

TDIU16: Process- och operativsystemprogrammering

Lab 1: Introduktion till Pintos och C-programmering

Filip Strömbäck, Klas Arvidsson, Daniel Thorén

Mål

Denna laboration har följande mål:

1. Installera Pintos på ditt system
2. Öva på programmering i C
3. Öva på felsökning med GDB

Som en del av punkt 2 ovan kommer du att implementera en *associativ container* som vi också kommer att använda i kommande laborationer i Pintos. Felsökning med GDB är också något som underlättar felsökning av framtida laborationer.

Tanken är att laborationen ska ta ca 4 timmar att göra. Laborationen är tänkt som en introduktion till C och GDB. I övrigt introducerar den inga nya koncept.

Del A Installera Pintos

Målet med denna del är att skapa din egen kopia av Pintos-repositoryt att jobba med under kursen, och se till så att du kan kompilera och köra Pintos framöver.

För information om detta, se rubrikerna "Introduktion", "Installation", och "Bra att känna till om C" i Pintos-Wiki.

Denna del av laborationen behöver du inte redovisa.

Del B Programmering i C och felsökning med GDB

Denna del har som mål att ge en introduktion till programmering i C, men fokus ligger på felsökning med GDB. I denna del kommer vi att arbeta utanför Pintos, vilket innebär att vi kan starta GDB som vanligt. För att felsöka kod inuti Pintos (relevant från laboration 2 och framåt) ser processen att starta Pintos lite annorlunda ut, men resten är detsamma. Detta är beskrivet under "Felsökning med GDB" under "Felsökning" i Pintos-Wiki. Det finns även en kort sammanfattning i slutet av denna del.

I mappen `src/standalone/lab1b/` finns tre C-program: `debug1.c`, `debug2.c`, och `debug3.c`. Dessa program innehåller fel som gör att programmen kraschar. Uppgiften är att hitta felen med hjälp av GDB, och att sedan korrigera felen på lämpligt sätt.

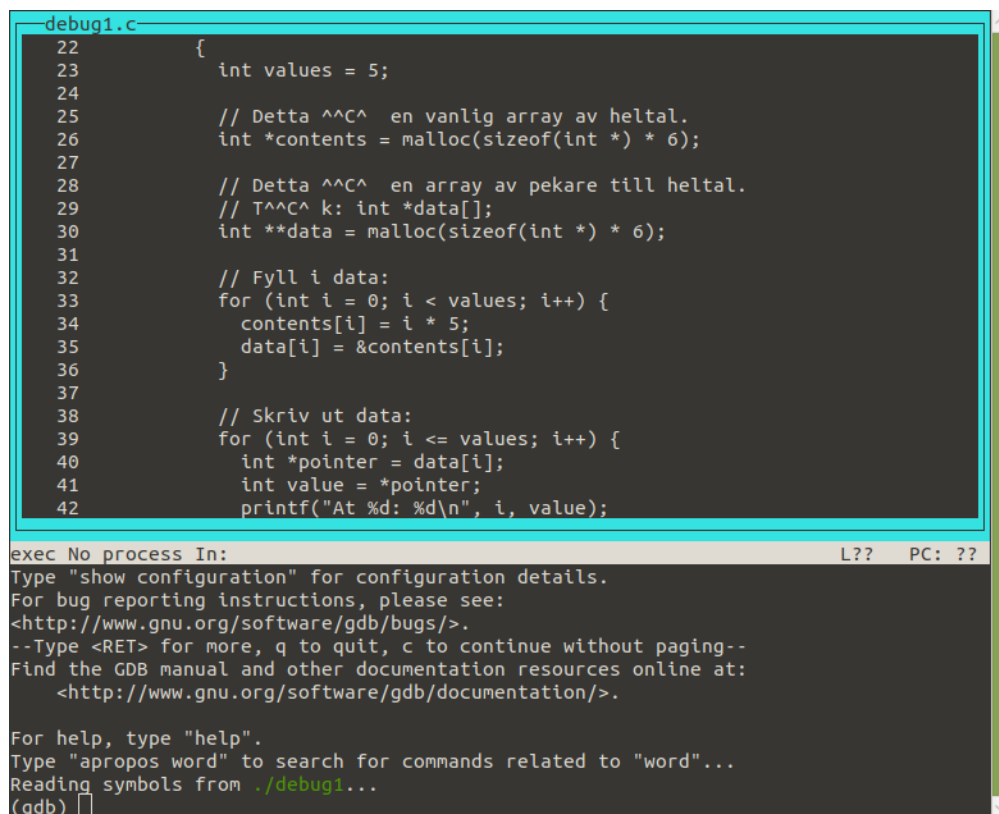
Notera: I den här delen används en version av funktionerna `malloc` och `free` som gör att programmet kraschar snabbare vid minnesfel. Detta gör felsökningen enklare i den här uppgiften. Detta betyder dock **inte** att du kan lita på att ditt program kraschar direkt ifall du råkar göra samma fel. Det kan vara så att programmet ser ut att fungera, men kraschar långt senare, när du lägger till ny men orelaterad kod.

Tips: Om du inte förstår vad som händer i någon del av programmen, så kan du använda visualiseringsverktyget som är beskrivet på kurshemsidan för att visualisera programmen om du vill. På skolans datorer och via ThinLinc kan det startas med kommandot `/courses/TDIU16/progvis/start` Instruktioner för att köra verktyget på egen dator finns på kurshemsidan. Se dock till att öva på att använda GDB också, eftersom Pintos inte går att köra i visualiseringsverktyget.

B.1 Program 1: `debug1.c`

Du kan kompilera programmet med kommandot `make debug1`. Du kan sedan köra programmet med kommandot `./debug1`. Du kommer då se att programmet kraschar med meddelandet `Segmentation fault (core dumped)`. Detta innebär att programmet försökte komma åt minne på en adress som av någon anledning inte var giltigt.

