



Data Structures

1

“Programming in the Large” Steps



Design & Implement

- Program & programming style (done)
- Common data structures and algorithms <-- we are here
- Modularity
- Building techniques & tools (done)

Debug

- Debugging techniques & tools (done)

Test

- Testing techniques (done)

Maintain

- Performance improvement techniques & tools

2

Goals of this Lecture



Help you learn (or refresh your memory) about:

- Common data structures: linked lists and hash tables
- Why? Deep motivation:
- Common data structures serve as “high level building blocks”
 - A power programmer:
 - Rarely creates programs from scratch
 - Often creates programs using high level building blocks

Why? Shallow motivation:

- Provide background pertinent to Assignment 3
- ... esp. for those who have not taken COS 226

3

Common Task



Maintain a collection of key/value pairs

- Each key is a **string**; each value is an **int**
- Unknown number of key-value pairs

Examples

- (student name, grade)
- (“john smith”, 84), (“jane doe”, 93), (“bill clinton”, 81)
- (baseball player, number)
 - (“Ruth”, 3), (“Gehrig”, 4), (“Mantle”, 7)
- (variable name, value)
 - (“maxLength”, 2000), (“i”, 7), (“j”, -10)

4

Agenda



Linked lists

Hash tables

Hash table issues

5

Linked List Data Structure

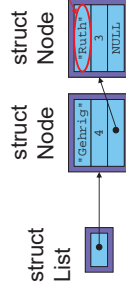


```
struct Node
{
    const char *key;
    int value;
    struct Node *next;
};

struct List
{
    struct Node *first;
};
```

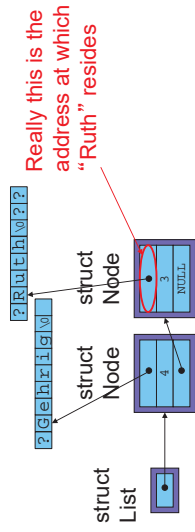
Your Assignment 3 data structures will be more elaborate

Really this is the address at which “Ruth” resides



6

Linked List Data Structure



7

Linked List Algorithms



Create

- Allocate `List` structure; set `first` to `NULL`
 - Performance: $O(1) \Rightarrow$ fast
- ### Add (no check for duplicate key required)
- Insert new node containing key/value pair at front of list
 - Performance: $O(1) \Rightarrow$ fast
- ### Add (check for duplicate key required)
- Traverse list to check for node with duplicate key
 - Insert new node containing key/value pair into list
 - Performance: $O(n) \Rightarrow$ slow

8

Linked List Algorithms



Search

- Traverse the list, looking for given key
- Stop when key found, or reach end
- Performance: $O(n) \Rightarrow$ slow

Free

- Free `Node` structures while traversing
- Free `List` structure
- Performance: $O(n) \Rightarrow$ slow



9

Agenda

- Linked lists
- Hash tables
- Hash table issues



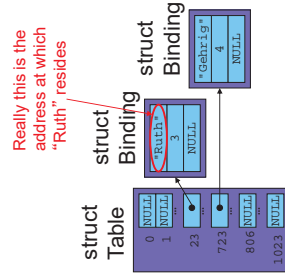
10

Hash Table Data Structure



Array of linked lists

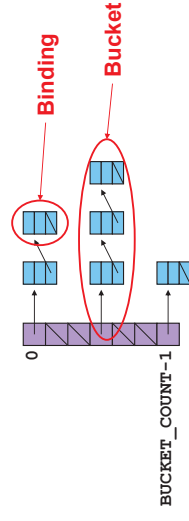
```
enum {BUCKET_COUNT = 1024};
struct Binding
{
    const char *key;
    int value;
    struct Binding *next;
};
struct Table
{
    struct Binding *buckets[BUCKET_COUNT];
};
```



Your Assignment 3 data structures will be more elaborate

11

Hash Table Data Structure



Hash function maps given key to an integer
Mod integer by `BUCKET_COUNT` to determine proper bucket

12

Hash Table Example



Example: `BUCKET_COUNT = 7`

Add (if not already present) bindings with these keys:

- the, cat, in, the, hat

13

Hash Table Example (cont.)



