with a "-", just as in regular algebra.
Again, if you encounter errors, check your code carefully against what I have provided, since it has been tested against a standard compiler. If you're sure there's a problem and can't resolve it yourself, feel free to contact me for assistance, giving a *specific* description of the error message you received. "It gives an error" will *not* help. I also need to know what make of computer you are using and, if possible, what brand and version of compiler.
When you run the program, be sure to try entering various combinations of values. Again, you may wish to try inserting a line which prints out the balance at each interest period, as suggested in chapter 2.

FURTHER EXPLORATION

Try taking on the challenge laid down above to improve the way monthly deposits are added. You may also wish to actually prompt for how often the deposits should be added rather than having them added solely on a monthly basis.

IN THE NEXT CHAPTER

We will write additional functions aside from main() and introduce arrays.

IF YOU NEED HELP

As I mentioned in chapter 2, I am building a network of more experienced C hackers and intermediate students willing to help new programmers. So far I have received surprisingly few questions, but if the number begins to grow, I'll be putting folks in touch with each other.

PLEASE DROP ME A LINE

If you're following the tutorial, please drop me a note by email to let me know. I'd like to get a sense for the number of folks actually following it (the number who asked for it was certainly large!). I'd be interested to know your level of experience with C and with programming in general.