

Introduction to Programming in C

Lab Project Instructions

- | | |
|---------------|---|
| 1. hello | Type in, compile and run a C program |
| 2. fahr1 | Open, compile and trace a C program |
| 3. vars | Work with simple variables |
| 4. box | Draw a solid box composed of '*' characters |
| 5. temps | Calculate average temperature |
| 6. multimod | Compiling and linking a multi-module program |
| 7. quad | Find roots of quadratic equation |
| 8. arsum | Compute sum of elements of an array |
| 9. pointers1 | Work with indirection |
| 10. pointers2 | Discover pointer arithmetic |
| 11. strapp | Convert an array-based function to use pointers |
| 12. beepstest | Process user input |
| 13. atoi | Implement the Standard Library <code>atoi()</code> function |
| 14. ech | Echo command-line arguments |
| 15. calc | Enhance calculator program by adding an operator |
| 16. struct1 | Accessing data in structures |
| 17. pnum | Generate a line-numbered file listing |
| 18. bits | Count the number of 1 and 0 bits in a string |

Lab #1: hello

hello.c Basic source code entry / compilation / execution

From within the Developer Studio: type the following as a new C Source file, save it as `cintro\labs\hello\hello.c`, then compile and execute it:

```
/* hello.c: Our first program */
#include <stdio.h>
int main()
{
    printf("hello, world!\n");
    printf("Testing... \n\n\none\t\ttwo\t\tthree\t");
    printf("four.\n");
    return 0;
}
```

Lab #2: fahr1

fahr.c Open, compile and trace an existing C source file.

Within the Developer Studio, open the source file `fahr.c`, located in the `cintro\labs\fahr1` directory.

Compile it, then enter debug mode (by pressing the F10 key to begin single-stepping through the program).

Trace through the program step by step for several iterations of the loop. Observe the behavior of the local variables watch window as values change from statement to statement.

Lab #3: vars

vars.c Simple variable usage

Define, assign a value to, and print out the value of (via `printf()`) one variable of each of the following types: `int`, `unsigned`, `long` and `double`. Note: The format conversion for signed long integer values is `%ld`.

Assign your `int` variable the value `-1`, and then print it using each of the following format conversions (you can do this in a single `printf()` call passing several copies of the value, or in separate `printf()` calls doing one conversion at a time): `%d` (decimal), `%u` (`unsigned int`), `%f` (floating point). Note: If the program crashes, eliminate the floating point conversion from the program.

Can you explain the results you get by switching the format conversion between `%d` and `%u`?

Lab #4: box

box.c Draw a solid box composed of '*' characters

Write a program named `box.c` that draws a solid square of side length `N` (where `N` is defined at the top of the program as a symbolic constant).

For example, if `N` were defined with
`#define N 5`

then the program's output would be:

```
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
```

Hint: display the box by a combination of

```
printf( "*" );
```

and

```
printf( "\n" );
```

statements.

Lab #5: temps

temps.c Compute average temperature

1. Write a program (**temp1.c**) that reads a week's worth of maximum temperatures from the user, then displays the average of those temperatures as output. Note: the `scanf()` conversion sequence for type `double` is: `%lf`
2. Modify the program to tell how many days had a high temperature above 80 degrees. (**temp2.c**)

Sample run (**temp2.c**):

```
Enter high temp for day 1: 70
Enter high temp for day 2: 90
Enter high temp for day 3: 85
Enter high temp for day 4: 77
Enter high temp for day 5: 88
Enter high temp for day 6: 65
Enter high temp for day 7: 68
The average temperature was: 77.57
There were 3 days having temperatures above 80.
```

3. (Optional): Try to modify the program so that invalid input is handled in a "sane" manner, e.g., have it detect that a non-number was entered, display a message to that effect, and re-prompt for a valid value until the user actually enters one (**temp3.c**).

Note: You'll probably encounter trouble getting your solution to work. If you end up with your program going into an "infinite loop" when incorrect input is entered, don't try to fix it. We'll discuss what's actually happening when we review the lab.

Lab #6: multimod

`ftest.c`, `ff1.c`, `ff2.c`

Multiple-module programs

For this project, you'll be using the command-line compiler for linking the executable programs.