First key: "the"

- $\text{hash}(\text{"the"}) = 965156977; 965156977 \% 7 = 1$

Search `buckets [1]` for binding with key "the"; not found



14

Hash Table Example (cont.)



Add binding with key "the" and its value to `buckets [1]`



15

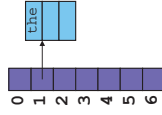
Hash Table Example (cont.)



Second key: "cat"

- $\text{hash}(\text{"cat"}) = 3895848756; 3895848756 \% 7 = 2$

Search `buckets [2]` for binding with key "cat"; not found

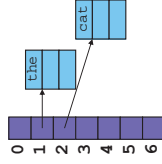


16

Hash Table Example (cont.)



Add binding with key "cat" and its value to `buckets [2]`



17

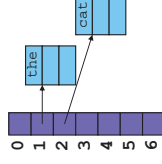
Hash Table Example (cont.)



Third key: "in"

- $\text{hash}(\text{"in"}) = 6888005; 6888005 \% 7 = 5$

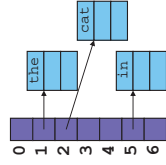
Search `buckets [5]` for binding with key "in"; not found



18

Hash Table Example (cont.)

Add binding with key "in" and its value to buckets [5]



19

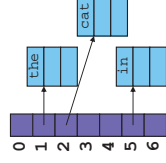
Hash Table Example (cont.)

Fourth word: "the"

- $\text{hash}(\text{"the"}) = 965156977; 965156977 \% 7 = 1$

Search buckets [1] for binding with key "the"; found it!

- Don't change hash table



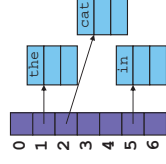
20

Hash Table Example (cont.)

Fifth key: "hat"

- $\text{hash}(\text{"hat"}) = 865559739; 865559739 \% 7 = 2$

Search buckets [2] for binding with key "hat"; not found

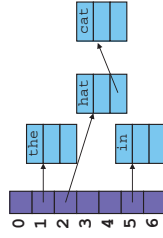


21

Hash Table Example (cont.)

Add binding with key "hat" and its value to buckets [2]

- At front or back? Doesn't matter
- Inserting at the front is easier, so add at the front



22

Hash Table Algorithms

Create

- Allocate `table` structure; set each bucket to `NULL`
- Performance: $O(1) \Rightarrow \text{fast}$

Add

- Hash the given key
- Mod by `BUCKET_COUNT` to determine proper bucket
- Traverse proper bucket to make sure no duplicate key
- Insert new binding containing key/value pair into proper bucket
- Performance: $O(1) \Rightarrow \text{fast}$

Is the add performance always fast?

23

Hash Table Algorithms

Search

- Hash the given key
- Mod by `BUCKET_COUNT` to determine proper bucket
- Traverse proper bucket, looking for binding with given key
- Stop when key found, or reach end
- Performance: $O(1) \Rightarrow \text{fast}$

Free

- Traverse each bucket, freeing bindings
- Free `table` structure
- Performance: $O(n) \Rightarrow \text{slow}$

Is the search performance always fast?

24

Agenda

- Linked lists
- Hash tables
- Hash table issues**

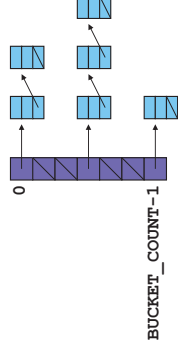
25

How Many Buckets?

Many!

