

Debug – It is an Art, not a Science

Pangfeng Liu
National Taiwan University

March 25, 2013

Introduction

- The art of debugging is to remove mistakes from your programs.
- It depends heavily on experiences.
- It may not be a scientific process.

Good Ways to Debug

- Use debugger, instead of printf.
- Use assertion.
- Write checking functions.
- Trace your code.

GNU GDB

- A debugging tool by GNU
- Step by step execution
- Graphic interface (?)
- Conditional breaking

GNU GDB

The GNU project allows you to see what is going on “inside” another program while it executes – or what another program was doing at the moment it crashed. ¹

¹<http://www.gnu.org/software/gdb/>

Commands

- file** Read the program to be debugged. The program should be compiled with `-g` to include symbol table.
- list** List program according to function or line number.
- break** Set a break point at a function or a line.
- condition** Set the condition of a breakpoint.
- run** Start running the program. You may specify argument, and input/output redirection are allowed.

Commands

`print` Print the variables.

`continue` Continue the execution.

`where` Print the contents of the calling stack.

`up` and `down` Go up and down of the calling stack.

`quit` Quit gdb.

Test Commands

Example 1: (gdb-test.c) Example

```
1 #include <stdio.h>
2 int main()
3 {
4     int a = 3;
5     int b = 5;
6     int c;
7     scanf("%d", c);
8     printf("%d\n", a + b + c);
9 }
```


Session 1

- ① Recompile the program with `-g`
- ② Use `gdb` to invoke the debugger.
- ③ Use `file` to read the executable.
- ④ Use `list` to list the program.
- ⑤ Use `break` to set the breakpoint at a line.
- ⑥ Use `run` to start the program.
- ⑦ Use `print` to see the variables.
- ⑧ Use `continue` to continue.
- ⑨ Use `where` to examine the calling stack.
- ⑩ Use `quit` to leave `gdb`.

Stepping Commands

- `next` Finish the next statement without going into functions encountered in this statement.
- `step` Finish the next statement but will go into the function call within this statement.
- `clear` Clear a break point.

Test Commands

Example 2: (fab.c) Example

```
1 #include <stdio.h>
2 int fab(int n)
3 {
4     return(fab(n - 1) + fab(n - 2));
5 }
6 int main()
7 {
8     printf("%d\n", fab(3));
9 }
```

Session 2

- 1 Recompile the program with `-g`
- 2 Use `gdb` to invoke the debugger.
- 3 Use `file` to read the executable.
- 4 Use `list` to list the program.
- 5 Use `break` to set the breakpoint, and this time we break at the function `fab`.
- 6 Use `run` to start the program.

Session 2

- ① Use print to see the variables.
- ② Use step to step into the next level stack.
- ③ Use where to examine the calling stack.
- ④ Use up and down to examine the contents of the stack.

Session 2

- ① Use clear to clear the breakpoint set.
- ② Use continue to recreate the segmentation fault.
- ③ Use where to examine the calling stack.
- ④ Use up and down to examine the contents of the stack.

Session 3

- Use the same fab example program.
- We suspect that the recursion causes negative `n`.
- We set a *conditional* breakpoint for `n` to find out.

Session 3

- ① Use `gdb` to invoke the debugger.
- ② Use `file` to read the executable.
- ③ Use `list` to list the program.
- ④ Use `break` to set a breakpoint. Remember the breakpoint id.
- ⑤ Use `conditional` to make the breakpoint stop only when `n` is negative.
- ⑥ Use `run` to start the program.
- ⑦ Use `where` to examine the stack.

Core Dump

- Core is memory
- To dump is to throw out something.
- A *core dump* is the last minute “snapshot” of your program before it was terminated by operating system.
- A core dump can tell us a story, just like a body.

Segmentation Faults

Example 3: (scanf.c) A scanf bug

```
1 #include <stdio.h>
2 main()
3 {
4     int i;
5     scanf("%d", i);
6     printf("%d\n", i);
7 }
```

Get Core Dump

You may need to set the limit to get the core dump.

```
1 pangfeng@linux4:> limit
2 cputime          unlimited
3 filesize         unlimited
4 datasize         unlimited
5 stacksize       8192 kbytes
6 coredumpsize    0 kbytes
7 memoryuse       unlimited
8 vmemoryuse      unlimited
9 descriptors     1024
10 memorylocked   64 kbytes
11 maxproc        1033286
12 maxlocks       unlimited
13 maxsignal      1033286
14 maxmessage     819200
15 maxnice        0
16 maxrtprio      0
17 maxrttime      unlimited
```

Get Core Dump

```
1 pangfeng@linux4:> limit coredumpsize unlimited
2 pangfeng@linux4:> limit
3 cputime          unlimited
4 filesize         unlimited
5 datasize         unlimited
6 stacksize       8192 kbytes
7 coredumpsize     unlimited
8 memoryuse        unlimited
9 vmemoryuse       unlimited
10 descriptors      1024
11 memorylocked    64 kbytes
12 maxproc         1033286
13 maxlocks        unlimited
14 maxsignal       1033286
15 maxmessage      819200
16 maxnice         0
17 maxrtprio       0
18 maxrttime       unlimited
19 pangfeng@linux4:/home/faculty/pangfeng/debug>
```

Session 4

- ① Set the limit so that the core dump can be generated.
- ② Run into segmentation fault and core dumped.
- ③ Recompile with `-g`.
- ④ Run the executable to generate the core dump.
- ⑤ Use `gdb` to examine the core, and pinpoint the line that causes the core dump.