*You may edit your source files using either the Developer Studio or the EDIT editor under DOS. You can also use the Developer Studio as a syntax checker and non-linking compiler by selecting the menu command **Build|Compile** to compile each module.*

PART I:

In a file named `ftest.c`, write a top-level `main()` function that defines an `int` and a `double` variable. Initialize each variable to some arbitrary value.

Assume the existence of a function (not yet written) named `ffunc()` which takes an `int` and a `double` as parameters, and returns a `double`. Call `ffunc()` passing it the required parameters, and display the value it returns on the screen (be careful to use the appropriate format conversion for floating point values!)

DO **NOT** WRITE A PROTOTYPE FOR `ffunc()` AT THIS TIME.

Write two versions of the `ffunc()` function, and place each into its own source file. Name one source file `ff1.c`, and name the other source file `ff2.c`. Make each of your two versions of `ffunc()` have the same input/output type characteristics as assumed by `main()` in `ftest.c`; however, have them perform different calculations so that they return different values for the same input parameters. E.g., you might have `ff1()` return the sum of its two arguments, and `ff2()` their product.

*Despite what you have been told in class, **do not use any prototypes** when writing this first part of the exercise!*

In a DOS box, make sure the command-line compiler has been initialized by running the `VCVARS32` batch file, and then, using the command line compiler, create two executable programs, one for each of the two versions of the `ffunc()` function. Use the following `cl` command lines to create `ff1.exe` and `ff2.exe`:

```
cl ff1.c ftest.c
cl ff2.c ftest.c
```

Each program should compile without any fatal errors at this point. Ignore any warnings about missing prototypes at this time.

Run each program by typing the executable name at the command prompt. Are the results correct? If not, do **not** attempt to debug the programs yet. (*continued*)

PART II:

Create a header file named `ffunc.h` containing the prototype for the `ffunc()` function (remember, the prototype is identical for both versions) and include it (using the `#include` directive) in all three source files you've written. The `#include` directive should read:

```
#include "ffunc.h"
```

where the header file name is enclosed in **double quotes**, *not* in angle brackets as in the case of `<stdio.h>`. Regenerate the two executable files `ff1.exe` and `ff2.exe` as shown above. Does the compiler give you any errors that it didn't before? Specifically, does the compiler tell you anything about the correctness of your calls? If not, that just means you had the calls specified correctly and there is no problem with parameter types.

Run your two executables now. If they do not work correctly, then go ahead and finish debugging them using any means at your disposal.

If they *do* run correctly, but they didn't previously, note that you may not have made any substantial changes to the source code logic; yet, one version worked and the other did not. Can you explain this behavior?

Can you justify the placement of the `#include "ffunc.h"` statements into the `ff1.c` and `ff2.c` files, even though there is nothing else in those files that could currently lead to a conflict?

Lab #7: quad`quad.c`

Practice with expressions and precedence rules

PART I:

Compute the roots x of the quadratic equation:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

for given values of a , b and c as follows:

$$a = 5 \quad b = 15 \quad c = 7$$

Use `doubles` for all your variables in this exercise.

The Standard Library function for calculating the square root of a number is called `sqrt`, and its prototype (located in the standard header file `<math.h>`) is:

```
double sqrt (double x);
```

(continued)

Hint: Coding the expressions for this program becomes much less confusing if you take advantage of the fact that the formula for each of the two roots contains a common subexpression (of considerable complexity).

Note: The values of the roots do *not* work out to nice, round numbers. Check your results against those of other students to determine if you got the correct results.

PART II:

Have `quad.c` prompt the user for the values of `a`, `b` and `c`, rather than having them be hard-wired into the program.

Be sure to perform a “sanity check” on the values before performing calculations that might cause a processor exception (attempting to take the square root of a negative number, division by 0, using uninitialized floating point values, etc). Whenever a problem is detected, inform the user of the problem and terminate the program with the statement

```
return 1;
```

to indicate an abnormal program termination, and make sure to define your `main()` as returning an `int`. Also, have `main()` return 0 on normal termination.

Lab #8: arsum

arsum.c Compute sum of elements of an array

Complete the program below:

```
#include <stdio.h>
int values[7] = { -3, 4, -9, 10, 2, 2, -21 };
void main()
{
    int n;
    n = array_sum(values, 7);
    printf("Sum of elements is: %d\n", n);
}
```

by supplying the `arsum()` function, having prototype as follows:

```
int array_sum(int array[], int array_size);
```

The function takes an `int` array and the size of that array as parameters, and returns an integer value equal to the sum of all elements in the array.

