Debugging

1 How To Debug

What is Debugging?

DEBUGGING is what you do to fix your program when it doesn't work. The term originated with the first tube computers. Sadly DEBUGGING has always been a part of computer programming. Modern programming techniques try to reduce or eliminate the need for extensive debugging of code, by preparing logically perfect programs.

The Best Cure is Prevention

When you come to debug your program, you already know it doesn't work. If you knew before that it didn't work then you could correct it before you began. If you *understand* your program you will be less likely to make mistakes and be better at correcting them. Write the program so you can understand it. Use comments and simple code.

Is Clarity More Important Than Correctness?

⇒ If your program is clear you can make it correct.
Can you be sure your program is correct if it isn't clear?

There's never time to do it right, but there's always time to do it again.

- Plan ahead
  Plan to have your program complete three days early. You can use the extra time to debug or carefully test your program.
- Plan your program before you type it
  - It will be less likely to be wrong
  - If it is wrong you will know where to look
- Program in modules (*encapsulation!*)
  - Write programs using stubs
  - Comment as you write
    - Use comments as an outline
    - Tidy up the comments before handing-in
  - Use PRE and POST conditions
    - What does each module do?
      - What must be true before the function is called?
      - What will be true after the function runs?
        e.g., what value will changed variables have?
        e.g., what will show up on the screen?
    - This will be easier to check

To Solve a Problem You Must Understand It

Find the code that generates the problem

- Print values of variables to trace program execution
  - Always print to `stderr`
  - Always print the procedure name (see example)
- Include debugging code

```
#ifdef DEBUG
  fprintf(stderr, "top of main loop k= %d\n",k);
#endif
```

  - This code will only be run if the `.c` file has a `#define DEBUG` *before* the `#ifdef`
  - Easy to turn on and off
  - Can also use DEBUG levels (e.g., `#if DEBUG > 1`)

2 Examples

2.1 A Short Example

main.h

```
#include <stdio.h>
  int fclose(FILE *stream);
  int fprintf(FILE *stream, const char *format,...);

typedef char *             string;
typedef short unsigned int bool;

#define TRUE  (1)
#define FALSE (0)

/* util.c prototypes */
void usage(const string name);
string basename(string command);
bool close_file(const string name,  const string proc, \
                const string fname, FILE * handle);
FILE * open_file(const string name,  const string proc, \
                 const string fname, const string mode);
```

util.c

```
#include "main.h"

static int open_files = 0; /* number of files currently open */

  ⋮

bool close_file(const string progname, /* name of the program       */
                const string proc,     /* name of function calling  */
                const string fname,    /* filename of file to close */
                      FILE * handle)   /* the file to be closed     */
{
   if (fclose(handle) == EOF) {
      fprintf(stderr, "%s[%s()] Error: cannot close file '%s'\n",
              name, proc, fname);
      return(FALSE);
   } else {
      open_files--;
#     if DEBUG > 1
         fprintf(stderr,"%s() closed %s\n", proc, fname);
#     endif
      return(TRUE);
   }
} /* close_file() */
```

2.2 A Longer Example

```
int read_graph(FILE  * source,  /* file to read from or stdin*/
               GRAPH * graph,   /* the graph that is being updated */
               char  * progname /* name of the program */

/*
PURPOSE: Reads all directed edges from a file (or stdin)into  a graph

Pre:  `source' points to an file open for reading,
      `graph' points to memory newly allocated for a GRAPH data structure
POST: `graph' has been initialized to contain all the edges in the file
      The number of edges read in has been returned.
    OR
      An error message has been sent to stderr and a negative number has
      been returned.

NOTES: Every line of the file should have this format:
             from to: weight
       where `from' and `to' are vertex numbers and `weight' is
       the edge weight.
*/

{
   char proc[]  = "read_nums"; /* name of function for error messages */
   int  linenum = 1;
   int  count   = 0;           /* count of numbers read so far        */
   char line[MAX_LINE_LEN];    /* buffer to read line from file into  */
   int  from,                  /* vertex number which the edge leaves */
        to,                    /* vertex number that the edge goes to */
        weight;                /* the weight of the edge              */

   if (init_graph(graph) == OKAY) {
      while((count < MAX_EDGES) && !ferror(source) && !feof(source)) {
         if (fgets(line, MAX_LINE_LEN, source) != NULL) {
            linenum++;
            if (sscanf(line, "%d %d: %d\n", &from, &to, &weight) == 3) {
               insert_edge(graph, from, to, weight);
               count++;
```

```
         } else {
            (void)fprintf(stderr,
                  "%s[%s()] Error: sscanf() couldn't parse line #%d\n",
                  progname, proc, linenum);
            (void)fprintf(stderr, "line = \"%s\"\n", line);
            return(-2);
         }
      } else { /* error from fgets() */
         if (ferror(source) || !feof(source)) {
            (void)fprintf(stderr,
                  "%s[%s()] Error: fgets() couldn't read line #%d\n",
                  progname, proc, linenum);
              return(-3);
         }
      }
   } /* while */
   if ((count == MAX_EDGES) || !feof(source)) {
      (void)fprintf(stderr,
            "%s[%s()] Warning: using only first %d edges for graph\n",
            progname, proc, MAX_EDGES);
   }
   return(count);
} else {
   (void)fprintf(stderr,
         "%s[%s()] Error: error from init_graph\n", progname, proc);
   return(-1);
}
```

3 Debuggers

3.1 Debuggers In General

- never depend on a debugger to find errors
- you can waste a lot of time stepping through
- many debuggers will show you the wrong data

  e.g., `int`s represented as `long`s

  e.g., memory (pointers) may be initialized differently

An ounce of analysis is worth a pound of debugging!

3.2 Using the `dbx` Debugger

Nota Bene

Be sure to compile with the `-g` switch and without any `-O` switches (Can you say 'Makefile'?)

Invocation

`dbx` *program*
`dbx` *program* `core`

Essential Commands

where	print current call stack (can be used to discover where the program stopped after a core dump)
stop in *func*	set a breakpoint at first executable line of function *func*, e.g. main
stop at *line*	set a breakpoint at line *line*
run *args*	run your program using any (optional) command line arguments you supply
print *expr*	display the value of the expression *expr* (e.g., value of a variable, result of executing a function)

Some Other Commands

cont	continue executing after a breakpoint
step	execute the current line of your program
next	same as **step** but will not enter functions
status	shows breakpoints, etc. you have set
list	displays source lines. Use depends on `dbx` version. In GAUL list *start* : *num* shows lines *start* to *start* + *num*. Other versions use list *start*, *stop*. Some versions use list *func* to display the source lines of function *func*.
alias	can be used to create your own names for `dbx` commands, may allow you to make macros (depends on `dbx` version)

Where to Get Help With `dbx`

- From Unix shell

  `man dbx`

- From Within `dbx`

  `sh man dbx` (or `!man dbx`)

  `help` (or `?`)

4 Recommended Reading

- ☐ The Elements of Programming Style (2nd ed.)
  Brian W. Kernighan and P.J. Plauger.
  ©1978 by McGraw-Hill
  TAYSTK QA76.6.K47 (the first edition, ©1974)

- ☐ C A Reference Manual (3rd ed.)
  Samuel P. Harbison and Guy L. Steele Jr.
  ©1991 by Prentice Hall
  TAYSTK QA76.73.C15H38 1991