Segmentation Faults

Example 4: (core.c) An array index bug

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int array[100];
6      printf("array[1000000] is %d\n",
7             array[1000000]);
8  }
```

Session 5

- Now try the process on file `core.c`.
- Try to generate a core dump, then examine it.

Valgrind

- Valgrind is an instrumentation framework for building dynamic analysis tools ².
- Most often we use the tools to check how our programs run.
 - memcheck
 - exp-sgcheck

²<http://valgrind.org/>

Memcheck

- Detects memory-management problems, and is aimed primarily at C and C++ programs.
- When a program is run under Memcheck's supervision, all reads and writes of memory are checked, and calls to `malloc/new/free/delete` are intercepted.

Detectable

- Bad frees of heap blocks (double frees, mismatched frees)
- Passing overlapping source and destination memory blocks to `memcpy()` and related functions
- Accesses memory it shouldn't (areas not yet allocated, areas that have been freed, areas past the end of heap blocks, inaccessible areas of the stack)
- Uninitialized values
- Memory leak

Usage

- 1 Compile your program with `-g` to include debugging information so that memcheck's error messages include exact line numbers³.
- 2 Run it.

³<http://valgrind.org/docs/manual/quick-start.html>

Usage

```
1 gcc -g myprog.c -o muprog
2 valgrind options myprog arg1 arg2
```

Valgrind Options


- `--tool=toolname`
- The default tool is `memcheck`⁴.
- `--leak-check` turns on the detailed memory leak detector.

⁴<http://valgrind.org/docs/manual/manual-core.html>

Example 5: (a.c) A memcheck example

```
1  #include <stdlib.h>
2
3  void f(void)
4  {
5      int* x = malloc(10 * sizeof(int));
6      x[10] = 0; // problem 1: heap block overrun
7  } // problem 2: memory leak -- x not freed
8
9  int main(void)
10 {
11     f();
12     return 0;
13 }
```

5

⁵<http://valgrind.org/docs/manual/quick-start.html> 

Results

```
1 pangfeng@linux4:> gcc -g a.c
2 pangfeng@linux4:> valgrind --leak-check=yes a.out
3 ==28040== Memcheck, a memory error detector
4 ==28040== Copyright (C) 2002-2011, and GNU GPL'd, by Julian
   Seward et al.
5 ==28040== Using Valgrind-3.7.0 and LibVEX; rerun with -h for
   copyright info
6 ==28040== Command: a.out
7 ==28040==
8 ==28040== Invalid write of size 4
9 ==28040==      at 0x40052A: f (a.c:6)
10 ==28040==     by 0x40053A: main (a.c:11)
11 ==28040== Address 0x51b9068 is 0 bytes after a block of
   size 40 alloc'd
12 ==28040==      at 0x4C28BED: malloc (vg_replace_malloc.c:263)
13 ==28040==     by 0x40051D: f (a.c:5)
14 ==28040==     by 0x40053A: main (a.c:11)
15 ==28040==
16 ==28040==
```

- The first line ("Invalid write...") tells you what kind of error it is. Here, the program wrote to some memory it should not have due to a heap block overrun.
- Below the first line is a stack trace telling you where the problem occurred. Stack traces can get quite large, and be confusing, especially if you are using the C++ STL. Reading them from the bottom up can help. If the stack trace is not big enough, use the `-num-callers` option to make it bigger.
- The code addresses are usually unimportant, but occasionally crucial for tracking down weirder bugs.
- Some error messages have a second component which describes the memory address involved. This one shows that the written memory is just past the end of a block allocated with `malloc()` on line 5 of `a.c`.

Results

```
17 ==28040==  HEAP SUMMARY:
18 ==28040==      in use at exit: 40 bytes in 1 blocks
19 ==28040==    total heap usage: 1 allocs, 0 frees, 40 bytes
    allocated
20 ==28040==
21 ==28040== 40 bytes in 1 blocks are definitely lost in loss
    record 1 of 1
22 ==28040==    at 0x4C28BED: malloc (vg_replace_malloc.c:263)
23 ==28040==    by 0x40051D: f (a.c:5)
24 ==28040==    by 0x40053A: main (a.c:11)
25 ==28040==
26 ==28040== LEAK SUMMARY:
27 ==28040==    definitely lost: 40 bytes in 1 blocks
28 ==28040==    indirectly lost: 0 bytes in 0 blocks
29 ==28040==    possibly lost: 0 bytes in 0 blocks
30 ==28040==    still reachable: 0 bytes in 0 blocks
31 ==28040==    suppressed: 0 bytes in 0 blocks
32 ==28040==
```

Memory Leak Messages

- The stack trace tells you where the leaked memory was allocated.
- There are several kinds of leaks; the two most important categories are:
 - “definitely lost” Your program is leaking memory – fix it!
 - “probably lost” Your program is leaking memory, unless you’re doing funny things with pointers (such as moving them to point to the middle of a heap block).

Summary Message

