Programming Language Specific Help

---

**c**

How to write C solutions.

**gdb**

How to debug infinite loops and crashes in C and C++ programs.
C Language Program Structure

Typical program structure including end of file detection, finding symbols such as ‘*’ that end a string of numbers, and debugging is:

```c
/* Standard Libraries. */
#include <stdlib.h> /* qsort */
#include <stdio.h>  /* scanf, fgets, printf */

/* String and character computation. */
#include <string.h> /* strcmp, strcpy, ... */
#include <ctype.h>  /* isalpha, islower, ... */

/* Math and Number Limits. */
#include <math.h>   /* M_PI, atan2, ... */
#include <limits.h> /* INT_MIN, INT_MAX, ... */
#include <float.h>  /* DBL_MIN, DBL_MAX */

/* Debugging. */
#include <assert.h> /* assert */

int debug = 0;
#define dprintf if ( debug ) printf
```

```c
int main ( int argc, char * argv[] )
{
    /* argc == number of program arguments + 1. */
    debug = ( argc > 1 );
    . . . .
    while ( fgets ( line, sizeof ( line ), stdin ) )
    {
        /* Skip `*' and preceding whitespace. */
        scanf ( " %" );
        . . . .
    }
}
```
After reading test case data with scanf, input will be just before the line feed at the end of the test case data. This must be skipped or else it will be erroneously read as an empty next test case name. Note: you CANNOT use scanf ("\n") to do this!

fgets ( line, sizeof ( line ), stdin );
assert ( line[0] == '\n' );
/* The line read here consists of just a line feed and is discarded. */

/* Compute. */
dprintf ( ... ); /* Print if debug. .......
/* Output. */
    printf ( ... );
}
/* Return 0 from ‘main’ to tell system there is no error. */
return 0;

Occasionally one has a conflict with some name defined in the ‘#include’s. E.g., you may have trouble naming a global variable ‘time’. To fix this, change the name of your variable. A simple way of doing this is to put the following right after the ‘#include’s:

#define time Time

Input
-----
Input is read from the standard input; you MUST NOT open any file for input.

You may assume that input is correctly formatted, except for the rare problem where you are told to produce special output if the input contains a formatting error. Do not waste time checking for input errors when you do not have to.

Test cases begin with a test case name line that is read by the first ‘fgets’. ‘fgets’ returns NULL on an end of file, which is interpreted as ‘false’, so the outer loop above processes one test case at a time. As soon as the test case name is successfully read into ‘line’, it is printed by ‘printf ( "%s", line )’. All test case name lines are 80 characters or less, so ‘line’ has enough room for 80 characters plus the line ending ‘\n’ and then the string ending ‘\0’, both of which are supplied by ‘fgets’.

After reading and printing the test case name line, the rest of the test case data is read with ‘scanf’. This data may consist of numbers, perhaps with a few characters like ‘*’ thrown in to indicate the end of a sequence of numbers. The lines containing numbers may be very long, so they should be read one number at a time with ‘scanf’, and NOT read in first with ‘fgets’. A ‘*’ may be detected by looking at the return value of ‘scanf’, which is the number of numbers actually read. If you are expecting 4 integers or a ‘*’, use ‘scanf ( "%d%d%d%d", ... )’ which returns 4 if it read 4 integers and 0 if it encountered a ‘*’. Then in the latter case you must skip the ‘*’, which can be done by ‘scanf ( "*" )’ which skips first the whitespace between the last number read and the input ‘*’, and then skips the input ‘*’.
See the ‘summer’ demonstration problem solution which reads numbers.

Floating point numbers can be read by ‘scanf’ by using %lf instead of %d, and non-numbers can be read by using %s or %c – see synopsis below. Sometimes a problem will specifically ask you to read lines to be processed as character strings, in which case use ‘fgets’ and be sure your line buffer is long enough.

For some problems there are no test case name lines but instead each test case has one input line containing text and one output line containing text. For these the test case input line is read the same way as the test case name line is read above.

See the ‘reverser’ demonstration problem solution which reads text lines.

Sometimes it is necessary to read input one character at a time. ‘int c = getchar();’ does this, returning the character read if there is one, or the special constant EOF if at the end-of-file. Notice that c is an ‘int’ and not a ‘char’; this is because EOF is usually defined to be -1. If you have gotten the character c this way, ‘ungetc ( c, stdin )’ backs up one character in the input stream, so the next character to be read will be the character c again. However you CANNOT reliably call ungetc several times in a row to back over more than one character at a time.

See the ‘vcalc’ demonstration problem solution which reads characters one at a time.