Lab #9: Pointers1

Write a program, `pointers.c` that performs each of the following steps in the order shown:

1. Define four simple variables: a `short int`, a `float`, and an array of two `doubles`, and initialize each to the value of 100.
2. Define four pointers: a `short int` pointer, a `float` pointer, and two `double` pointers, and initialize each pointer variable to point to its corresponding simple variable (i.e., the `short int` pointer pointing to the `short int` variable, the `float` pointer pointing to the `float` variable, each `double` pointer pointing to one of the two array elements)
3. For each simple variable v , print out the values of v and $v+1$ by using the actual names of the variables themselves (note: `short int` uses the same `printf()` format conversion as regular `int`, `%d`)
4. For each simple variable v , print out the values of v and $v+1$ *indirectly* by using the pointers instead of the actual variable names.
5. Write a function with prototype:

```
void setsum(double *sum, double x, double y);
```

This function should accept a `double` by reference as the first argument, and set it to the sum of the 2nd and 3rd arguments (passed by value).

Test the function by calling it to set the value of your first `double` array element to the sum of it and the second `double` array element in your program.

Hint: Don't bother using temporary variables for calculating the expression values; simply write each expression to be printed as another parameter in the `printf()` line. For example, if `foo` were an integer, to print the values of `foo+1` and `foo+2`, you'd write:

```
printf("foo + 1 = %d, foo + 2 = %d\n",  
      foo + 1, foo + 2);
```

Lab #10: Pointers2

Modify your `pointers.c` file from the previous project as follows:

1. For each pointer p , print out the values p , $p+1$ and $p+2$ (that is, the values of the pointers themselves, **not** the values of the objects they may be pointing to.) Use the conversion `%p` for printing all pointer values. This will result in the display of the hexadecimal (base 16) values of the addresses represented by the pointers.
2. Also print out the value of the *difference* of the two double pointers (i.e., $p1 - p2$). Use `%d` to print this value.

When you have successfully compiled and executed your program, examine the values that were printed. Do you notice anything “funny” about the values printed for the results of the expressions involving the pointer variables?

How might the compiler be handling arithmetic on pointers in order to explain the results produced? In other words, what might a practical application be of the strange behavior of “pointer arithmetic”?

Lab #11: strapp

strapp.c Re-write array version of `strapp()` function using pointers

Usage:

```
strapp
```

Re-write the `strapp()` function in `strapp.c` to use pointers instead of arrays. The prototype for this new version of `strapp()` can appear as follows:

```
void strapp(char *str1, char *str2);
```

Or, you can leave the prototype as it stands, using array notation. Since pointers and arrays are equivalent as formal parameters, the two prototypes (and header lines) are completely interchangeable.

Lab #12: beeptest

`beeptest.c` Add user delay control

Modify the program to let the user select the duration of the delay between beeps. Accomplish this as follows:

1. Define a variable to hold the delay value
2. Prompt the user to enter a delay factor
3. Using `gets()` and `atoi()`, read the delay factor from the user and assign it to the variable you created in step 1.
4. Modify the `beepwait()` function to accept a single parameter for the delay value, and use it to control the outermost (`count`) loop.
5. Modify your `main` function to pass the delay value to `beepwait()`.

Optional: After you have `beeptest.c` working as described above, enhance it to check the delay value entered by the user and validate that it falls within the range specified by lower and upper limit values (which you could define as, say, symbolic constants named `LOWER` and `UPPER`). Keep re-prompting until the user enters a number within the valid range.

Lab #13: atoi

`atoi.c` Write the `atoi` function

Write your own `atoi()` function for performing ASCII-to-integer conversion. It should ignore any leading whitespace in the input string, as well as be able to handle a leading '-' character to indicate a negative number. One possible prototype for `atoi()` is:

```
int atoi(char *string);
```

To test your function, you'll find a copy of the single-module version of the `psquare.c` program (from Project #9) in your `labs\atoi\lev1` directory. Start with this file, and it will show you where to place your version of `atoi`.