```
33 ==28040== For counts of detected and suppressed errors,  
    rerun with: -v  
34 ==28040== ERROR SUMMARY: 2 errors from 2 contexts (  
    suppressed: 4 from 4)  
35 pangfeng@linux4:/home/faculty/pangfeng/debug>
```

Cachegrind

- Cachegrind⁷ is a cache profiler. It performs detailed simulation of the I1, D1 and L2 caches in your CPU and so can accurately pinpoint the sources of cache misses in your code.
- It identifies the number of cache misses, memory references and instructions executed for each line of source code, with per-function, per-module and whole-program summaries. It is useful with programs written in any language. Cachegrind runs programs about 20–100x slower than normal.

⁷<http://valgrind.org/docs/manual/cg-manual.html>

Cachegrind

- Some modern machines have three levels of cache for which Cachegrind simulates the first-level and third-level caches.
- L3 cache has the most influence on runtime, as it masks accesses to main memory.
- L1 caches often have low associativity, so simulating them can detect cases where the code interacts badly with this cache.

Cachegrind

- Cachegrind always refers to the L1, D1 and LL (last-level) caches.
- L cache reads (Lr, which equals the number of instructions executed), L1 cache read misses (L1mr) and LL cache instruction read misses (LLmr).
- D cache reads (Dr, which equals the number of memory reads), D1 cache read misses (D1mr), and LL cache data read misses (LLmr).
- D cache writes (Dw, which equals the number of memory writes), D1 cache write misses (D1mw), and LL cache data write misses (LLmw).
- D1 total accesses is given by $D1mr + D1mw$, and that LL total accesses is given by $LLmr + LLmr + LLmw$.

Cachegrind Example

Example 6: (cache.c) Cachegrind Example

```
1  #include <stdio.h>
2  #define DIMX 1000
3  #define DIMY 100000
4  int array[DIMX][DIMY];
5  int main()
6  {
7      int x, y;
8      #ifdef XY
9          for (x = 0; x < DIMX; x++)
10             for (y = 0; y < DIMY; y++)
11                 array[x][y] = x + y;
12      #else
13          for (y = 0; y < DIMY; y++)
14             for (x = 0; x < DIMX; x++)
15                 array[x][y] = x + y;
16      #endif
17      return 0;
18  }
```

Makefile Example

```
1 cache-xy: cache.c
2     gcc -DXY cache.c -o cache-xy
3
4 cache-yx: cache.c
5     gcc cache.c -o cache-yx
6
7 run: cache-xy cache-yx
8     valgrind --tool=cachegrind cache-xy
9     valgrind --tool=cachegrind cache-yx
10    time cache-xy
11    time cache-yx
```


Cachegrind Results

```
1 pangfeng@linux4:/home/faculty/pangfeng/debug> make run
2 gcc -DXY cache.c -o cache-xy
3 gcc cache.c -o cache-yx
4 valgrind --tool=cachegrind cache-xy
5 ==2896== Cachegrind, a cache and branch-prediction profiler
6 ==2896== Copyright (C) 2002-2011, and GNU GPL'd, by Nicholas Nethercote et al.
7 ==2896== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright info
8 ==2896== Command: cache-xy
9 ==2896==
10 --2896-- warning: L3 cache found, using its data for the LL simulation.
11 --2896-- warning: pretending that LL cache has associativity 30 instead of
    actual 20
12 ==2896==
13 ==2896== I    refs:          1,300,109,477
14 ==2896== I1  misses:          715
15 ==2896== LLi misses:          703
16 ==2896== I1  miss rate:         0.00%
17 ==2896== LLi miss rate:         0.00%
18 ==2896==
19 ==2896== D    refs:          700,042,603 (600,030,323 rd + 100,012,280 wr)
20 ==2896== D1  misses:          6,251,739 (    1,263 rd +   6,250,476 wr)
21 ==2896== LLd misses:          6,251,565 (    1,118 rd +   6,250,447 wr)
22 ==2896== D1  miss rate:         0.8% (    0.0% +    6.2% )
23 ==2896== LLd miss rate:         0.8% (    0.0% +    6.2% )
24 ==2896==
25 ==2896== LL refs:          6,252,454 (    1,978 rd +   6,250,476 wr)
26 ==2896== LL misses:          6,252,268 (    1,821 rd +   6,250,447 wr)
27 ==2896== LL miss rate:         0.3% (    0.0% +    6.2% )
```

Cachegrind Results