Output
--------

Output is written to the standard output; you MUST NOT open any file for output; you MUST NOT write to the standard error output.

In general, to be correct your program must produce EXACTLY the one and only correct sequence of output characters. The main exception is that when floating point numbers are output with a given number of decimal places, you are permitted to output numbers which differ from other correct output by one unit in the last decimal place. You must use the correct upper or lower case and use only a single space character as a separator unless instructed to line things up in columns.

Output can be written with printf. Some formats that may be useful are "%.3f" to print a double with exactly 3 decimal places in as few columns as possible, "%10.3f" to print the double right adjusted in 10 columns, "%10s" to print a string right adjusted in 10 columns, and "%10s" to print a string LEFT adjusted in 10 columns. Here the numbers 3 and 10 are merely representative, and can be replaced by any other non-zero positive integers.

See the ‘summer’ demonstration problem solution which outputs floating point ‘double’s.

Debugging
---------

When your program is executed by the judge, it will NOT be passed ANY arguments. A standard debugging technique is to output extra debugging information if and only if your program is passed an argument.
The above program defines ‘debug’ to be true if and only if the program is called with one or more arguments, and defines dprintf to do what printf does if and only if ‘debug’ is true. Thus dprintf can be used to print debugging information.

Debugging is best done with information printed by ‘dprintf’, and not with a debugger like ‘gdb’. The exception is debugging programs that crash or go into an infinite loop, which is best done with a debugger like ‘gdb’. See ‘help gdb’ for how to use ‘gdb’ on programs that crash or go into an infinite loop.

It is also a good idea to use ‘assert’ statements to check that assumptions you have made are valid during actual program execution. For example, if you are certain that scanf should read 3 integers, then instead of writing

```c
    scanf ( "%d%d%d", ... );
```

it might be better to write

```c
    assert ( scanf ( "%d%d%d", ... ) == 3 );
```

Similarly, you could replace ‘scanf ( " * " )’ in the above code by

```c
    assert ( scanf ( " %c", line[0] ) == 1 );
    assert ( line[0] == ' * ' );
```

See the ‘summer’ demonstration problem solution which uses dprintf and assert.

### Function and Macro Synopsis

--- ---- ----- --------

You can get full documentation of the functions mentioned above plus other useful information by using the UNIX commands:

- `man scanf`
- `man fgets`
- `man getchar`
- `man 3 printf`
- `man assert`
- `man math.h`
- `man limits.h`
- `man float.h`

The following is an extract from ‘man’ pages of details needed to use the functions mentioned above, and also a few numeric constants. Here we assume you already know something about how to use ‘scanf’ and ‘printf’, so we do not have to explain them from scratch.

Note that an ‘int’ is guaranteed to be at least 32 bits and a ‘long long int’ is guaranteed to be at least 64 bits. Floating point computations should always be done using ‘double’s, and NOT ‘float’s, to avoid having too little precision.

```c
    int scanf ( const char * format, ... )
```

Format can contain the following directives:

- `%d` First skips whitespace. Then if this is followed by non-whitespace with the format of an ‘int’, reads the ‘int’.
- `%lld` Ditto but for ‘long long int’ instead of ‘int’.
%f  Ditto but for ‘double’ instead of ‘int’.
%e  Ditto but for a string of non-whitespace characters, instead of ‘int’.
%c  Reads a single character. Does NOT skip preceding whitespace.

Whitespace  Skips any amount of whitespace, including none. Line feeds are whitespace in this sense.

Other  Skips just the given character. E.g., if ‘‘’ is in the format, then ‘*’ must be next in the input if the input is to move forward. If you want to skip whitespace followed by ‘*’, you MUST put "**" and NOT just "*" in the format.

Scanf returns the number of items successfully read by `%...` format directives. Thus if you are expecting either a float or a ‘‘’, ‘scanf ( "%%le", ...)’ will return 1 if a number was found and read, or 0 if a ‘‘’ was found and NOT read. Note that if it returns 0 it will have skipped the whitespace before the ‘‘’, so the ‘*’ will be next.

char * fgets ( char * buffer, sizeof ( buffer ), stdin )

Read characters from the standard input into the buffer until a line feed is read and stored, and then store a ‘0’ in the buffer. If there are NO characters to be read because we are at the end of file, return NULL, which can be interpreted as ‘false’; otherwise return the address of the buffer, which can be interpreted as ‘true’.