- Too few \Rightarrow large buckets \Rightarrow slow add, slow search
- But not too many!
- Too many \Rightarrow memory is wasted

This is OK:



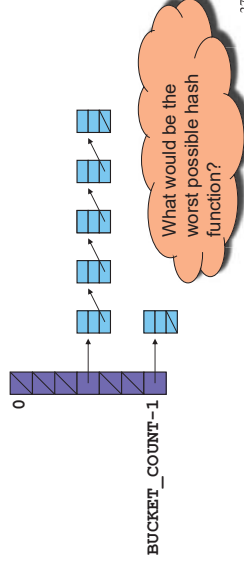
26

What Hash Function?

Should distribute bindings across the buckets well

- Distribute bindings over the range $0, 1, \dots, \text{BUCKET_COUNT}-1$
- Distribute bindings evenly to avoid very long buckets

This is not so good:



27

How to Hash Strings?

Simple hash schemes don't distribute the keys evenly enough

- Number of characters, mod BUCKET_COUNT
- Sum the numeric codes of all characters, mod BUCKET_COUNT
- ...

A reasonably good hash function:

- Weighted sum of characters s_i in the string s
- $(\sum a^i s_i) \bmod \text{BUCKET_COUNT}$
- Best if a and BUCKET_COUNT are relatively prime
- E.g., $a = 65599, \text{BUCKET_COUNT} = 1024$

Why?

28

How to Hash Strings?

Potentially expensive to compute $\sum a^i s_i$

So let's do some algebra ("Horner's rule")

- (by example, for string s of length 5, $a=65599$):

$$\begin{aligned} h &= \sum 65599^i \cdot s_i \\ h &= 65599^0 \cdot s_0 + 65599^1 \cdot s_1 + 65599^2 \cdot s_2 + 65599^3 \cdot s_3 + 65599^4 \cdot s_4 \\ \text{Direction of traversal of } s &\text{ doesn't matter, so...} \\ h &= 65599^0 \cdot s_4 + 65599^1 \cdot s_3 + 65599^2 \cdot s_2 + 65599^3 \cdot s_1 + 65599^4 \cdot s_0 \\ h &= 65599^4 \cdot s_0 + 65599^3 \cdot s_1 + 65599^2 \cdot s_2 + 65599^1 \cdot s_3 + 65599^0 \cdot s_4 \\ h &= (((((s_0) * 65599 + s_1) * 65599 + s_2) * 65599 + s_3) * 65599) + s_4 \end{aligned}$$

29

How to Hash Strings?

Yielding this function

```
size_t hash(const char *s, size_t bucketCount)
{
    size_t i;
    size_t h = 0;
    for (i=0; s[i]!='\0'; i++)
        h = h * 65599 + (size_t)s[i];
    return h % bucketCount;
}
```

30

How to Protect Keys?

Suppose `Table_add()` function contains this code:

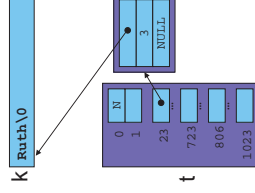
```
void Table_add(struct Table *t, const char *key, int value)
{
    ...
    struct Binding *p =
        (struct Binding*)malloc(sizeof(struct Binding));
    p->key = key;
    ...
}
```

31

How to Protect Keys?

Problem: Consider this calling code:

```
struct Table *t;
char k[100] = "Ruth";
...
Table_add(t, k, 3);
```

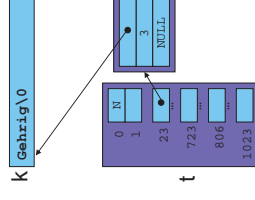


32

How to Protect Keys?

Problem: Consider this calling code:

```
struct Table *t;
char k[100] = "Ruth";
...
Table_add(t, k, 3);
strcpy(k, "Gehrig");
```



What happens if the client searches `t` for "Ruth"? For Gehrig?

33

How to Protect Keys?