```
28 valgrind --tool=cachegrind cache-yx
29 ==2968== Cachegrind, a cache and branch-prediction profiler
30 ==2968== Copyright (C) 2002-2011, and GNU GPL'd, by Nicholas Nethercote et al.
31 ==2968== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright info
32 ==2968== Command: cache-yx
33 ==2968==
34 --2968-- warning: L3 cache found, using its data for the LL simulation.
35 --2968-- warning: pretending that LL cache has associativity 30 instead of
    actual 20
36 ==2968==
37 ==2968== I    refs:          1,300,802,477
38 ==2968== I1   misses:          715
39 ==2968== LLi  misses:          703
40 ==2968== I1   miss rate:         0.00%
41 ==2968== LLi  miss rate:         0.00%
42 ==2968==
43 ==2968== D    refs:          700,438,603 (600,327,323 rd + 100,111,280 wr)
44 ==2968== D1   misses:          100,001,738 (    1,263 rd + 100,000,475 wr)
45 ==2968== LLd  misses:           6,252,564 (    1,118 rd +   6,251,446 wr)
46 ==2968== D1   miss rate:         14.2% (    0.0% +   99.8% )
47 ==2968== LLd  miss rate:         0.8% (    0.0% +    6.2% )
48 ==2968==
49 ==2968== LL refs:          100,002,453 (    1,978 rd + 100,000,475 wr)
50 ==2968== LL misses:           6,253,267 (    1,821 rd +   6,251,446 wr)
51 ==2968== LL miss rate:         0.3% (    0.0% +    6.2% )
```

Cachegrind Results

```
52 time cache-xy
53 0.38user 0.18system 0:00.58elapsed 95%CPU (0
    avgtext+0avgdata 391004maxresident)k
54 0inputs+0outputs (0major+97796minor)pagefaults 0
    swaps
55 time cache-yx
56 1.96user 0.28system 0:02.28elapsed 98%CPU (0
    avgtext+0avgdata 391008maxresident)k
57 0inputs+0outputs (0major+97798minor)pagefaults 0
    swaps
```

Example 7: (prime.c) Prime number

```
1 #include <stdio.h>
2 #include <math.h>
3 #include <assert.h>
4
5 #define PRIME_RANGE 20
6 bool notprime[PRIME_RANGE] = {0};
7 int n_prime = 0;
8 int primes[PRIME_RANGE] = {0};
```

Example 8: (prime.c) Prime number

```
12 int main()
13 {
14     int i,j;
15     int bound = (int) (sqrt(PRIME_RANGE)+.5);
16     for(i=2;i<=bound;i++){
17         if(!notprime[i]){
18             for(j=i+i;j<=PRIME_RANGE;j+=i){
19                 notprime[j] = 1;
20             }
21         }
22     }
23     for(i=2;i<PRIME_RANGE;i++){
24         if(notprime[i]==0){
25             primes[n_prime++] = i;
26         }
27     }
28     for(i=0; i<n_prime; ++i)
29         printf("%d\n", primes[i]);
30     return 0;
31 }
```

```
1 0
2 2
3 3
4 5
5 7
6 11
7 13
8 17
9 19
```

Use debugging tool to debug this program.

Assertion

- Must include the header file `assert.h`.
- A good practice for assuring that all the variables, including the parameters, are in order.
- Assertion should not be used to verify statement that has side effects.
 - `assert((fp = fopen("file" , "r")) != NULL)`

Pre-and-post Condition

- A pre-condition of a code block is an assertion that is always true before entering this code block.
- A post-condition of a code block is an assertion that is always true after leaving this code block, based on the assumption that the pre-condition is true.

An Example

Why the bubble sort is correct?

- The precondition of the i -th loop is that the j -th largest element has been moved to the j -th right most position in the array, for all j less or equal to i .
- The post-condition of the i -th loop is that the j -th largest element has been moved to the j -th right most position in the array, for all j less or equal to $i + 1$.

Bubble Sort Correctness

- It is easy to see that the post-condition of the i -th loop becomes the pre-condition of the $i + 1$ -th loop.
- When i is the number of elements, the post-condition of the $i - 1$ -th loop guarantees the correctness of the program.

Checking Functions

- Write a checking function to verify the integrity of your data structures.
- Periodically call the checking function to make sure that the data structure is correct.

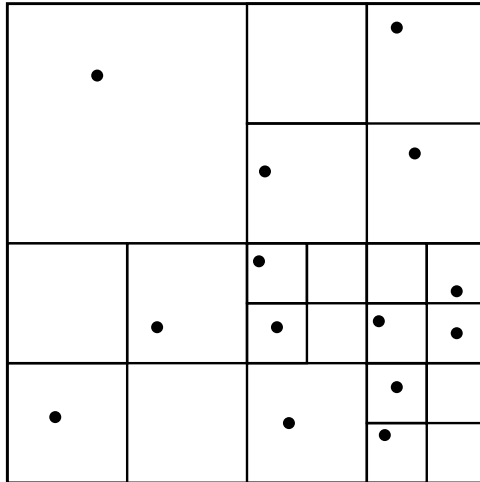
An Example

- Barnes-Hut Tree
- There are N particles in space.
- For simplicity we assume that the particles are in 2-dimension.
- First find a square bounding box to include all particles.
- The boxes are called “domains”.

BH Tree

- A domain is divided into four quadrants if it contains more than one particle.
- Eventually every particle is in its own domain.
- The domains form a *tree*.

BH Tree



Tree Node

Each tree node has the following information.

- The total number of particles in this node.
- The total mass of particles in this node.
- The pointers to all of its children.
- A pointer to its parent.
- A linked list containing all the particles in this node.

Particle List

- All the particles are in a linked list.
- Each particle has the following information.
 - The position
 - The mass
 - A pointer for the master linked list
 - A pointer for the linked list for a particular tree node

The Checking Function

- Check to see if the parent of each child is itself.
- Make sure that each internal node has at least two children.
- Make sure that the particle number is the sum of the particle numbers from its children.
- Make sure that the leaf node has exactly one particle, both in the linked list and in the particle number data.

The Checking Function

- Make sure that for each leaf the number of particles in the linked list is consistent with the number-of- particles data.
- Make sure that the mass of a node is equal to the sum of mass of it children.
- Check the position of the particles.
- Make sure that we have all the particles accounted for, i.e., we do not have missing particles.

Hand Trace

- A simple but fundamental technique.
- Make sure that you know the meaning of important variables.
- First check the values of important variables under normal conditions.
- Then check for extreme conditions.
- Step-by-step trace your code on the paper.

Source of Bugs

- Typos
- Initialization
- Logical error
- Array indices
- Overflow
- Cases not considered
- Memory bugs
- Bad macro
- Simply plain stupid

Typos

- `=` is different from `==`.
- `if (i = 0)` is actually an assignment.
- In Pascal assignment is `:=`.
- Use a variable that is very similar to another one.
- Do not use meaningless short variable names.
 - 1 and l.
 - 0 and o.

Assignment

- `if ((something = somefunction()) = NULL)`
- `if ((something = somefunction()) == NULL)`
- This should be taken apart into two statements.
 - `something = somefunction();`
 - `if (something == NULL)`

Index Variables

Example 9: (index-1.c) One loop