För att felsöka med GDB kan vi köra kommandot: `gdb -tui ./debug1`. Flaggan `-tui` öppnar det som GDB kallar för ett *Text User Interface* (därav flaggan `-tui`). Glömmer man flaggan `-tui` går det även att starta TUI:t genom att skriva `tui enable` inuti GDB, eller genom att trycka Ctrl+X följt av Ctrl+A. När du har startat GDB kommer du se något liknande bilden nedan:



```

debug1.c
22      {
23          int values = 5;
24
25          // Detta ^^C^ en vanlig array av heltal.
26          int *contents = malloc(sizeof(int *) * 6);
27
28          // Detta ^^C^ en array av pekare till heltal.
29          // T^^C^ k: int *data[];
30          int **data = malloc(sizeof(int *) * 6);
31
32          // Fyll i data:
33          for (int i = 0; i < values; i++) {
34              contents[i] = i * 5;
35              data[i] = &contents[i];
36          }
37
38          // Skriv ut data:
39          for (int i = 0; i <= values; i++) {
40              int *pointer = data[i];
41              int value = *pointer;
42              printf("At %d: %d\n", i, value);
43          }
44      }
45  }
46
47  int main(void) {
48      debug1();
49      return 0;
50  }
51
52  #include <stdio.h>
53  #include <stdlib.h>
54
55  void debug1() {
56      // ...
57  }
58
59  #endif
60
61  #endif
62
63  #endif
64
65  #endif
66
67  #endif
68
69  #endif
70
71  #endif
72
73  #endif
74
75  #endif
76
77  #endif
78
79  #endif
80
81  #endif
82
83  #endif
84
85  #endif
86
87  #endif
88
89  #endif
90
91  #endif
92
93  #endif
94
95  #endif
96
97  #endif
98
99  #endif
100
101  #endif
102
103  #endif
104
105  #endif
106
107  #endif
108
109  #endif
110
111  #endif
112
113  #endif
114
115  #endif
116
117  #endif
118
119  #endif
120
121  #endif
122
123  #endif
124
125  #endif
126
127  #endif
128
129  #endif
130
131  #endif
132
133  #endif
134
135  #endif
136
137  #endif
138
139  #endif
140
141  #endif
142
143  #endif
144
145  #endif
146
147  #endif
148
149  #endif
150
151  #endif
152
153  #endif
154
155  #endif
156
157  #endif
158
159  #endif
160
161  #endif
162
163  #endif
164
165  #endif
166
167  #endif
168
169  #endif
170
171  #endif
172
173  #endif
174
175  #endif
176
177  #endif
178
179  #endif
180
181  #endif
182
183  #endif
184
185  #endif
186
187  #endif
188
189  #endif
190
191  #endif
192
193  #endif
194
195  #endif
196
197  #endif
198
199  #endif
200
201  #endif
202
203  #endif
204
205  #endif
206
207  #endif
208
209  #endif
210
211  #endif
212
213  #endif
214
215  #endif
216
217  #endif
218
219  #endif
220
221  #endif
222
223  #endif
224
225  #endif
226
227  #endif
228
229  #endif
230
231  #endif
232
233  #endif
234
235  #endif
236
237  #endif
238
239  #endif
240
241  #endif
242
243  #endif
244
245  #endif
246
247  #endif
248
249  #endif
250
251  #endif
252
253  #endif
254
255  #endif
256
257  #endif
258
259  #endif
260
261  #endif
262
263  #endif
264
265  #endif
266
267  #endif
268
269  #endif
270
271  #endif
272
273  #endif
274
275  #endif
276
277  #endif
278
279  #endif
280
281  #endif
282
283  #endif
284
285  #endif
286
287  #endif
288
289  #endif
290
291  #endif
292
293  #endif
294
295  #endif
296
297  #endif
298
299  #endif
300
301  #endif
302
303  #endif
304
305  #endif
306
307  #endif
308
309  #endif
310
311  #endif
312
313  #endif
314
315  #endif
316
317  #endif
318
319  #endif
320
321  #endif
322
323  #endif
324
325  #endif
326
327  #endif
328
329  #endif
330
331  #endif
332
333  #endif
334
335  #endif
336
337  #endif
338
339  #endif
340
341  #endif
342
343  #endif
344
345  #endif
346
347  #endif
348
349  #endif
350
351  #endif
352
353  #endif
354
355  #endif
356
357  #endif
358
359  #endif
360
361  #endif
362
363  #endif
364
365  #endif
366
367  #endif
368
369  #endif
370
371  #endif
372
373  #endif
374
375  #endif
376
377  #endif
378
379  #endif
380
381  #endif
382
383  #endif
384
385  #endif
386
387  #endif
388
389  #endif
390
391  #endif
392
393  #endif
394
395  #endif
396
397  #endif
398
399  #endif
400
401  #endif
402
403  #endif
404
405  #endif
406
407  #endif
408
409  #endif
410
411  #endif
412
413  #endif
414
415  #endif
416
417  #endif
418
419  #endif
420
421  #endif
422
423  #endif
424
425  #endif
426
427  #endif
428
429  #endif
430
431  #endif
432
433  #endif
434
435  #endif
436
437  #endif
438
439  #endif
440
441  #endif
442
443  #endif
444
445  #endif
446
447  #endif
448
449  #endif
450
451  #endif
452
453  #endif
454
455  #endif
456
457  #endif
458
459  #endif
460
461  #endif
462
463  #endif
464
465  #endif
466
467  #endif
468
469  #endif
470
471  #endif
472
473  #endif
474
475  #endif
476
477  #endif
478
479  #endif
480
481  #endif
482
483  #endif
484
485  #endif
486
487  #endif
488
489  #endif
490
491  #endif
492
493  #endif
494
495  #endif
496
497  #endif
498
499  #endif
500
501  #endif
502
503  #endif
504
505  #endif
506
507  #endif
508
509  #endif
510
511  #endif
512
513  #endif
514
515  #endif
516
517  #endif
518
519  #endif
520
521  #endif
522
523  #endif
524
525  #endif
526
527  #endif
528
529  #endif
530
531  #endif
532
533  #endif
534
535  #endif
536
537  #endif
538
539  #endif
540
541  #endif
542
543  #endif
544
545  #endif
546
547  #endif
548
549  #endif
550
551  #endif
552
553  #endif
554
555  #endif
556
557  #endif
558
559  #endif
560
561  #endif
562
563  #endif
564
565  #endif
566
567  #endif
568
569  #endif
570
571  #endif
572
573  #endif
574
575  #endif
576
577  #endif
578
579  #endif
580
581  #endif
582
583  #endif
584
585  #endif
586
587  #endif
588
589  #endif
590
591  #endif
592
593  #endif
594
595  #endif
596
597  #endif
598
599  #endif
600
601  #endif
602
603  #endif
604
605  #endif
606
607  #endif
608
609  #endif
610
611  #endif
612
613  #endif
614
615  #endif
616
617  #endif
618
619  #endif
620
621  #endif
622
623  #endif
624
625  #endif
626
627  #endif
628
629  #endif
630
631  #endif
632
633  #endif
634
635  #endif
636
637  #endif
638
639  #endif
640
641  #endif
642
643  #endif
644
645  #endif
646
647  #endif
648