Solution: `Table_add()` saves a **defensive copy** of the given key

```
void Table_add(struct Table *t, const char *key, int value)
{
    ...
    struct Binding *p =
        (struct Binding*)malloc(sizeof(struct Binding));
    p->key = (const char*)malloc(strlen(key) + 1);
    strcpy((char*)p->key, key);
    ...
}
```

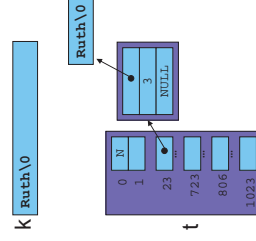
Why add 1?

34

How to Protect Keys?

Now consider same calling code:

```
struct Table *t;
char k[100] = "Ruth";
...
Table_add(t, k, 3);
```

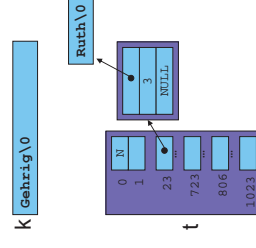


35

How to Protect Keys?

Now consider same calling code:

```
struct Table *t;
char k[100] = "Ruth";
...
Table_add(t, k, 3);
strcpy(k, "Gehrig");
```



Hash table is not corrupted

36

Who Owns the Keys?

Then the hash table **owns** its keys

- That is, the hash table owns the memory in which its keys reside
- `Hash_free()` function must free the memory in which the key resides



Summary

Common data structures and associated algorithms

- Linked list
- (Maybe) fast add
- Slow search
- Hash table
- (Potentially) fast add
- (Potentially) fast search
- Very common

Hash table issues

- Hashing algorithms
- Defensive copies
- Key ownership



Debugging (Part 2)



39

“Programming in the Large” Steps

Design & Implement

- Program & programming style (done)
- Common data structures and algorithms
- Modularity
- Building techniques & tools (done)

Test

- Testing techniques (done)

Debug

- Debugging techniques & tools <-- we are still here

Maintain

- Performance improvement techniques & tools



Goals of this Lecture

Help you learn about:

- Debugging strategies & tools related to **dynamic memory management (DMM)** *

Why?

- Many bugs occur in code that does DMM
- DMM errors can be difficult to find
 - DMM error in one area can manifest itself in a distant area
- A power programmer knows a wide variety of DMM debugging **strategies**
- A power programmer knows about **tools** that facilitate DMM debugging

- * Management of heap memory via `malloc()`, `calloc()`, `realloc()`, and `free()`



Agenda

- (9) Look for common DMM bugs
- (10) Diagnose seg faults using gdb
- (11) Manually inspect malloc calls
- (12) Hard-code malloc calls
- (13) Comment-out free calls
- (14) Use Meminfo
- (15) Use Valgrind



42



37

38

40

41

Look for Common DMM Bugs

Some of our favorites:

```
int *p; /* value of p undefined */
...
*p = somevalue;
```

```
char *p; /* value of p undefined */
...
fgets(p, 1024, stdin);
```

```
int *p;
...
p = (int*)malloc(sizeof(int));
...
*p = 5;
...
free(p);
...
*p = 6;
```

What are the errors?



43

Look for Common DMM Bugs

Some of our favorites:

```
int *p;
...
p = (int*)malloc(sizeof(int));
...
*p = 5;
...
p = (int*)malloc(sizeof(int));
```

```
int *p;
...
p = (int*)malloc(sizeof(int));
...
*p = 5;
...
free(p);
...
free(p);
```

What are the errors?



44

Agenda

- (9) Look for common DMM bugs
- (10) Diagnose seg faults using gdb**
- (11) Manually inspect malloc calls
- (12) Hard-code malloc calls
- (13) Comment-out free calls
- (14) Use Meminfo
- (15) Use Valgrind



45

Diagnose Seg Faults Using GDB

Segmentation fault ⇒ make it happen in gdb

- Then issue the gdb **where** command
- Output will lead you to the line that caused the fault
 - But that line may not be where the error resides!