```
1  for (i = 0; i < 100; i++) {  
2      some processing;  
3  }
```

Index Variables

Example 10: (index-2.c) Two loops

```
1  for (i = 0; i < 100; i++)  
2  {  
3      some processing;  
4      for (i = 0; i < 40; i++)  
5          some other processing;  
6  }
```


Index Variables

Example 11: (switch.c) A switch statement

```
5  int type = 10;
6  int i = 10;
7  switch (type) {
8  case 1:
9      i = 0; printf("i = %d\n", i);
10     break;
11 case 2:
12     i = 4; printf("i = %d\n", i);
13     break;
14 default:
15     i = 5; printf("i = %d\n", i);
16     break;
17 }
```

Initialization

- Initialize variables.
- Some program may run differently in different systems.
- Some execution environment may initialize the variables for you, some may not.

An Example

Example 12: (init.c) Initialization

```
1  int main()  
2  {  
3      int sum;  
4      while (...) {  
5          some processing;  
6          sum += some data;  
7      }  
8  }
```

Another Example

Example 13: (init-2.c) Initialization

```
1  int sum;  
2  
3  int main()  
4  {  
5      while (...) {  
6          some processing;  
7          sum += some data;  
8      }  
9  }
```

Input

File 14: (sudoku.in), Input

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 5 | 3 | 0 | 0 | 7 | 0 | 0 | 0 | 0 |
| 2 | 6 | 0 | 0 | 1 | 9 | 5 | 0 | 0 | 0 |
| 3 | 0 | 9 | 8 | 0 | 0 | 0 | 0 | 6 | 0 |
| 4 | 8 | 0 | 0 | 0 | 6 | 0 | 0 | 0 | 3 |
| 5 | 4 | 0 | 0 | 8 | 0 | 3 | 0 | 0 | 1 |
| 6 | 7 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 6 |
| 7 | 0 | 6 | 0 | 0 | 0 | 0 | 2 | 8 | 0 |
| 8 | 0 | 0 | 0 | 4 | 1 | 9 | 0 | 0 | 5 |
| 9 | 0 | 0 | 0 | 0 | 8 | 0 | 0 | 7 | 9 |

Output

File 15: (sudoku.out), Output

```
1 5 3 1 2 7 4 9 10 10
2 6 4 10 1 9 5 10 10 10
3 10 9 8 10 10 10 1 6 2
4 8 1 2 7 6 10 10 10 3
5 4 10 10 8 10 3 10 10 1
6 7 10 3 5 2 1 4 9 6
7 1 6 5 10 10 10 2 8 10
8 3 10 10 4 1 9 10 10 5
9 10 10 4 3 8 2 6 7 9
```

Variables

Example 16: (sudoku.c) Variables

```
2  int  sudoku[9][9];  
3  int  row_count[9][10];  
4  int  col_count[9][10];  
5  int  box_count[3][3][10];
```

Main Program

Example 17: (sudoku.c) main program

```
54  int r, c;
55  for (r = 0; r < 9; ++r)
56      for(c = 0; c < 9; ++c) {
57          scanf("%d", &sudoku[r][c]);
58          set_count(r, c);
59      }
60  if (solver(0, 0)){
61      for (r = 0; r < 9; ++r) {
62          for (c = 0; c < 8; ++c)
63              printf("%d ", sudoku[r][c]);
64              printf("%d\n", sudoku[r][8]);
65      }
66  } else
67      printf("no solution\n");
```


Set

Example 18: (sudoku.c) Set