649  #endif
650
651  #endif
652
653  #endif
654
655  #endif
656
657  #endif
658
659  #endif
660
661  #endif
662
663  #endif
664
665  #endif
666
667  #endif
668
669  #endif
670
671  #endif
672
673  #endif
674
675  #endif
676
677  #endif
678
679  #endif
680
681  #endif
682
683  #endif
684
685  #endif
686
687  #endif
688
689  #endif
690
691  #endif
692
693  #endif
694
695  #endif
696
697  #endif
698
699  #endif
700
701  #endif
702
703  #endif
704
705  #endif
706
707  #endif
708
709  #endif
710
711  #endif
712
713  #endif
714
715  #endif
716
717  #endif
718
719  #endif
720
721  #endif
722
723  #endif
724
725  #endif
726
727  #endif
728
729  #endif
730
731  #endif
732
733  #endif
734
735  #endif
736
737  #endif
738
739  #endif
740
741  #endif
742
743  #endif
744
745  #endif
746
747  #endif
748
749  #endif
750
751  #endif
752
753  #endif
754
755  #endif
756
757  #endif
758
759  #endif
760
761  #endif
762
763  #endif
764
765  #endif
766
767  #endif
768
769  #endif
770
771  #endif
772
773  #endif
774
775  #endif
776
777  #endif
778
779  #endif
780
781  #endif
782
783  #endif
784
785  #endif
786
787  #endif
788
789  #endif
790
791  #endif
792
793  #endif
794
795  #endif
796
797  #endif
798
799  #endif
800
801  #endif
802
803  #endif
804
805  #endif
806
807  #endif
808
809  #endif
810
811  #endif
812
813  #endif
814
815  #endif
816
817  #endif
818
819  #endif
820
821  #endif
822
823  #endif
824
825  #endif
826
827  #endif
828
829  #endif
830
831  #endif
832
833  #endif
834
835  #endif
836
837  #endif
838
839  #endif
840
841  #endif
842
843  #endif
844
845  #endif
846
847  #endif
848
849  #endif
850
851  #endif
852
853  #endif
854
855  #endif
856
857  #endif
858
859  #endif
860
861  #endif
862
863  #endif
864
865  #endif
866
867  #endif
868
869  #endif
870
871  #endif
872
873  #endif
874
875  #endif
876
877  #endif
878
879  #endif
880
881  #endif
882
883  #endif
884
885  #endif
886
887  #endif
888
889  #endif
890
891  #endif
892
893  #endif
894
895  #endif
896
897  #endif
898
899  #endif
900
901  #endif
902
903  #endif
904
905  #endif
906
907  #endif
908
909  #endif
910
911  #endif
912
913  #endif
914
915  #endif
916
917  #endif
918
919  #endif
920
921  #endif
922
923  #endif
924
925  #endif
926
927  #endif
928
929  #endif
930
931  #endif
932
933  #endif
934
935  #endif
936
937  #endif
938
939  #endif
940
941  #endif
942
943  #endif
944
945  #endif
946
947  #endif
948
949  #endif
950
951  #endif
952
953  #endif
954
955  #endif
956
957  #endif
958
959  #endif
960
961  #endif
962
963  #endif
964
965  #endif
966
967  #endif
968
969  #endif
970
971  #endif
972
973  #endif
974
975  #endif
976
977  #endif
978
979  #endif
980
981  #endif
982
983  #endif
984
985  #endif
986
987  #endif
988
989  #endif
990
991  #endif
992
993  #endif
994
995  #endif
996
997  #endif
998
999  #endif
1000
1001  #endif
1002
1003  #endif
1004
1005  #endif
1006
1007  #endif
1008
1009  #endif
1010
1011  #endif
1012
1013  #endif
1014
1015  #endif
1016
1017  #endif
1018
1019  #endif
1020
1021  #endif
1022
1023  #endif
1024
1025  #endif
1026
1027  #endif
1028
1029  #endif
1030
1031  #endif
1032
1033  #endif
1034
1035  #endif
1036
1037  #endif
1038
1039  #endif
1040
1041  #endif
1042
1043  #endif
1044
1045  #endif
1046
1047  #endif
1048
1049  #endif
1050
1051  #endif
1052
1053  #endif
1054
1055  #endif
1056
1057  #endif
1058
1059  #endif
1060
1061  #endif
1062
1063  #endif
1064
1065  #endif
1066
1067  #endif
1068
1069  #endif
1070
1071  #endif
1072
1073  #endif
1074
1075  #endif
1076
1077  #endif
1078
1079  #endif
1080
1081  #endif
1082
1083  #endif
1084
1085  #endif
1086
1087  #endif
1088
1089  #endif
1090
1091  #endif
1092
1093  #endif
1094
1095  #endif
1096
1097  #endif
1098
1099  #endif
1100
1101  #endif
1102
1103  #endif
1104
1105  #endif
1106
1107  #endif
1108
1109  #endif
1110
1111  #endif
1112
1113  #endif
1114
1115  #endif
1116
1117  #endif
1118
1119  #endif
1120
1121  #endif
1122
1123  #endif
1124
1125  #endif
1126
1127  #endif
1128
1129  #endif
1130
1131  #endif
1132
1133  #endif
1134
1135  #endif
1136
1137  #endif
1138
1139  #endif
1140
1141  #endif
1142
1143  #endif
1144
1145  #endif
1146
1147  #endif
1148
1149  #endif
1150
1151  #endif
1152
1153  #endif
1154
1155  #endif
1156
1157  #endif
1158
1159  #endif
1160
1161  #endif
1162
1163  #endif
1164
1165  #endif
1166
1167  #endif
1168
1169  #endif
1170
1171  #endif
1172
1173  #endif
1174
1175  #endif
1176
1177  #endif
1178
1179  #endif
1180
1181  #endif
1182
1183  #endif
1184
1185  #endif
1186
1187  #endif
1188
1189  #endif
1190
1191  #endif
1192
1193  #endif
1194
1195  #endif
1196
1197  #endif
1198
1199  #endif
1200
1201  #endif
1202
1203  #endif
1204
1205  #endif
1206
1207  #endif
1208
1209  #endif
1210
1211  #endif
1212
1213  #endif
1214
1215  #endif
1216
1217  #endif
1218
1219  #endif
1220
1221  #endif
1222
1223  #endif
1224
1225  #endif
1226
1227  #endif
1228
1229  #endif
1230
1231  #endif
1232
1233  #endif
1234
1235  #endif
1236
1237  #endif
1238
1239  #endif
1240
1241  #endif
1242
1243  #endif
1244
1245  #endif
1246
1247  #endif
1248
1249  #endif
1250
1251  #endif
1252
1253  #endif
1254
1255  #endif
1256
1257  #endif
1258
1259  #endif
1260
1261  #endif
1262
1263  #endif
1264
1265  #endif
1266
1267  #endif
1268
1269  #endif
1270
1271  #endif
1272
1273  #endif
1274
1275  #endif
1276
1277  #endif
1278
1279  #endif
1280
1281  #endif
1282
1283  #endif
1284
1285  #endif
1286
1287  #endif
1288
1289  #endif
1290
1291  #endif
1292
1293  #endif
1294
1295  #endif
1296
1297  #endif
1298
1299  #endif
1300
1301  #endif
1302
1303  #endif
1304
1305  #endif
1306
1307  #endif
1308
1309  #endif
1310
1311  #endif
1312
1313  #endif
1314
1315  #endif
1316
1317  #endif
1318
1319  #endif
1320
1321  #endif
1322
1323  #endif
1324
1325  #endif
1326
1327  #endif
1328
1329  #endif
1330
1331  #endif
1332
1333  #endif
1334
1335  #endif
1336
1337  #endif
1338
1339  #endif
1340
1341  #endif
1342
1343  #endif
1344
1345  #endif
1346
1347  #endif
1348