46

Agenda

- (9) Look for common DMM bugs
- (10) Diagnose seg faults using gdb
- (11) Manually inspect malloc calls**
- (12) Hard-code malloc calls
- (13) Comment-out free calls
- (14) Use Meminfo
- (15) Use Valgrind



47

Manually Inspect Malloc Calls

Manually inspect each call of `malloc()`

- Make sure it allocates enough memory

Do the same for `calloc()` and `realloc()`



48

Manually Inspect Malloc Calls

Some of our favorites:

```
char *s1 = "hello, world";
char *s2;
s2 = (char*)malloc(strlen(s1));
strcpy(s2, s1);
```

```
char *s1 = "Hello";
char *s2;
s2 = (char*)malloc(sizeof(s1));
strcpy(s2, s1);
```

```
long double *p;
p = (long double*)malloc(sizeof(long double*));
```

```
long double *p;
p = (long double*)malloc(sizeof(p));
```

What are the errors?



49

Agenda

- (9) Look for common DMM bugs
- (10) Diagnose seg faults using gdb
- (11) Manually inspect malloc calls
- (12) Hard-code malloc calls**
- (13) Comment-out free calls
- (14) Use Meminfo
- (15) Use Valgrind



50

Hard-Code Malloc Calls

Temporarily change each call of `malloc()` to request a large number of bytes

- Say, 10000 bytes
- If the error disappears, then at least one of your calls is requesting too few bytes

Then incrementally restore each call of `malloc()` to its previous form

- When the error reappears, you might have found the culprit

Do the same for `calloc()` and `realloc()`



51

Agenda

- (9) Look for common DMM bugs
- (10) Diagnose seg faults using gdb
- (11) Manually inspect malloc calls
- (12) Hard-code malloc calls
- (13) Comment-out free calls**
- (14) Use Meminfo
- (15) Use Valgrind



52

Comment-Out Free Calls

Temporarily comment-out every call of `free()`

- If the error disappears, then program is
 - Freeing memory too soon, or
 - Freeing memory that already has been freed, or
 - Freeing memory that should not be freed,
 - Etc.

Then incrementally "comment-in" each call of `free()`

- When the error reappears, you might have found the culprit



53

Agenda

- (9) Look for common DMM bugs
- (10) Diagnose seg faults using gdb
- (11) Manually inspect malloc calls
- (12) Hard-code malloc calls
- (13) Comment-out free calls
- (14) Use Meminfo**
- (15) Use Valgrind



54

Use Meminfo

Use the **Meminfo** tool

- Simple tool
- Initial version written by Dondero
- Current version written by COS 217 alumnus RJ Lijffstrom
- Reports errors **after** program execution
 - Memory leaks
 - Some memory corruption
- User-friendly output

Appendix 1 provides example buggy programs

Appendix 2 provides Meminfo analyses



55

Agenda

- (9) Look for common DMM bugs
- (10) Diagnose seg faults using gdb
- (11) Manually inspect malloc calls
- (12) Hard-code malloc calls
- (13) Comment-out free calls
- (14) Use Meminfo
- (15) Use **Valgrind**



56

Use Valgrind

Use the **Valgrind** tool

- Complex tool
- Written by multiple developers, worldwide
 - See www.valgrind.org
- Reports errors **during** program execution
 - Memory leaks
 - Multiple frees
 - Dereferences of dangling pointers
 - Memory corruption
- Comprehensive output
 - But not always user-friendly



57

Use Valgrind

Appendix 1 provides example buggy programs

Appendix 3 provides Valgrind analyses



58

Summary

Strategies and tools for debugging the DMM aspects of your code:

- Look for common DMM bugs
- Diagnose seg faults using gdb
- Manually inspect malloc calls
- Hard-code malloc calls
- Comment-out free calls
- Use Meminfo
- Use Valgrind