Note: ‘fgets’ as it is used above is similar to ‘gets ( buffer )’, but the ‘gets’ function has been ‘deprecated’, meaning that implementations need not and eventually will not support it.

int c = getchar()
ungetc ( c, stdin )
getchar() reads one character from the standard input stream, or the special constant EOF (typically defined to be -1) if at end-of-file. If c is the character just read by getchar(), then the call ungetc ( c, stdin ) backspaces up one character in the standard input, so c will be re-read by the next getchar(). Ungetc CANNOT reliably be called several times in a row to back over more than one character at a time.

int printf ( const char * format, ... )

Format can contain the following directives, in which W and P stand for sequences of decimal digits, where W is used for ‘widths’ and P for ‘precisions’:

% d  Outputs an ‘int’ as a decimal integer with no spaces before or after.
% W d  Outputs an ‘int’ as a decimal integer right adjusted in W columns.
% ll d  Outputs a ‘long long int’ as a decimal integer with no spaces before or after.
% W ll d  Outputs a ‘long long int’ as a decimal integer right adjusted in W columns.
% f  Outputs a ‘double’ as a floating point number with 6 decimal places and no spaces before or after.
\%Pf \hspace{2em} \textit{Ditto but with P decimal places instead of 6 decimal places.}

\%Wf \hspace{2em} \textit{Outputs a ‘double’ as a floating point number with 6 decimal places right adjusted in W columns.}

\%W.Pf \hspace{2em} \textit{Ditto but with P decimal places instead of 6 decimal places.}

\%s \hspace{2em} \textit{Outputs a ‘const char ’ or ‘char ’ string with no spaces before or after.}

\%Ws \hspace{2em} \textit{Ditto but outputs the string RIGHT adjusted in W columns.}

\%-Ws \hspace{2em} \textit{Ditto but outputs the string LEFT adjusted in W columns.}

\%c \hspace{2em} \textit{Outputs a ‘char’ as an ASCII character with no spaces before or after.}

\texttt{INT\_MAX, INT\_MIN, LLONG\_MAX, LLONG\_MIN, DBL\_MAX, DBL\_MIN}

\hspace{2em} \textit{These are the maximum and minimum numbers that can be stored respectively in an ‘int’, ‘long long int’, or ‘double’. The integer values are from ‘limits.h’ and the floating point values from ‘float.h’.

\texttt{M\_PI, M\_E}

\hspace{2em} \textit{The mathematical constants PI and e from math.h.}

File: \hspace{2em} \texttt{c}

Author: \hspace{2em} \texttt{Bob Walton <walton@seas.harvard.edu>}

Date: \hspace{2em} \textit{See top of file.}

The authors have placed this file in the public domain; they make no warranty and accept no liability for this file.
Debugging with GDB       Wed Jan 30 07:32:03 EST 2013

Introduction
-----------

It is recommended that a debugger like GDB not be used unless your program crashes or goes into an infinite loop. In all other cases it is better to put print statements (such as printf or dout – see ‘help c’ or ‘help c++’) into your code to print just exactly what you need to see.

So in this help file we assume you are debugging a C or C++ program that crashes or loops infinitely.

Running the Program
---------

Assume the program you are debugging is named PPPP and its input file is PPPP.in. Then

% gdb PPPP
(gdb) run <PPPP.in

If the program is going to crash, it will do so. Otherwise if the program loops infinitely, wait until it is so looping, and type control-C to stop the program.

Examining a Crashed or Stopped Program
---------

Once stopped execute

    (gdb) back

to see the backtrace of functions that have been called. Each function call is referred to as a ‘frame’. The frames are numbered, 0, 1, 2, ..., with 0 being the frame that was executing at the time of the crash or control-C, and the highest number frame being the ‘main()’ function.

You can see what the current instruction and variables of frame N are by executing

    (gdb) frame N
    (gdb) p VARIABLE

Here ‘p’ stands for ‘print’ and variables that can be printed in a frame are the local variables of the frame plus any global variables visible to the code that executes the frame.

When you execute ‘frame N’ the line at which frame N stopped executing will be printed, along with its line number. You can use an editor to look at the source code, or you can use the command

    (gdb) list M,N

to list lines, where M,N are the first, last line numbers to be listed.
If your program is in an infinite loop, it may be helpful to continue the program. The following may be useful:

Continue: (gdb) c

Go to next statement in current frame: (gdb) n

Go to next statement in current frame or frame called by current frame: (gdb) s

Set a breakpoint at line N: (gdb) b N

Continue to next breakpoint: (gdb) c

List breakpoints: (gdb) i b

Delete breakpoint NUMBER K: (gdb) d K

Output documentation for command C: (gdb) h C

Find commands concerning word W: (gdb) apropos W

File: gdb
Author: Bob Walton <walton@seas.harvard.edu>
Date: See top of file.

The authors have placed this file in the public domain; they make no warranty and accept no liability for this file.