1349  #endif
1350
1351  #endif
1352
1353  #endif
1354
1355  #endif
1356
1357  #endif
1358
1359  #endif
1360
1361  #endif
1362
1363  #endif
1364
1365  #endif
1366
1367  #endif
1368
1369  #endif
1370
1371  #endif
1372
1373  #endif
1374
1375  #endif
1376
1377  #endif
1378
1379  #endif
1380
1381  #endif
1382
1383  #endif
1384
1385  #endif
1386
1387  #endif
1388
1389  #endif
1390
1391  #endif
1392
1393  #endif
1394
1395  #endif
1396
1397  #endif
1398
1399  #endif
1400
1401  #endif
1402
1403  #endif
1404
1405  #endif
1406
1407  #endif
1408
1409  #endif
1410
1411  #endif
1412
1413  #endif
1414
1415  #endif
1416
1417  #endif
1418
1419  #endif
1420
1421  #endif
1422
1423  #endif
1424
1425  #endif
1426
1427  #endif
1428
1429  #endif
1430
1431  #endif
1432
1433  #endif
1434
1435  #endif
1436
1437  #endif
1438
1439  #endif
1440
1441  #endif
1442
1443  #endif
1444
1445  #endif
1446
1447  #endif
1448
1449  #endif
1450
1451  #endif
1452
1453  #endif
1454
1455  #endif
1456
1457  #endif
1458
1459  #endif
1460
1461  #endif
1462
1463  #endif
1464
1465  #endif
1466
1467  #endif
1468
1469  #endif
1470
1471  #endif
1472
1473  #endif
1474
1475  #endif
1476
1477  #endif
1478
1479  #endif
1480
1481  #endif
1482
1483  #endif
1484
1485  #endif
1486
1487  #endif
1488
1489  #endif
1490
1491  #endif
1492
1493  #endif
1494
1495  #endif
1496
1497  #endif
1498
1499  #endif
1500
1501  #endif
1502
1503  #endif
1504
1505  #endif
1506
1507  #endif
1508
1509  #endif
1510
1511  #endif
1512
1513  #endif
1514
1515  #endif
1516
1517  #endif
1518
1519  #endif
1520
1521  #endif
1522
1523  #endif
1524
1525  #endif
1526
1527  #endif
1528
1529  #endif
1530
1531  #endif
1532
1533  #endif
1534
1535  #endif
1536
1537  #endif
1538
1539  #endif
1540
1541  #endif
1542
1543  #endif
1544
1545  #endif
1546
1547  #endif
1548
1549  #endif
1550
1551  #endif
1552
1553  #endif
1554
1555  #endif
1556
1557  #endif
1558
1559  #endif
1560
1561  #endif
1562
1563  #endif
1564
1565  #endif
1566
1567  #endif
1568
1569  #endif
1570
1571  #endif
1572
1573  #endif
1574
1575  #endif
1576
1577  #endif
1578
1579  #endif
1580
1581  #endif
1582
1583  #endif
1584
1585  #endif
1586
1587  #endif
1588
1589  #endif
1590
1591  #endif
1592
1593  #endif
1594
1595  #endif
1596
1597  #endif
1598
1599  #endif
1600
1601  #endif
1602
1603  #endif
1604
1605  #endif
1606
1607  #endif
1608
1609  #endif
1610
1611  #endif
1612
1613  #endif
1614
1615  #endif
1616
1617  #endif
1618
1619  #endif
1620
1621  #endif
1622
1623  #endif
1624
1625  #endif
1626
1627  #endif
1628
1629  #endif
1630
1631  #endif
1632
1633  #endif
1634
1635  #endif
1636
1637  #endif
1638
1639  #endif
1640
1641  #endif
1642
1643  #endif
1644
1645  #endif
1646
1647  #endif
1648
1649  #endif
1650
1651  #endif
1652
1653  #endif
1654
1655  #endif
1656
1657  #endif
1658
1659  #endif
1660
1661  #endif
1662
1663  #endif
1664
1665  #endif
1666
1667  #endif
1668
1669  #endif
1670
1671  #endif
1672
1673  #endif
1674
1675  #endif
1676
1677  #endif
1678
1679  #endif
1680
1681  #endif
1682
1683  #endif
1684
1685  #endif
1686
1687  #endif
1688
1689  #endif
1690
1691  #endif
1692
1693  #endif
1694
1695  #endif
1696
1697  #endif
1698
1699  #endif
1700
1701  #endif
1702
1703  #endif
1704
1705  #endif
1706
1707  #endif
1708
1709  #endif
1710
1711  #endif
1712
1713  #endif
1714
1715  #endif
1716
1717  #endif
1718
1719  #endif
1720
1721  #endif
1722
1723  #endif
1724
1725  #endif
1726
1727  #endif
1728
1729  #endif
1730
1731  #endif
1732
1733  #endif
1734
1735  #endif
1736
1737  #endif
1738
1739  #endif
1740
1741  #endif
1742
1743  #endif
1744
1745  #endif
1746
1747  #endif
1748
1749  #endif
1750
1751  #endif
1752
1753  #endif
1754
1755  #endif
1756
1757  #endif
1758
1759  #endif
1760
1761  #endif
1762
1763  #endif
1764
1765  #endif
1766
1767  #endif
1768
1769  #endif
1770
1771  #endif
1772
1773  #endif
1774
1775  #endif
1776
1777  #endif
1778
1779  #endif
1780
1781  #endif
1782
1783  #endif
1784
1785  #endif
1786
1787  #endif
1788
1789  #endif
1790
1791  #endif
1792
1793  #endif
1794
1795  #endif
1796
1797  #endif
1798
1799  #endif
1800
1801  #endif
1802
1803  #endif
1804
1805  #endif
1806
1807  #endif
1808
1809  #endif
1810
1811  #endif
1812
1813  #endif
1814
1815  #endif
1816
1817  #endif
1818
1819  #endif
1820
1821  #endif
1822
1823  #endif
1824
1825  #endif
1826
1827  #endif
1828
1829  #endif
1830
1831  #endif
1832
1833  #endif
1834
1835  #endif
1836
1837  #endif
1838
1839  #endif
1840
1841  #endif
1842
1843  #endif
1844
1845  #endif
1846
1847  #endif
1848
1849  #endif
1850
1851  #endif
1852
1853  #endif
1854
1855  #endif
1856
1857  #endif
1858
1859  #endif
1860
1861  #endif
1862
1863  #endif
1864
1865  #endif
1866
1867  #endif
1868
1869  #endif
1870
1871  #endif
1872
1873  #endif
1874
1875  #endif
1876
1877  #endif
1878
1879  #endif
1880
1881  #endif
1882
1883  #endif
1884
1885  #endif
1886
1887  #endif
1888
1889  #endif
1890
1891  #endif
1892
1893  #endif
1894
1895  #endif
1896
1897  #endif
1898
1899  #endif
1900
1901  #endif
1902
1903  #endif
1904
1905  #endif
1906
1907  #endif
1908
1909  #endif
1910
1911  #endif
1912
1913  #endif
1914
1915  #endif
1916
1917  #endif
1918
1919  #endif
1920
1921  #endif
1922
1923  #endif
1924
1925  #endif
1926
1927  #endif
1928
1929  #endif
1930
1931  #endif
1932
1933  #endif
1934
1935  #endif
1936
1937  #endif
1938
1939  #endif
1940
1941  #endif
1942
1943  #endif
1944
1945  #endif
1946
1947  #endif
1948
1949  #endif
1950
1951  #endif
1952
1953  #endif
1954
1955  #endif
1956
1957  #endif
1958
1959  #endif
1960
1961  #endif
1962
1963  #endif
1964
1965  #endif
1966
1967  #endif
1968
1969  #endif
1970
1971  #endif
1972
1973  #endif
1974
1975  #endif
1976
1977  #endif
1978
1979  #endif
1980
1981  #endif
1982
1983  #endif
1984
1985  #endif
1986
1987  #endif
1988
1989  #endif
1990
1991  #endif
1992
1993  #endif
1994
1995  #endif
1996
1997  #endif
1998
1999  #endif
2000
2001  #endif
2002
2003  #endif
2004
2005  #endif
2006
2007  #endif
2008
```

