How to Debug Programs

Table of Contents

This document explains how to write computer programs that *work* and that are understandable to other intelligent beings! This two attributes are *not* independent! In general, programs that other programmers can not understand do not work very well. (Not to mention the fact that they are maintenance nightmares!)

Writing structured programs (structured code *and* data!) helps greatly in debugging the code. Here is a quick review of some of the features of a structured program.

1. *Lots* of well-defined functions!
2. Using structured loop constructs (*i.e.*, `while` and `for`) instead of `goto`.
3. Using variables that have *one* purpose and *meaningful names*.
4. Using structured data types to represent complex data.
5. Using the ADT (*Abstract Data Type*) or OOP (*Object-Oriented Programming*) paradigm of programming

1. How to Start

The most common types of mistakes when programming are:

1. Programming without *thinking*
2. Writing code in an unstructured manner

Let's take these in order

1.1 Thinking about Programming

When a real programmer (or programming team) is given a problem to solve, they do *not* immediately sit down at a terminal and start typing in code! They first design the program by thinking about the numerous ways the problem's solution may be found.

One of the biggest myths of programming is that: *The sooner I start coding the sooner a working program will be produced.* **This is NOT true!!** A program that is planned before coding will become a working program before an unplanned program. Yes, an unplanned program will be typed in and maybe compiled faster, but these are just the *first* steps of creating a *working* program! Next comes the debugging stage. This is where the benefits of a planned program will appear. In the vast majority of the time, a planned program will have less bugs than an unplanned program. In addition, planned programs are generally more structured than unplanned programs. Thus, finding the bugs will be easier in the planned program.

So how does one *design* a program before coding? There are several different techniques. One of the most common is called *top-down design*. Here an outline of the program is first created. (Essentially, one first looks at the general form of the `main()` function and then recursively works down to the lowest level functions.) There are many references on how to write programs in this manner.

Top-down design divides the program into sub-tasks. Each sub-task is a smaller problem that must be solved. Problems are solved by using an algorithm. Functions (and data) in the program implement the algorithm. Before writing the actual code, take a very small problem and trace by hand how the your chosen algorithm would solve it. This serves several purposes:

1. It checks out the algorithm to see if will actually work on the given problem. (If it does not work, you can immediately start looking for another algorithm. Note that if you had immediately starting coding you would probably not discover the algorithm would not work until many lines of code had been entered!)
2. Makes sure that *you* understand how the algorithm actually works! (If you can not trace the algorithm by hand, you will *not* be able to write a program to do it!)
3. Gives you the detail workings of a short, simple run of the algorithm that can be used later when debugging the code.

Only when you are confident that you understand how the entire program will look should you start typing in code.

1.2 Structured Programming

When a program is structured, it is divided into sub-units that may be tested separately. This is a *great* advantage when debugging! Code in a sub-unit may be debugged separately from the rest of the program code. I find it useful to debug each sub-unit as it is written. If no debugging is performed until the entire program is written, debugging is much harder. The entire source of the program must be searched for bugs. If sub-units are debugged separately, the source code that must be searched for bugs is much smaller! Storing the sub-units of a program into separate source files can make it easier to debug them separately.

The ADT and OOP paradigms also divide programs into sub-units. Often with these methods, the sub-units are even more independent than with normal structured code. This has two main advantages:

1. Sub-units are even easier to debug separately.
2. Sub-units can often be *reused* in other programs. (Thus, a new program can use a previously debugged sub-unit from an earlier program!)

Sub-units are generally debugged separately by writing a small *driver program*. Driver programs set up data for the sub-task that the sub-unit is supposed to solve, calls the sub-unit to perform his sub-task, and then displays the results so that they can be checked.

Of course, all the following debugging methods can be used to debug a sub-unit, just as they can be used to debug the entire program. Again, the advantage of sub-units are that they are only part of the program and so are *easier* to debug than the entire program at once!

2. Compiling Programs

The first step after typing in a program (or just a sub-unit of a program) is to compile the program. The compiling process converts the source code file you typed in into machine language and is stored in an *object file*. This is known as *compiling*. With most systems, the object file is automatically *linked* with the system libraries. These libraries contain the code for the functions that are part of the languages library. (For example, the C libraries contain the code for the `printf()` function.)

2.1 Compiler Errors

Every language has syntax rules. These rules determine which statements are legal in the language and which are not. Compiler programs are designed to enforce these rules. When a rule is broken, the compiler prints an error message and an object file is not created. Most compilers will continue scanning the source file after an error and report other errors it finds. However, once an error has been found, the compiler made be confused by later perfectly legal statements and report them as errors. This brings us to the first rule of compiler errors:

First Rule of Compiler Errors

The first listed compiler error is always a true error; however, later errors may not be true errors.

Succeeding errors may disappear when the first error is removed. So, if later error messages are puzzling, ignore them. Fix the errors that you are sure are errors and re-compile. The puzzling errors may magically disappear when the other true errors are removed.