```
19 void set_count(int r, int c) {  
20     ++row_count[r][sudoku[r][c]];  
21     ++col_count[c][sudoku[r][c]];  
22     ++box_count[r/3][c/3][sudoku[r][c]];  
23 }  
24 void reset_count(int r, int c) {  
25     --row_count[r][sudoku[r][c]];  
26     --col_count[c][sudoku[r][c]];  
27     --box_count[r/3][c/3][sudoku[r][c]];  
28 }
```

Solver

Example 19: (sudoku.c) Solver

```
31 int solver(int r, int c) {  
32     if (r == 9) /* all cells are filled */  
33         return 1;  
34     if (sudoku[r][c]) /* the cell is prefilled */  
35         return (c < 8)? solver(r, c + 1) : solver(r + 1, 0);  
36  
37     for (sudoku[r][c] = 1; sudoku[r][c] <= 9; ++sudoku[r][c]){  
38         set_count(r, c);  
39         if (check(r, c))  
40             if (c < 8) {  
41                 if(solver(r, c + 1)) return 1;  
42             } else {  
43                 if(solver(r + 1, 0)) return 1;  
44             }  
45         reset_count(r, c);  
46     }  
47     return 0;
```

Logical Misunderstanding

What is the correct condition for the following?

- if
- while
- do while

Coin Changing

Given an amount of money M and the denomination of coins, find the minimum number of coins so that the value of coins is M .

Coin Changing

Assume that the maximum denomination is 500.

Example 20: (coin.c) Coin Changing

```
1  if (amount > 500)
2      amount = amount - 500;
3
4  if (amount >= 500)
5      amount = amount - 500;
```

Or

What does “or” mean?

Example 21: (or.c) Or

```
1  /* k is equal to 5 or 7. */  
2  
3  /* This will not work */  
4  if (k == 5 || 7)  
5      something;  
6  
7  /* k is equal to 5 or k is equal to 7 */  
8  if (k == 5 || k == 7)
```

Cubic

Given a metal bar, find the way to cut it into cubic for the maximum profit.

- The unit price of a cubic is proportional to its volume.
- If the type is 47 and the dimension is $12 \times 18 \times 24$, then the total profit is 11197440, which is $6^3 \times (10 \times 6^3) \times (2 \times 3 \times 4)$.
- We assume that the computation of type is correct.

Main

Example 22: (phantom.c) Main program

```
4  int main()  
5  {  
6      int type, w, h, l;  
7  
8      scanf("%d %d %d %d", &type, &w, &h, &l);  
9      printf("The metal value is %d.\n",  
10          value(type, w, h, l));  
11  
12     return 0;  
13 }
```


Value

Example 23: (phantom.c) Value

```
16 int value(int type, int width, int height, int length)
17 {
18     int price, side, volumn;
19     switch(type){
20     case 79: price = 30; break;
21     case 47: price = 10; break;
22     case 29: price = 4; break;
23     case 82: price = 5; break;
24     case 26: price = 3; break;
25     case 22: price = 9; break;
26     default: return -1;
27     }
28     if (width <= 0 || height <= 0 || length <= 0)
29         return -2;
30     side = gcd(height, length);
31     side = gcd(side, width);
32     volumn = side * side * side;
33     return price * volumn * width * height * length;
34 }
```

GCD

Example 24: (phantom.c) GCD

```
37 int gcd(int a, int b)
38 {
39     int c;
40     while(c) {
41         c = a % b;
42         a = b;
43         b = c;
44     }
45     return a;
46 }
```

Robots

Two robots moving at a m by n plane. Find out if they will run into each other and explode.

- R_1 starts at x_1, y_1 , and R_2 starts at x_2, y_2 .
- R_1 moves north for n_1 steps then east for e_1 steps, and R_2 moves east for e_2 steps then north for n_2 steps.
- Will “wrap” around the boundary.
- Has $f_1 f_2$ units of fuel each. Consumes one unit for every move.

Robots

Example 25: (2robots.c) main

```
4  int m,n,x1,y1,e1,n1,f1,x2,y2,e2,n2,f2;
5  int t=0;
6  scanf ("%d%d%d%d%d%d%d%d%d%d",
7          &m,&n,&x1,&y1,&e1,&n1,&f1,&x2,&y2,&e2,&n2,&f2);
8  while((f1>0)|| (f2>0)){
9      if (f1>0){ /* first if */
10         if (t%(n1+e1)<n1)
11             y1++;
12         else
13             x1++;
14         if (x1>m-1)
15             x1=x1-m;
16         if (y1>n-1)
17             y1=y1-n;
18         f1--;
19     }
```

Robots

Example 26: (2robots.c) main