59

Appendix 1: Buggy Programs

leak.c

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. int main(void)
4. { int *pi;
5.   pi = (int*)malloc(sizeof(int));
6.   *pi = 5;
7.   printf("%d\n", *pi);
8.   pi = (int*)malloc(sizeof(int));
9.   *pi = 6;
10.  printf("%d\n", *pi);
11.  free(pi);
12.  return 0;
13. }
```

Memory leak:

Memory allocated at line 5 is leaked



60

Appendix 1: Buggy Programs

doublefree.c

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. int main(void)
4. { int *pi;
5.   pi = (int*)malloc(sizeof(int));
6.   *pi = 5;
7.   printf("%d\n", *pi);
8.   free(pi);
9.   return 0;
11. }
```

Multiple free:

Memory allocated at line 5 is freed twice

61

Appendix 1: Buggy Programs

danglingptr.c

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. int main(void)
4. { int *pi;
5.   pi = (int*)malloc(sizeof(int));
6.   *pi = 5;
7.   printf("%d\n", *pi);
8.   free(pi);
9.   printf("%d\n", *pi);
10. return 0;
11. }
```

Dereference of dangling pointer:

Memory accessed at line 9 already was freed

62

Appendix 1: Buggy Programs

toosmall.c

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. int main(void)
4. { int *pi;
5.   pi = (int*)malloc(1);
6.   *pi = 5;
7.   printf("%d\n", *pi);
8.   free(pi);
9.   return 0;
10. }
```

Memory corruption:

Too little memory is allocated at line 5
Line 6 corrupts memory

63

Appendix 2: Meminfo

Meminfo can detect memory leaks:

```
$ gcc217m leak.c -o leak
$ leak
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
```

64

Appendix 2: Meminfo

Meminfo can detect memory corruption:

```
$ gcc217m toosmall.c -o toosmall
$ toosmall
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
```

65

Appendix 2: Meminfo

Meminfo caveats:

- Don't mix .o files built with gcc217 and gcc217m
- meminfo*.out files can be large
 - Should delete frequently
- Programs built with gcc217m run slower than those built with gcc217
 - Don't build with gcc217m when doing timing tests

66

Appendix 3: Valgrind

Valgrind can detect memory leaks:

```
$ gcc217 leak.c -o leak
$ valgrind leak
==31921== Memcheck, a memory error detector
==31921== Using Valgrind-3.8.1 and LibVEX; rerun with -h for copyright info
==31921== Command: leak
==31921==
5
6
==31921== HEAP SUMMARY:
==31921==   in use at exit: 4 bytes in 1 blocks
==31921== total heap usage: 2 allocs, 1 frees, 8 bytes allocated
==31921==
==31921== LEAK SUMMARY:
==31921==   definitely lost: 4 bytes in 1 blocks
==31921==   indirectly lost: 0 bytes in 0 blocks
==31921==   possibly lost: 0 bytes in 0 blocks
==31921==   still reachable: 0 bytes in 0 blocks
==31921==   suppressed: 0 bytes in 0 blocks
==31921== Rerun with --leak-check=full to see details of leaked memory
==31921==
==31921== For counts of detected and suppressed errors, rerun with: -v
==31921== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 6 from 6)
```

67

Appendix 3: Valgrind

Valgrind can detect memory leaks:

```
$ valgrind --leak-check=full leak
==476== Memcheck, a memory error detector
==476== Copyright (C) 2002-2012, and GNU GPL'd, by Julian Seward et al.
==476== Using Valgrind-3.8.1 and LibVEX; rerun with -h for copyright info
==476== Command: leak
==476==
5
6
==476== HEAP SUMMARY:
==476==   in use at exit: 2 allocs, 1 frees, 8 bytes allocated
==476== total heap usage: 2 allocs, 1 frees, 8 bytes allocated
==476==
==476== 4 bytes in 1 blocks are definitely lost in loss record 1 of 1
==476==   at 0x4A068E: malloc (vg_replace_malloc.c:270)
==476==   by 0x4A00585: main (leak.c:5)
==476==
==476== LEAK SUMMARY:
==476==   definitely lost: 4 bytes in 1 blocks
==476==   indirectly lost: 0 bytes in 0 blocks
==476==   possibly lost: 0 bytes in 0 blocks
==476==   still reachable: 0 bytes in 0 blocks
==476==   suppressed: 0 bytes in 0 blocks
==476==
==476== For counts of detected and suppressed errors, rerun with: -v
==476== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 6 from 6)
```