Ctrl+L. Alltså, ser något konstigt ut, testa att trycka på Ctrl+L, det skadar aldrig!

Nu är vi redo att börja felsöka. Starta programmet genom att skriva kommandot **run** och tryck på enter. Värt att notera är att (nästan) alla kommandon i GDB går att förkorta. Så länge det är tydligt för GDB vad du menar så kommer den ofta att göra rätt sak. Det går exempelvis att förkorta **run** till **r**. I den här instruktionen skrivs hela kommandona ut, och förkortningar anges inom parentes efteråt där det är relevant.

När du skrivit **run** (**r**) och tryckt på retur så kommer GDB att starta programmet och låta det köra. I det här fallet så skriver programmet ut lite saker och kraschar sedan. På mitt system ser det då ut som i bilden nedan:

```
26         int *contents = malloc(sizeof(int *) * 6);
37
38         // Skriv ut data:
39         for (int i = 0; i <= values; i++) {
40             int *pointer = data[i];
41             int value = *pointer;
42             printf("At %d: %d\n", i, value);
43         }
44
45         free(data);
46         free(contents);
47
48         return 0;
49     }

```

native process 1403157 In: L?? PC: ??
<<http://www.gnu.org/software/gdb/bugs/>>.
--T>41 int value = *pointer; continue without paging--
Find the GDB manual and other documentation resources online at:
43 }
44
45 free(data);
46 free(contents);
47
48 return 0;
49 }

<<http://www.gnu.org/software/gdb/documentation/>>.
main L41 PC: 0x555555552a1

(gdb) run
Starting program: /home/filst04/TDIU16/given/src/standalone/lab1b/debug1
At 0: 0
At 1: 5
At 2: 10
At 3: 15
At 4: 20

Program received signal SIGSEGV, Segmentation fault.
0x0000555555552a1 in main () at debug1.c:41
(gdb) □

Här ser GDB trasigt ut (på grund av utdatan från programmet), alltså trycker jag på Ctrl+L och får följande:

```

debug1.c
32      // Fyll i data:
33      for (int i = 0; i < values; i++) {
34          contents[i] = i * 5;
35          data[i] = &contents[i];
36      }
37
38      // Skriv ut data:
39      for (int i = 0; i <= values; i++) {
40          int *pointer = data[i];
>41          int value = *pointer;
42          printf("At %d: %d\n", i, value);
43      }
44
45      free(data);
46      free(contents);
47
48      return 0;
49  }

native process 1403157 In: main L41 PC: 0x5555555552a1
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./debug1...
(gdb) run
Starting program: /home/filst04/TDIU16/given/src/standalone/lab1b/debug1

Program received signal SIGSEGV, Segmentation fault.
0x000055555552a1 in main () at debug1.c:41
(gdb)

```

I den övre delen kan vi se att GDB har markerat raden som höll på att köras när programmet kraschade. I den nedre delen kan vi se meddelandet **Program received signal SIGSEGV, Segmentation fault**. Detta är samma information som vi såg i terminalen förut. Nästa rad säger att programmet kraschade på adress `0x000055555552a1`. Detta är i sig inte intressant (vi vet inte vad som finns där, den varierar dessutom antagligen från körning till körning). GDB säger dock att det motsvarar rad 41 i filen `debug1.c`, och att denna rad ligger i funktionen `main`. Samma information visas också i övre halvan av fönstret på ett lite mer grafiskt sätt.

För att inse vad som gick fel kan vi nu inspektera vårt program med hjälp av kommandot `print (p)`. Kommandot evaluerar ett C-uttryck och skriver ut resultatet. Det mesta som fungerar i C fungerar även i GDB (men inte riktigt allt). Här kan vi exempelvis se värdet av variabeln `i` genom att skriva: `print i` (eller `p i`). Vi får då följande svar:

```
$1 = 5
```

Detta innebär alltså att `i` var 5. Delen `$1 =` innebär att vi kan använda `$1` i ett senare uttryck ifall vi vill återanvända resultatet på ett smidigt sätt (exempelvis `print $1 + 2` ger 7).

Insikten att `i` var 5 gör att vi kan fokusera på element nummer 5 i arrayen. Vi kan återigen använda `print`-kommandot för att inspektera programmet. Antingen kan vi skriva `print data[i]`, eller så ser vi att värdet vi är ute efter redan finns i variabeln `pointer` från raden innan och skriver ut `pointer` i stället. Vi får då följande svar:

```
$2 = (int *) 0xffffffffffffffff
```

Detta innebär att element fem i arrayet (och pekaren) har värdet `0xffffffffffffffff`, och att typen är `int *` (pekare till `int`). I och med att vi kraschade på raden där programmet försökte avreferera pekaren så kan vi från detta misstänka att pekaren inte är giltig, men för att vara säkra kan vi fråga GDB genom att

skriva: `print *pointer` (dvs. uttrycket som vi tror kraschade). Vi får då följande svar:

```
Cannot access memory at address 0xffffffffffffffcc
```

Detta bekräftar våra misstankar om att pekaren var ogiltig. Härifrån kan vi alltså dra slutsatsen att pekaren i element nummer 5 i `data` inte är korrekt. Nästa fråga är då varför detta är fallet. Vi kan felsöka detta med GDB, exempelvis genom att skapa en så kallad *brytpunkt* (eng: *breakpoint*). Detta kan vi göra med kommandot `break` (**b**). Till kommandot `break` behöver vi också ange var brytpunkten ska vara. Man kan antingen ge den ett namn på en funktion (exempelvis `break main`) eller ett filnamn och ett radnummer (`break debug1.c:41`). I vårt fall vill vi skapa en brytpunkt på rad 35, och vi kan därmed skriva `break debug1.c:41` (eller bara `break 35` eftersom det är den filen vi ser i övre halvan). Som svar markerar GDB brytpunkten med **b+** till vänster om radnumret, och svarar med:

```
Breakpoint 1 at 0x1242: file debug1.c, line 35.
```

Vår brytpunkt fick alltså nummer 1. Om vi senare vill stänga av den kan vi skriva `disable 1` (**dis 1**). Vi kan nu starta om programmet med kommandot `run` (**r**) som förut. GDB säger att den kommer stänga av instansen som körs just nu, och frågar om det är okej. Svara ja på frågan (**y**). GDB startar då om vårt program och skriver ut följande:

```
Breakpoint 1, main () at debug1.c:41
```

Detta innebär att programmet har stoppats vid brytpunkt nummer 1 som vi skapade nyss. Vi kan också notera att **b+** till vänster om radnumret har ändrats till **B+** för att indikera att programmet har stannat vid brytpunkten åtminstone en gång. Nu kan vi likt tidigare skriva `print i` (**p i**) för att se vad loopvariabeln har för innehåll. Första gången är den (föga förvånande) 0. Härifrån kan vi nu be GDB att fortsätta köra programmet med kommandot `continue` (**c**). GDB skriver då ut:

```
Continuing.
```

```
Breakpoint 1, main () at debug1.c:41
```

Detta innebär att vi återigen står vid brytpunkt 1. Vi kan verifiera att programmet är i nästa iteration av loopen genom att skriva ut `i` igen. Detta bör ge 1 den här gången. Eftersom vi kommer vilja skriva ut värdet på `i` många gånger kan vi be GDB att skriva ut det efter varje kommando. Detta kan vi göra med kommandot `display i` (**disp i**). GDB svarar då med följande information:

```
1: i = 1
```

Detta innebär att vårt uttryck fick ID 1, och att `i` hade värdet 1. Fortsätter vi i loopen med kommandot `continue` (**c**) så kommer GDB att skriva ut något i stil med följande varje gång:

```
Continuing.
```

```
Breakpoint 1, main () at debug1.c:41
```

```
1: i = 2
```

Om vi någon gång i framtiden inte längre vill se uttrycket hela tiden kan vi skriva `undisplay 1` (där 1 är ID:t) för att sluta visa uttrycket.