If you have many errors, the first error may scroll off the screen. One solution to this problem is to save the errors into a file using redirection. One problem is that errors are written to `stderr` not `stdout` which the > redirection operator uses. To redirect output to `stderr` use the 2> operator. Here's an example:

```
$cc x.c 2>errors
$more errors
```

The compiler is just a program that other humans created. Often the error messages it displays are confusing (or just plain *wrong*!). Do not assume that the line that an error message refers to is always the line where the true error actually resides. The compiler scans source files from the top sequentially to the bottom. Sometimes an error is not detected by the compiler until many lines below where the actual error is. Often the compiler is too stupid to realize this and refers to the line in the source file where it realized something is wrong. The true error is earlier in the code. This brings us to the second rule of compiler errors:

Second Rule of Compiler Errors

> *A compiler error may be caused by any source code line <u>above</u> the line referred to by the compiler; however, it can <u>not</u> be caused by a line below.*

In C (and C++), do not forget that the `#include` preprocessor statement inserts the code of a header file into the source file. An error in the header file, may cause a compiler error referencing a line in the main source file. Most systems allow the preprocessed code (that the C compiler actually compiles!) to be stored in a file. This allows you to see exactly what is being compiled. This file will also show how each C macro was expanded. This can be *very* helpful to discover the cause of normally very hard to find errors.

A useful technique for finding the cause of puzzling compiler errors is to delete (or comment out) preceding sections of code until the error disappears. When the error disappears, the last section removed must have caused the error.

The compiler can also display *warnings*. A warning is not a syntax error; however, it *may* be a logical error in the program. It marks a statement in your program that is legal, but is suspicious. You should treat warnings as errors unless you understand why the warning was generated. Often compilers can be set to different warning levels. It is to your advantage to set this level as high as possible, to have the compiler give as many warnings as possible. Look at these warnings *very* carefully!.

Remember that just because a program compiles with no errors or warnings does *not* mean that the program is correct! It only means that every line of the program is syntactically correct. That is, the compiler understands what each statement says to do. The program may still have many *logical errors*! An English paper may be grammatically correct (*i.e.*, have nouns, verbs, *etc.* in the correct places), but be gibberish.

2.2 Linker Errors

The *linker* is a program that links object files (which contain the compiled machine code from a single source file) and libraries (which are files that are collections of object files) together to create an executable program. The linker matches up functions and global variables used in object files to their definitions in other object files. The linker uses the *name* (often the term *symbol* is used) of the function or global variable to perform the match.

The most common type of linker error is an unresolved symbol or name. This error occurs when a function or global variable is used, but the linker can not find a match for the name. For example, on an IBM AIX system, the error message looks like this:

```
0706-317 ERROR: Unresolved or undefined symbols detected:
```

```
                Symbols in error (followed by references) are
                dumped to the load map.
                The -bloadmap:<filename> option will create a load map.
.fun
```

This message means that a function (or global variable) named `fun` (ignore the period) was referenced in the program, but never defined. There are two common causes of these errors:

Misspelling the name of the function
: In the example, above there was a function named `func`. This is *not* a compiler error. Code in one source file can use functions defined in another. The compiler assumes that any function referenced, but not defined in the file that references it, will be defined in another file and linked. It is only at the link stage that this assumption can be checked. (Note that C++ compilers will usually generate compiler errors for this, since C++ requires prototypes for *all* referenced functions!)

The correct libraries or object files where not linked
: The linker must know what libraries and object files are needed to form the executable program. The standard C libraries are automatically linked. UNIX systems, like the AIX system, do *not* automatically link in the standard C math library! To link in the math library on the AIX system, use the `-lm` flag on the compile command. For example, to compile a C program that uses `sqrt`, type:

```
cc prog.c -lm
```

Remember that the `#include` statement only inserts text into source files. It is a common *myth* that it also links in the appropriate library! The linker *never* sees this statement!

There are also bugs related to the linker. One difficult bug to uncover occurs when there are two definitions of a function or global variable. The linker will pick the first definition it finds and ignores the other. Some linkers will display a warning message when this occurs (The AIX linker does not!)

Another bug related to linking occurs when a function is called with the wrong arguments. The linker only looks at the name of the function when matching. It does no argument checking. Here's an example:

File: x.c

```
int f( int x, int y)
{
  return x + y;
}
```

File: y.c

```
int main()
{
  int s = f(3);
  return 0;
}
```

These types of bugs can be prevented by using prototypes. For example, if the prototype:

```
int f( int, int);
```

is added to *y.c* the compiler will catch this error. Actually, the best idea is to put the prototype in a header file and include it in both *x.c* and *y.c*. Why use a header file? So that there is only one instance of the prototype that all files use. If a separate instance is typed into each source file, there is no guarantee that each instance is the same. If there is only one instance, it can not be inconsistent with itself! Why include it in *x.c* (the file the function is defined in)? So that the compiler can check the prototype and ensure that it is consistent with the function's definition. (Additional note: C++ uses a technique called *name mangling* to catch these type of errors.)