68

Appendix 3: Valgrind

Valgrind can detect multiple frees:

```
$ gcc217 doublefree.c -o doublefree
$ valgrind doublefree
==31951== Memcheck, a memory error detector
==31951== Copyright (C) 2002-2012, and GNU GPL'd, by Julian Seward et al.
==31951== Using Valgrind-3.8.1 and LibVEX; rerun with -h for copyright info
==31951== Command: doublefree
==31951==
5
==31951== Invalid free() / delete / delete[] / realloc()
==31951==   at 0x4A00585: main (doublefree.c:9)
==31951==   by 0x4A00585: main (doublefree.c:9)
==31951== Address 0x4A02A040 is 0 bytes inside a block of size 4 free'd
==31951==   at 0x4A00370: free (vg_replace_malloc.c:446)
==31951==   by 0x4A00589: main (doublefree.c:8)
==31951==
==31951== HEAP SUMMARY:
==31951==   in use at exit: 0 bytes in 0 blocks
==31951== total heap usage: 1 allocs, 2 frees, 4 bytes allocated
==31951==
==31951== All heap blocks were freed -- no leaks are possible
==31951==
==31951== For counts of detected and suppressed errors, rerun with: -v
==31951== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 6 from 6)
```

69

Appendix 3: Valgrind

Valgrind can detect memory leaks:

```
$ gcc217 danglingptr.c -o danglingptr
$ valgrind danglingptr
==336== Memcheck, a memory error detector
==336== Copyright (C) 2002-2012, and GNU GPL'd, by Julian Seward et al.
==336== Using Valgrind-3.8.1 and LibVEX; rerun with -h for copyright info
==336== Command: danglingptr
==336==
5
6
==336== Invalid read of size 4
==336==   at 0x4A00585: main (danglingptr.c:9)
==336==   Address 0x4A02A040 is 0 bytes inside a block of size 4 free'd
==336==   at 0x4A00370: free (vg_replace_malloc.c:446)
==336==   by 0x4A00589: main (danglingptr.c:8)
==336==
5
6
==336== HEAP SUMMARY:
==336==   in use at exit: 0 bytes in 0 blocks
==336== total heap usage: 1 allocs, 1 frees, 4 bytes allocated
==336==
==336== All heap blocks were freed -- no leaks are possible
==336==
==336== For counts of detected and suppressed errors, rerun with: -v
==336== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 6 from 6)
```

70

Appendix 3: Valgrind

Valgrind can detect memory leaks:

```
$ gcc217 toosmall.c -o toosmall
$ valgrind toosmall
==436== Memcheck, a memory error detector
==436== Copyright (C) 2002-2012, and GNU GPL'd, by Julian Seward et al.
==436== Using Valgrind-3.8.1 and LibVEX; rerun with -h for copyright info
==436== Command: toosmall
==436==
5
6
==436== Invalid write of size 4
==436==   at 0x4A00585: main (toosmall.c:6)
==436==   Address 0x4A0A068E: malloc (vg_replace_malloc.c:270)
==436==   by 0x4A00585: main (toosmall.c:5)
==436==
==436== Invalid read of size 4
==436==   at 0x4A00578: main (toosmall.c:7)
==436==   Address 0x4A0A068E: malloc (vg_replace_malloc.c:270)
==436==   by 0x4A00585: main (toosmall.c:5)
==436==
5
6
```