Fortsätter vi stega igenom loopen med `continue` (**c**) så kommer vi se att element nummer 5 aldrig fylls i innan vi kommer till den andra loopen. Det är alltså därför loopen kraschar. Nu kan vi avsluta GDB med kommandot `quit` (**q**) och lösa problemet på lämpligt sätt.

Det finns andra sätt att stega igenom koden än att sätta brytpunkter och låta programmet köra fram till dem. Här är de mest användbara:

- **step** (**s**): Stega till nästa rad kod som körs. Om markören står på ett funktionsanrop så kommer GDB stega in i funktionen (eftersom nästa rad som körs är inuti funktionen). Kallas ibland *step into*.

- **next (n)**: Stega till nästa rad kod som körs i den nuvarande funktionen. Hoppa alltså över eventuella funktionsanrop på nuvarande rad. Kallas ibland *step over*.
- **finish (fin)**: Stega till slutet av funktionen som nu körs.

Till sist kan det vara värt att veta att man oftast inte behöver starta om GDB om man har gjort ändringar till sitt program. Om man kompilerar programmet i en annan terminal, och sedan ber GDB att starta om programmet (med **run**) så kommer GDB att inse att programmet har blivit ändrat, och gör sitt bästa för att bevara brytpunkter och dylikt trots ändringarna. Däremot så misslyckas den ibland, och det händer att den kraschar, så ha förväntningarna på en lagom nivå.

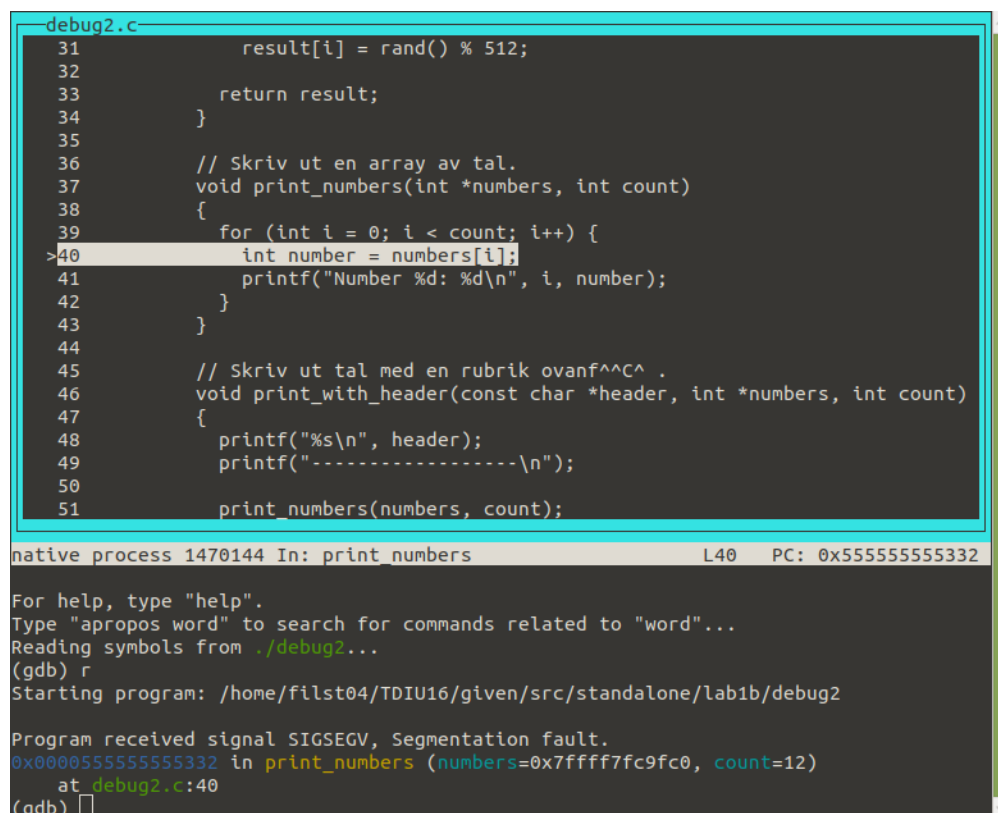
B.2 Program 2: debug2.c

Du kan kompilera programmet med kommandot **make debug2**. Du kan sedan köra programmet med kommandot **./debug2**. Du kommer då se att programmet skriver ut en lista med tal, sedan kraschar det med meddelandet **Segmentation fault** igen.

Denna gång består uppgiften av två delar:

- att lösa problemet så att programmet inte längre kraschar
- att skriva ut tabellen så att alla tal är högerjusterade (dvs. så att alla `:` är på en snygg rad, och sista siffran på varje rad är i samma kolumn)

Likt program 1 kan vi felsöka detta med GDB. Starta GDB med **gdb -tui ./debug2** och kör programmet tills det kraschar (med **run** eller **r**). Detta bör ge ett felmeddelande i stil med bilden nedan (efter att ha tryckt **Ctrl+L**):



```

debug2.c
31         result[i] = rand() % 512;
32
33         return result;
34     }
35
36     // Skriv ut en array av tal.
37     void print_numbers(int *numbers, int count)
38     {
39         for (int i = 0; i < count; i++) {
40             int number = numbers[i];
41             printf("Number %d: %d\n", i, number);
42         }
43     }
44
45     // Skriv ut tal med en rubrik ovanf^^C^ .
46     void print_with_header(const char *header, int *numbers, int count)
47     {
48         printf("%s\n", header);
49         printf("-----\n");
50     }
51     print_numbers(numbers, count);

```

```

native process 1470144 In: print_numbers          L40    PC: 0x555555555332

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./debug2...
(gdb) r
Starting program: /home/filst04/TDIU16/given/src/standalone/lab1b/debug2

Program received signal SIGSEGV, Segmentation fault.
0x0000555555555332 in print_numbers (numbers=0x7ffff7fc9fc0, count=12)
    at debug2.c:40
(gdb)

```

Här kan vi se att GDB skriver ut parametrarna till `print_numbers`, vilket kan vara användbart. Likt tidigare kan vi skriva ut värdet på `i` med `print i (p i)`:

```
$1 = 0
```

Då ser vi att `i` är 0. Vi kan skriva ut värdet inuti `numbers` för att se vad som står där. Gör vi det så får vi följande svar (adressen är antagligen en annan):

```
Cannot access memory at address 0x7ffff7fc9fc0
```