```
21  if (f2>0){ /* second if */
22      if (t%(n2+e2)<e2)
23          x2++;
24      else
25          y2++;
26      if (x2>m-1)
27          x2=x2-m;
28      if (y2>n-1)
29          y2=y2-n;
30      f2--;
31  }
32  t++;
33  if((x1==x2)&&(y1==y2)){
34      printf("robots explode at time %d",t);
35      break;
36  }
37  } /* end while */
38  if ((x1!=x2)&&(y1!=y2))
39      printf("robots will not explode");
```

Array Index

- The computation of array indices may be complicated.
- Things get more complicated when the array indices starts with 1.

Array Index

What will happen when you print month[January]?

Example 27: (enum.c) Enumeration type

```
1 enum Month =  
2     {January = 1, February, March, April,  
3       May, June, July, August, September, October,  
4       November, December};  
5 char *month[] =  
6     {"January", "February", "March",  
7       "April", "May", "June", "July", "August",  
8       "September", "October", "November",  
9       "December"};
```

The “Right” Way

Example 28: (enum-2.c) Enumeration type

```
1 char *month[] =  
2     {"", "January", "February",  
3     "March", "April", "May", "June", "July",  
4     "August", "September", "October",  
5     "November", "December"};
```


July

Previously, it was called Quintilis in Latin, since it was the fifth month in the ancient Roman calendar, which traditionally set March as the beginning of the year before it was changed to January at the time of the decemvirs about 450 BC. The name was then changed by Augustus to honor Julius Caesar, who was born in July.

August

This month was originally named Sextilis in Latin, because it was the sixth month in the original ten-month Roman calendar when March was the first month of the year. About 700 BC it became the eighth month when January and February were added to the year before March by King Numa Pompilius, who also gave it 29 days. Julius Caesar added two days when he created the Julian calendar in 45 BC giving it its modern length of 31 days. In 8 BC it was renamed in honor of Augustus. According to a Senatus consultum quoted by Macrobius, he chose this month because it was the time of several of his great triumphs, including the conquest of Egypt.

Array Index Shifting

Example 29: (shift.c) Array index shifting

```
1  int count[26];  
2  ...  
3  if (when I see a lower case letter c)  
4      count[c]++;
```

The Correct Way

Example 30: (shift-2.c) Array index shifting

```
1  int count[26];  
2  ...  
3  if (when I see a lower case letter)  
4      count[c - 97]++;
```

The Even More Correct Way

Example 31: (shift-3.c) Array index shifting

```
1  int count[26];  
2  ...  
3  if (when I see a lower case letter)  
4      count[c - 'a']++;
```

Array Index Computation

Consider a binary heap implemented by an array. If you know the index of the parent, what are the indices of its children?

- $2i$ and $2i + 1$
- $2i + 1$ and $2i + 2$

Why 0?

Why C array index starts with 0?

- $A[i]$ is actually $*(A + i)$.
- The address is $A + i * \text{sizeof}(A[0])$.

Multi-Dimension

- We have `int a[4][5][6];`
- What is the address of `a[i][j][k]`?
- $a + (i \times 5 \times 6 + j \times 6 + k) \times 4$
- What will happen if the index starts at 1?

Function Prototype

Example 32: (array.c) Which will compile?

```
1 void print_array(int array[]);  
2 void print_array(int array[4][5]);  
3 void print_array(int array[][5]);  
4 void print_array(int array[4][]);
```

Array

Example 33: (2darray.c) What is the output?

```
1  #include <stdio.h>
2  void print_matrix(int a[4][3], int i, int j)
3  {
4      printf("a[%d][%d] = %d\n", i, j, a[i][j]);
5      return;
6  }
7  int main(void)
8  {
9      int array[3][4] = {{0, 1, 2, 3},
10                        {4, 5, 6, 7},
11                        {8, 9, 10, 11}};
12      printf("array[2][1] = %d\n", array[2][1]);
13      print_matrix(array, 2, 1);
14      printf("array[0][2] = %d\n", array[0][2]);
15      print_matrix(array, 0, 2);
16      return 0;
17  }
```

Rename a File

Example 34: (rename.c) Rename a file

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 char newname[256];
6 char search_str[256];
7 char replace_str[256];
```

Rename a File

Example 35: (rename.c) Rename a file

```
24 int main()  
25 {  
26     char filename[256];  
27     while(scanf("%s%s%s", filename, search_str,  
28               replace_str) != EOF){  
29         Func_rename(filename);  
30         printf("%s\n", newname);  
31     }  
32     return 0;  
33 }
```

Rename a File

Example 36: (rename.c) Rename a file

```
9 void Func_rename(char *filename){
10     int i = 0, j = 0;
11     char *chptr;
12     for(j = 0; j<256; j++)
13         newname[j] = 0;
14     while(strstr((filename + j), search_str) != NULL){
15         chptr = strstr((filename + j), search_str);
16         i = chptr - filename - j;
17         strncat(newname, filename + j, i);
18         strncat(newname, replace_str, strlen(replace_str));
19         j = i + strlen(search_str);
20     }
21     strncat(newname, (filename + j), (strlen(filename) - j));
22 }
```