71

Appendix 3: Valgrind

Valgrind can detect memory corruption (cont.):

```
$ gcc217 toosmall.c -o toosmall
$ valgrind toosmall
==436== Memcheck, a memory error detector
==436== Copyright (C) 2002-2012, and GNU GPL'd, by Julian Seward et al.
==436== Using Valgrind-3.8.1 and LibVEX; rerun with -h for copyright info
==436== Command: toosmall
==436==
5
6
==436== Invalid read of size 4
==436==   at 0x4A00578: main (toosmall.c:7)
==436==   Address 0x4A0A068E: malloc (vg_replace_malloc.c:270)
==436==   by 0x4A00585: main (toosmall.c:5)
==436==
==436== Invalid read of size 4
==436==   at 0x4A00578: main (toosmall.c:7)
==436==   Address 0x4A0A068E: malloc (vg_replace_malloc.c:270)
==436==   by 0x4A00585: main (toosmall.c:5)
==436==
5
6
```

72

Appendix 3: Valgrind

Valgrind can detect dereferences of dangling pointers:

```
$ gcc217 danglingptr.c -o danglingptr
$ valgrind danglingptr
==336== Memcheck, a memory error detector
==336== Copyright (C) 2002-2012, and GNU GPL'd, by Julian Seward et al.
==336== Using Valgrind-3.8.1 and LibVEX; rerun with -h for copyright info
==336== Command: danglingptr
==336==
5
6
==336== Invalid read of size 4
==336==   at 0x4A00585: main (danglingptr.c:9)
==336==   Address 0x4A02A040 is 0 bytes inside a block of size 4 free'd
==336==   at 0x4A00370: free (vg_replace_malloc.c:446)
==336==   by 0x4A00589: main (danglingptr.c:8)
==336==
5
6
==336== HEAP SUMMARY:
==336==   in use at exit: 0 bytes in 0 blocks
==336== total heap usage: 1 allocs, 1 frees, 4 bytes allocated
==336==
==336== All heap blocks were freed -- no leaks are possible
==336==
==336== For counts of detected and suppressed errors, rerun with: -v
==336== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 6 from 6)
```

70

Appendix 3: Valgrind

Valgrind can detect memory corruption:

```
$ gcc217 toosmall.c -o toosmall
$ valgrind toosmall
==436== Memcheck, a memory error detector
==436== Copyright (C) 2002-2012, and GNU GPL'd, by Julian Seward et al.
==436== Using Valgrind-3.8.1 and LibVEX; rerun with -h for copyright info
==436== Command: toosmall
==436==
5
6
==436== Invalid write of size 4
==436==   at 0x4A00585: main (toosmall.c:6)
==436==   Address 0x4A0A068E: malloc (vg_replace_malloc.c:270)
==436==   by 0x4A00585: main (toosmall.c:5)
==436==
==436== Invalid read of size 4
==436==   at 0x4A00578: main (toosmall.c:7)
==436==   Address 0x4A0A068E: malloc (vg_replace_malloc.c:270)
==436==   by 0x4A00585: main (toosmall.c:5)
==436==
5
6
```

71

Appendix 3: Valgrind

Valgrind can detect memory corruption (cont.):

```
$ gcc217 toosmall.c -o toosmall
$ valgrind toosmall
==436== Memcheck, a memory error detector
==436== Copyright (C) 2002-2012, and GNU GPL'd, by Julian Seward et al.
==436== Using Valgrind-3.8.1 and LibVEX; rerun with -h for copyright info
==436== Command: toosmall
==436==
5
6
==436== Invalid write of size 4
==436==   at 0x4A00585: main (toosmall.c:6)
==436==   Address 0x4A0A068E: malloc (vg_replace_malloc.c:270)
==436==   by 0x4A00585: main (toosmall.c:5)
==436==
==436== Invalid read of size 4
==436==   at 0x4A00578: main (toosmall.c:7)
==436==   Address 0x4A0A068E: malloc (vg_replace_malloc.c:270)
==436==   by 0x4A00585: main (toosmall.c:5)
==436==
5
6
```

72

Appendix 3: Valgrind



Valgrind caveats:

- Not intended for programmers who are new to C
 - Messages may be cryptic
- Suggestion:
 - Observe line numbers referenced by messages
 - Study code at those lines
 - Infer meanings of messages