Detta tyder på att pekaren pekar på minne som inte är giltigt. Frågan är dock var pekaren kom ifrån. Med kommandot `backtrace (bt)` kan vi be GDB att visa en så kallad *stack trace*:

```
#0 0x0000555555555332 in print_numbers (numbers=0x7ffff7fc9fc0, count=12)
    at debug2.c:40
#1 0x00005555555553a0 in print_with_header (header=0x555555556032 "Second time:",
    numbers=0x7ffff7fc9fc0, count=12) at debug2.c:51
#2 0x0000555555555406 in main () at debug2.c:62
```

Här ser vi en numrerad lista med så kallade *stack frames*. Just nu är vi i frame nummer 0, eftersom vi står i `print_numbers`. Den anropades av koden i frame nummer 1, inuti `print_with_header`. Denna anropades i sin tur av `main`. Vi kan hoppa till tidigare frames med kommandot `frame (f)`. Går vi till frame 1 (`frame 1` eller `f 1`) så påminner GDB oss om var vi är:

```
#1 0x00005555555553a0 in print_with_header (header=0x555555556032 "Second time:",
    numbers=0x7ffff7fc9fc0, count=12) at debug2.c:51
```

Här kan vi se att `numbers` kommer direkt som en parameter till `print_with_header`, och att det enda som sker med `numbers` är att den skickas vidare till `print_numbers` (vi ser också att parametern `numbers` innehåller samma pekarvärde som vi såg i `print_numbers`). Det är alltså antagligen inget fel här. Vi går vidare till frame 2 (med `frame 2` eller `f 2`):

```
#2 0x0000555555555406 in main () at debug2.c:62
```

Här kan vi se att samma `numbers` skickas som parameter till båda anrop till `print_with_header`. Vi vet från tidigare att det första anropet lyckades, så vad hände däremellan? Vi lägger en brytpunkt på rad 59 (raden med första anropet till `print_with_header`) med `break 59 (b 59)` och startar om programmet med `run (r)`.

Nu kan vi se vad som händer med variabeln `numbers` när vi kör kod. Eftersom vi kommer att vilja se den hela tiden kan vi köra `display numbers` för att skriva ut adressen i variabeln. För att smidigt se vad den innehåller kan vi också lägga till `display numbers[0]`. Just nu borde GDB svara med något i stil med nedan (adresser och tal kan variera):

```
1: numbers = (int *) 0x7ffff7fc9fc0
2: numbers[0] = 299
```

Vi kan nu stega framåt i programmet med kommandot `next (n)`. När vi hamnar på raden `printf("\n");` får vi följande rapport från GDB (programmet skriver ut saker, så tryck på Ctrl+L för att se en korrekt vy):

```
1: numbers = (int *) 0x7ffff7fc9fc0
2: numbers[0] = <error: Cannot access memory at address 0x7ffff7fc9fc0>
```

Intressant. Adressen i `numbers` är densamma som tidigare (vilket är som förväntat), men vi kan inte längre läsa det första elementet. Något hände inuti `print_with_header`! Vi startar om programmet med `run (r)` för att komma tillbaka till vår brytpunkt. Denna gång stegar vi in i funktionen `print_with_header` med kommandot `step (s)` i stället. GDB skriver ut ett meddelande om att vi är inuti en annan funktion för att informera oss om detta. Eftersom vi nu är i en annan funktion så måste vi säga åt den att visa våra variabler igen (det kanske är något annat som är intressant i en annan funktion!). Vi skriver alltså `display numbers`

och `display numbers[0]` igen. Här borde vi se samma utdata som i `main` tidigare (men kanske med ett annat slumptal):

```
1: numbers = (int *) 0x7ffff7fc9fc0
2: numbers[0] = 299
```

Vi kan nu stega igenom hela `print_with_header` med kommandot `next (n)`. För att slippa skriva `n` hela tiden räcker det att köra kommandot en gång. Sedan kan man helt enkelt trycka på enter utan att skriva något för att köra föregående kommando igen. På så sätt kommer man snabbt och smidigt igenom hela funktionen. Notera att eftersom funktionen skriver ut saker kan man behöva trycka Ctrl+L ibland.

När vi stegar igenom funktionen så ser vi att utskriften av `numbers[0]` är korrekt ända fram tills efter anropet av `free`. Det är alltså anropet till `free` som är problemet!

Lös nu problemet på lämpligt sätt, och modifiera funktionen `print_numbers` så att den skriver ut talen högerjusterade.

B.3 Program 3: `debug3.c`

Du kan kompilera programmet med kommandot `make debug3`. Du kan sedan köra programmet med kommandot `./debug3`. Du kommer då se att programmet skriver meddelandet `Strings in C are fun!` och kraschar sedan med meddelandet `Segmentation fault` igen.

Likt tidigare så kör vi programmet i GDB med `gdb -tui ./debug3` och kör det med `run (r)`. Denna gång visas dock inte någon rad i vår källkod, utan GDB säger bara:

```
Program received signal SIGSEGV, Segmentation fault.
__strlen_avx2 () at ../sysdeps/x86_64/multiarch/strlen-avx2.S:141
../sysdeps/x86_64/multiarch/strlen-avx2.S: No such file or directory.
```

Det verkar som att funktionen `__strlen_avx2` kraschade när den försökte göra sin uppgift (det råkar vara en implementation av `strlen` i standardbiblioteket, den räknar hur lång en sträng är). GDB säger också att den inte hittade källkoden för funktionen, vilket är varför den inte visar något i den övre delen av skärmen. Vid första anblick skulle man kunna tro att vi har hittat en bugg i standardbiblioteket. Det är dock inte nödvändigtvis fallet, eftersom det är mer sannolikt att vi har använt standardbiblioteket på ett felaktigt sätt. För att se vad som kan vara fel kör vi `backtrace (bt)`. Då får vi följande svar från GDB:

```
#0 __strlen_avx2 () at ../sysdeps/x86_64/multiarch/strlen-avx2.S:141
#1 0x00007ffff7df1d15 in __vfprintf_internal (s=0x7ffff7f666a0 <_IO_2_1_stdout_>,
    format=0x5555555556028 "Copy:      %s\n", ap=ap@entry=0x7ffff7fffd5c0,
    mode_flags=mode_flags@entry=0) at vfprintf-internal.c:1688
#2 0x00007ffff7ddad3f in __printf (format=<optimized out>) at printf.c:33
#3 0x00005555555552ca in main () at debug3.c:38
```

Detta är en lista över så kallade *stack frames*. Varje frame motsvarar ett anrop till en funktion. Listan kan se lite kryptisk ut till en början. En bra idé är att leta efter funktionsnamn i raderna efter varje nummer. Här ser vi att frame 0 motsvarar anropet till funktionen `__strlen_avx2` där programmet kraschade. Frame 1 är då funktionen som anropade `__strlen_avx2`, vilket här är funktionen `_vfprintf_internal`. Frame 2 motsvarar i sin tur `__printf` (vilket råkar vara det interna namnet på `printf`), och frame 3 motsvarar `main`.

En annan observation som blir relevant i Pintos senare är att parametern till frame nummer 2 (`__printf`) bara säger `<optimized out>`. Detta händer ibland när man kör med optimeringsflaggor. Då kan kompilatorn ha beslutat att ett värde på en viss variabel inte behövs på ett visst ställe i koden, och därmed kan GDB inte visa det. Det kan också hända när man använder `print`-kommandot eller `display`-kommandot.