Qsort

Example 37: (overflow.c) Qsort

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int values[] = {-2147483640, 50, 100};
4  int compare (const void * a, const void * b)
5  {
6      return ( *(int*)a - *(int*)b );
7  }
8  int main ()
9  {
10     int n;
11     qsort (values, 3, sizeof(int), compare);
12     for (n = 0; n < 3 ; n++)
13         printf ("%d ", values[n]);
14     return 0;
15 }
```

8

⁸<http://pangfengliu.blogspot.com/2011/01/this-is-why-you-should-not-use.html>

Unsigned

Example 38: (unsign-1.c) Unsigned int

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int i = 2147483647;
6      unsigned int ui = 2147483647;
7      if (i + 1 < 0)
8          printf("i + 1 < 0\n");
9      if (ui + 1 > 0)
10         printf("ui + 1 > 0\n");
11     if (ui + 1 > i + 1)
12         printf("ui + 1 > i + 1\n");
13     return 0;
14 }
```

Unsigned

Example 39: (unsign-2.c) Unsigned int

```
1  #include <stdio.h>
2
3  int main()
4  {
5      unsigned int ui = 2147483647;
6      if (ui + 1 > 0)
7          printf("ui + 1 > 0\n");
8      if (ui + 1 < -1)
9          printf("ui + 1 < -1\n");
10     return 0;
11 }
```


Compiler-Generated Runtime Checks

- Visual C++ .NET 2003 includes native runtime checks that catch truncation errors as integers are assigned to shorter variables that result in lost data.
- The gcc and g++ compilers include an -ftrapv compiler option that provides limited support for detecting integer exceptions at runtime.
- Try it on angel-wing.c

Cases not Considered

- What does it mean by “between i and j ”?
- `for (x = i; x <= j; x++)` did not produce the correct answer?
- The value of i is not necessarily less than j .
- Orz

Between?

Example 40: (between.c) Between i and j

```
1 start = min(i, j);  
2 end = max(i, j);  
3 for (x = start; x <= end; x++)
```

Memory Bugs

- Memory contents are accidentally changed.
- C is notorious for memory problems.
- Buffer overrun
- The contents are larger than the memory buffer provided.
- Invalid array index
- The index is less than 0 or greater than the array bound.
- Invalid pointer
- The pointer goes wild.

Buffer Overrun

- The function gets should be avoided.
- It does not provide any checking on the buffer length.
- Use fgets instead.

Buffer Overrun

Remember the prime number bug?

String Operations

Example 41: (string-op.c) String operations

```
1 char string1[100];  
2 char string2[100];  
3 ...  
4 strcpy(string1, string2);
```

- strcpy does not check for enough space before copying.
- It may happen that the copying could erase the 0 at the end of string1.

I/O Operation

- Both read and fread should be used carefully.
- If the contents of variables are changed for no reasons, check for the surrounding string and buffers.
- Make sure that the operations on them do not change other data.

Trailing 0

Example 42: (io-op.c) Trailing 0

```
1 char strings[6][80];  
2 /* read 6 strings.*/  
3 for (i = 0; i < 6; i++)  
4     printf("%s\n", string[i]);
```

- What will happen if all the input strings have 80 characters?
- What will happen if the reading starts at strings[9] and goes backward?

Bubble Sort

Example 43: (swap.c) Bubble sort

```
1  #include <stdio.h>
2  #define swap(x, y) x ^= y ^= x ^= y
3  int main()
4  {
5      int n[10] = {1, 2, 7, 5, 3, 10, 9, 8, 4, 4};
6      int i, j, temp;
7      for (i = 8; i >= 0; i--)
8          for (j = 0; j <= i; j++)
9              if (n[j] > n[j + 1])
10                 swap(n[j], n[j+1]);
11     for (i = 0; i < 10; i++)
12         printf("%d\n", n[i]);
13     return 0;
14 }
```

Forest

Example 44: (forest.c) Forest

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #define MAX 200
4
5 #define inside(x, y) (0<=(x) && (x)<n && 0<=(y) && (y)<m)
6
7 char map[MAX+1][MAX];
8 int n, m;
9 int area[MAX*MAX], num;
```

Forest

Example 45: (forest.c) Forest

```
29  int main()
30  {
31      int i, j;
32      scanf("%d%d", &n, &m);
33      for(i=0; i<n; ++i)
34          scanf("%s", map[i]);
35
36      num = 0;
37      for(i=0; i<n; ++i)
38          for(j=0; j<m; ++j)
39              if(map[i][j] == '1')
40                  area[num++] = count(i, j);
41      qsort(area, num, sizeof(int), comp);
42      for(i=0; i<num; ++i)
43          printf("%d\n", area[i]);
44      return 0;
45  }
```

Forest

Example 46: (forest.c) Forest

```
11 int count(int x, int y){
12     static int dx[] = {1, 0, -1, 0};
13     static int dy[] = {0, 1, 0, -1};
14     int i, ret = 1;
15
16     map[x][y] = '0';
17     for(i=0; i<4; ++i){
18         if(inside(x+=dx[i], y+=dy[i]) && map[x][y] == '1')
19             ret += count(x, y);
20         x-=dx[i];
21         y-=dy[i];
22     }
23     return ret;
24 }
25 int comp(const void* a, const void* b){
26     return *(int*)a > *(int*)b ? 1 : *(int*)a == *(int*)b ? 0 : -1
27 }
```