Hints: Consider using the following library functions:

```
isdigit()    Determines if a character is a decimal digit  
isspace()    Determines if a character is "whitespace"    (continued)
```

If you are experienced in algorithm design, go ahead and write the `atoi()` function now by first designing your own algorithm for it. If you need assistance with the algorithm, here is a basic one for performing ASCII-to-integer conversion, recognizing negation:

1. initialize SUM to 0
2. initialize SIGN to 1.
3. skip any leading spaces in the input string
4. is first non-space character a “-” (for negation)?
 yes: set SIGN to -1, skip over the “-” character
5. set CH to the next character of the input string
6. if CH is NOT a decimal digit:
 All done. return (SUM * SIGN) as result.
7. set SUM = 10 * SUM + (CH - '0')
8. go to step 5.

For less experienced programmers, here is a simpler algorithm that does not recognize negation:

1. initialize SUM to 0
2. skip any leading spaces in the input string
3. set CH to the next character in the input string.
4. if CH is NOT a decimal digit:
 All done. return SUM as result.
5. set SUM = 10 * SUM + (CH - '0')
6. go to step 3.

Lab #14: ech

ech.c Echo command-line arguments

This program should tell how many command line parameters were given, and then display them one per line, numbered, on the standard output. The first line of output should be the name of the program itself.

For example, if the ECH.EXE command were invoked with:

```
ech this is a test
```

then the output should be as follows:

```
Program name is: C:\CLASS\LABS\ECH\LEV1\ECH.EXE
There were 4 parameters specified:
Argument #1 is: "this"
Argument #2 is: "is"
Argument #3 is: "a"
Argument #4 is: "test"
```

Lab #15: calc

`calc.c` Enhance the calculator program

Add an additional operator to the existing `calc.c` program. The name of the operator is `$`, and its function is exponentiation. E.g., an input line containing:

```
3 $ 2
```

would produce the following line of output from the `calc` program:

```
= 9
```

Use the standard library function `pow()`, having its prototype in `<math.h>`, to perform the exponentiation operation.

Lab #16: struct1

`struct1.c` Structure access

As supplied, `struct1.c` reads in and sets up a structure of names and ages from a supplied data file (the function `initdata()` is used to actually perform the input, and it returns a count of the number of items read in.)

Modify `struct1.c` to ask for a name from the user. If the user types a carriage-return, exit the program. Otherwise, tell whether or not the name is present in the data structure, print out the corresponding age if it is, and ask for another name.

Use the following standard library functions:

`gets()` to read in the name from the user

`strcmp()` to compare text strings. **WARNING!** Use the on-line help facility to *carefully* examine the specifications of what the `strcmp()`'s return value actually means!

Your modification will consist of additional code that you will insert following the large comment near the end of the `main` function.

Do not use any `goto` statements in this program!

Enhancement #1: Once you have it working, add a symbolic constant named `SIGNIF` to control how many characters are considered enough to signify a match. Instead of `strcmp()`, use the `strncmp()` library function for this variation.

Lab #17: pnum

pnum.c Generated a line-numbered file listing

Usage:

```
pnum filename
```

Given the name of a text file as argument on the command line, print out the file to the standard output with each line numbered.

Provide the user with a usage message if `pnum` is not invoked correctly, and be sure to report an error if the filename provided cannot be opened for any reason.

Use the `fgets()` standard library function, and note that it includes the terminating newline character as part of the string read in from a file (so you do not need to include a newline after the line is printed.)

NOTE: Don't read the entire file into memory for this task; just process a line at a time

HINT: The main program loop can be concisely implemented in just two lines of code.

To test your implementation, select the Project...Settings “Debug” tab and set the program arguments to, say, “`c:\autoexec.bat`”

Enhancement #1:

The `stdin`, `stdout` and `stderr` streams may be treated as “pre-opened” file pointers; i.e., you can use them in the place of “normal” `FILE *` arguments in all file I/O functions. Given that is the case, modify `pnum` so that invocations of the form:

```
pnum < filename
```

work by processing the standard input rather than a named file. Note that the command:

```
pnum
```

would be perfectly legal using this new form; it would read the input from the keyboard.

Lab #18: bits

bits.c Count bits

The skeletal source file `bits.c` sets up the environment for an exercise in bit-level operations. The purpose of the program is to count up the total number of 1 bits and 0 bits among all characters in a supplied literal string. The variables you will use are all defined for you, and the code to display the results is also pre-written.

Your task is to code the appropriate loops and expressions to compute the total number of 1 and 0 bits in the string.