C Programming, Part 4: Debugging

Edit New Page

daeyun edited this page on Oct 6, 2014 · 2 revisions

The Hitchhiker's Guide to Debugging C Programs

Feel free to add anything that you found helpful in debugging C programs including but not limited to, debugger usage, recognizing common error types, gotchas, and effective googling tips.

1. Make your code modular using helper functions. If there is a repeated task (getting the pointers to contiguous blocks in MP2 for example), make them helper functions. And make sure each function does one thing very well, so that you don't have to debug twice.

2. Use assertions to make sure your code works up to a certain point -- and importantly, to make sure you don't break it later. For example, if your data structure is a doubly linked list, you can do something like, assert(node->size == node->next->prev->size) to assert that the next node has a pointer to the current node. You can also check the pointer is pointing to an expected range of memory address, not null, ->size is reasonable etc. The NDEBUG macro will disable all assertions, so don't forget to set that once you finish debugging.
   http://www.cplusplus.com/reference/cassert/assert/

GDB

Introduction: http://www.cs.cmu.edu/~gilpin/tutorial/

Setting breakpoints programmatically

A very useful trick when debugging complex C programs with GDB is setting breakpoints in the source code.

```
int main() {
    int val = 1;
    val = 42;
    asm("int $3"); // set a breakpoint here
    val = 7;
}
```

```
$ gcc main.c -g -o main && ./main
(gdb) r
[...]
Program received signal SIGTRAP, Trace/breakpoint trap.
main () at main.c:6
6       val = 7;
```

```
(gdb) p val
$1 = 42
```

Checking memory content

http://www.delorie.com/gnu/docs/gdb/gdb_56.html

For example,

```
int main() {
    char bad_string[3] = {'C', 'a', 't'};
    printf("%s", bad_string);
}
```

```
$ gcc main.c -g -o main && ./main
$ Cat ZVQ$
```

```
(gdb) l
1   #include <stdio.h>
2   int main() {
3       char bad_string[3] = {'C', 'a', 't'};
4       printf("%s", bad_string);
5   }
(gdb) b 4
Breakpoint 1 at 0x100000f57: file main.c, line 4.
(gdb) r
[...]
Breakpoint 1, main () at main.c:4
4       printf("%s", bad_string);
(gdb) x/16xb bad_string
0x7fff5fbff9cd: 0x63    0x61    0x74    0xe0    0xf9    0xbf    0x5f    0xff
0x7fff5fbff9d5: 0x7f    0x00    0x00    0xfd    0xb5    0x23    0x89    0xff
```

Here, by using the `x` command with parameters `16xb`, we can see that starting at memory address `0x7fff5fbff9c` (value of `bad_string`), printf would actually see the following sequence of bytes as a string because we provided a malformed string without a null terminator.

```
0x43 0x61 0x74 0xe0 0xf9 0xbf 0x5f 0xff 0x7f 0x00
```