3. Runtime Errors

A runtime error occurs when the program is running and usually results in the program aborting. On a UNIX/Linux system, an aborting program creates a *coredump*. A coredump is a binary file named `core` that contains information about the state of program when it aborted. Debuggers like *gdb* and *dbx* can read this file and tell you useful information about what the program was doing when it aborted. There are several types of runtime errors:

Illegal memory access
: This is probably the most common type of error. Under UNIX/Linux, the program will coredump with the message `Segmentation fault(coredump)`.
: Using Win95 or NT, programs will also abort. However, traditional DOS does not check for illegal memory accesses; the program continues running, but the results are unpredictable. The DOS Borland/Turbo C/C++ compilers will check for data written to the NULL address. However, the error message `NULL pointer assignment` is not displayed until the program terminates.

Division by zero
: All operating systems detect this error and abort the program.

4. Debugging Tools

Many methods of debugging a program compare the program's behavior with the correct behavior in great detail. Usually the normal output of the program does not show the detail needed. Debugging tools allow you to examine the behavior of the in more detail.

4.1 The `assert` Macro

The `assert` macro is a quick and easy way to put debugging tests into a C/C++ program. To use this macro, you must include the `assert.h` header file near the top of your source file. Using `assert` is simple. The format is:

```
assert(boolean (or int) expression);
```

If the *boolean expression* evaluates to true (*i.e.*, not zero), the `assert` does nothing. However, if it evaluates to false (zero), `assert` prints an error message and aborts the program. As an example, consider the following `assert`:

```
assert( x != 0 );
```

If *x* is zero, the following will be displayed:

```
Assertion failed: x != 0, file err.c, line 6
Abnormal program termination
```

and the program will abort. Notice that the actual assertion, the name of the file and the line number in the file are displayed.

The `assert` macro is very useful for putting sanity checks into programs. These are conditions that should always be true if the program is working correctly. It should not be used for user error checking (such as when the file a user requested to read does not exist). Normal `if` statements should be used for these runtime errors.

Of course, in a commercial program, an assertion failure is not particular helpful to an end user. Also, checking assertions will make the program run at least a little slower than without them. Fortunately, it is easy to disable the `assert` macro without even removing it. If the macro `NDEBUG` is defined (above the statement that includes `assert.h`!), the `assert` macro does absolutely nothing. If the assertions need to be enabled later, just remove the line that defines `NDEBUG`. (If this technique is used, be sure that the `assert` statements do not execute code needed for the program to run correctly. If `NDEBUG` is defined, the code would *not* be run!)

4.2 Print Statements

This time honored method of debugging involves inserting debugging print statements liberally in your program. The print statements should be designed to show both what code the program is executing and what values critical variables have.

4.3 Debuggers

The previous method of debugging by adding print statements has two disadvantages:

1. When new print statements are added, program must be recompiled.
2. Information output is fixed and can not be changed as program is running.

Source-level debuggers provide a much easier way to trace the execution of programs. They allow one to:

1. Look at the value of any variable as the program is running.
2. Pause execution when program reaches any desired statement. (This position in the program is called a *breakpoint*).
3. Single step statement by statement through a program.

I *strongly* recommend that you learn to use the debugger for whatever system you program on. A debugger can save *lots* of time when debugging your program!

4.4 Lint

The `lint` program checks C programs for a common list of bugs. It scans your C source code and prints out a list of possible problems. Be warned that `lint` is *very* picky! For example, the line:

```
    printf("Hello, World ");
```

will produce a warning message because `printf` returns an integer value that is not stored. The return value of `printf` is often ignored, but `lint` still produces an warning. There are several ways to make `lint` happy with this statement, one is:

```
    (void) printf("Hello, World ");
```

This says to ignore the return value.

4.5 Walk through

A *walk through* is a process of hand checking the logic of a program. The programmer sits down with someone else (best if another programmer, but anybody will do) and walks through the program for an example case. Often it is the programmer himself who finds the bug in the process of explaining how the program is supposed to work and carefully looking at his code. However, it is easy for the programmer to *"know"* what the program *should be doing* and remain blind to what the program is *actually doing*.

Students need to be very careful using this approach with other students. Two students in the same class should not walk through a program together.

5. General Tips

Here are some general tips for debugging programs.

5.1 Finding Bugs

Before bugs are removed they must be discovered!

- Aggressively test programs!
- Start with *small* problems that can be easily checked by hand. (You should already have one of these worked out from the planning stage!)
- Test every feature of the program at least once! And is once really enough? Test features in different ways if possible.
- Do not forget to test trivial problems.
- Do not make invalid assumptions about input data.

5.2 Determining the Causes of Bugs

A bug can only be caused by the code in the program that has already executed. Be sure you do not waste time searching through code that has not run yet. A debugger or print statements can be used to determine which code has executed and which has not.

Do not fix bugs by mindlessly changing code until it seems to work. You need to figure out why one statement does work and another does not. You should have a good reason for every line of code. *"It doesn't work without this line"* is not a good reason!

If you are using C/C++, you might find my Common C Errors Page helpful.