När man ser den här typen av fel så är det en bra idé att gå igenom frames tills man hittar kod som man själv har skrivit och börja där. Exempelvis så kan vi börja med att skriva `frame 1 (f 1)` och se om vi känner

igen koden där. Sedan försöka med **frame 2** (**f 2**) och så vidare tills vi känner igen oss. I just det här fallet så kommer vi inte se den relevanta källkoden förrän vi kommer till **frame 3**, då vi är i **main** på raden när programmet försöker skriva ut **copy** genom att anropa **printf**.

Eftersom programmet kraschade när det försökte skriva ut **copy** kan det vara något problem där. Vi försöker att skriva ut variabeln med **print copy** (**p copy**):

```
$2 = 0x7ffff7fc9fe0 "Strings in C are fun!\314\314\314\314\314\314\314\314",
<incomplete sequence \314>
```

Här ser vi att GDB är snäll och tolkar variabeln som en C-sträng. Vi ser också att början av strängen är korrekt, men att det är skräp i slutet av strängen. Vi ser också att GDB säger **<incomplete sequence \314>** i slutet. Detta innebär är att den inte hittade tecknet 0 innan den kom fram till ogiltigt minne. Eftersom tecknet 0 används för att indikera slutet av en sträng i C betyder det antagligen att **strlen** (som anropades av **printf**) försökte leta efter tecknet 0, men kraschade innan den hittade det tecknet. Det verkar alltså som att koden i **my_strdup** har misslyckats med att lägga in ett 0-tecken.

Lös probelemet på lämpligt sätt.

B.4 GDB i Pintos

Nedan följer en kort förklaring av hur GDB används i Pintos. Det är inte en del av denna del av labben, men finns här för att visa hur det fungerar. Som nämnt ovan så finns ytterligare information om hur felsökning i Pintos sker i Pintos-Wiki. Här följer en kort sammanfattning:

1. Starta GDB med kommandot **pintos-gdb -tui build/kernel.o** (givet att du står i **src/userprog**)
2. Starta Pintos i en annan terminal. Ersätt kommandot **pintos** i den kommandorad du körde med **debugpintos**, lämna resterande argument som de var. Om du bara vill testa att starta GDB kan du använda kommandot **debugpintos --fs-disk=2 -- -f -q**.
3. Koppla ihop GDB med Pintos genom att skriva **debugpintos** i GDB-prompten.

Värt att notera är att i steg 2 så kommer kommandot **debugpintos** starta en virtuell maskin som är redo att köra Pintos. Den kommer dock inte att börja köra Pintos förrän GDB har anslutits.

I och med att GDB inte kan starta Pintos hur som helst så fungerar inte kommandot **run** som ovan. I stället får man hjälpa GDB genom att starta om Pintos manuellt. Detta kan göras genom att koppla ifrån Pintos med kommandot **kill (k)** i GDB, starta om Pintos i den andra terminalen (med samma **debugpintos**-kommando), och till sist köra kommandot **debugpintos** inuti GDB igen. På så sätt kommer GDB att komma ihåg dina brytpunkter, och de variablerna du visade med **display**.

Del C Associativ databehållare

I ett operativsystem behöver man ofta hålla reda på olika resurser och ge dem "namn" i form av heltal. Därför ska du i denna del implementera en modul som fungerar som en associativ behållare för detta ändamål. Vi kommer exempelvis att använda den för att hålla koll på fildeskriptorer i nästa laboration.

För att illustrera detta, tänk dig att vi vill lagra strängar (`const char *`) i behållaren. Vi vill då kunna göra följande:

```
int main(void) {
    struct map m;
    map_init(&m);                // initiera behållaren
    int a = map_insert(&m, "hello"); // returnerar exempelvis 0
    int b = map_insert(&m, "world"); // returnerar exempelvis 1
    map_find(&m, a);              // returnerar "hello"
    map_find(&m, b);              // returnerar "world"
    map_remove(&m, a);            // returnerar "hello" och tar bort elementet
    map_find(&m, a);              // returnerar NULL
}
```

I mappen `src/standalone/lab1c` finns det ett testprogram för detta, samt filer `map.h` och `map.c` för din implementation. I `map.h` är det en bra idé att inkludera `#include <stdbool.h>` för att kunna använda booleans som i C++.

Notera: Tanken med uppgiften är att öva på programmering i C utanför Pintos. Detta gör vi genom att bygga och testa en enkel datastruktur som vi sedan har nytta av i Pintos. Tanken är *inte* att datastrukturen ska vara särskilt komplicerad. Tvärt om är det en bra idé att hålla implementationen så *enkel* som möjligt. Det gör det enklare att fokusera på det som är nytt i kursen — programmering i en kernel, systemanrop, och synkronisering. Sikta på en implementation som är så enkel att du enkelt kan se att den är korrekt genom att titta på koden. En sådan implementation bör inte ta mer än ett par timmar att implementera och testa.

C.1 Funktionalitet

Din modul ska innehålla en datastruktur som heter `struct map`. Den ska ha operationerna som beskrivs nedan. Dessa operationer tar alla en pekare till en instans av datastrukturen. För enkelhets skull använder vi definitionerna `key_t` och `value_t` för att benämna nyckel- och värdetyper. Du kan dock anta att `key_t` alltid kommer vara en heltalstyp (dvs. `int`, `long`, eller liknande), och att `value_t` kommer vara någon form av pekare (ex.vis `char *` eller `struct file *`). De kan definieras som nedan:

```
typedef char* value_t;
typedef int key_t;
```

Funktionerna ska bete sig enligt följande:

- `void map_init(struct map* m);`

Denna funktion motsvarar en konstruktor i C++. I C måste den anropas manuellt varje gång en variabel av typen `struct map` skapas. Ansvar för detta ligger på användaren av modulen. `map_init` har därmed till uppgift att initiera alla medlemmar i `struct map`, så att den motsvarar en tom container. Kom ihåg: när variabeln skapas så finns inga garantier på innehållet i dess medlemmar. Vi kan alltså inte ens anta att alla datamedlemmar är 0.

- `key_t map_insert(struct map* m, value_t v);`

En funktion för att sätta in ett nytt värde i behållaren. För att senare kunna hitta värdet igen ska funktionen hitta en nyckel som inte tidigare användes och returnera den. Denna nyckel ska sedan kunna användas för att hitta värdet igen.

- `value_t map_find(struct map* m, key_t k);`

Denna funktion ska hämta ett värde som tidigare är insatt med `map_insert`. Om värdet hittas ska det returneras. Annars ska `NULL` returneras.

- `value_t map_remove(struct map* m, key_t k);`

Denna funktion ska ta bort elementet med nyckeln *k* ur datastrukturen. Fungerar som `map_find` med skillnaden att den också tar bort elementet.

Utöver ovanstående funktioner kommer det att finnas behov av lite mer avancerad funktionalitet. Detta går att implementera generellt i C med hjälp av funktionspekare (jämför med lambdafunktioner eller funktionsobjekt i C++). All funktionalitet som behövs i labbserien går att implementera med hjälp av de två generella funktionerna nedan, men det går också bra att göra speciella funktioner för de specifika fall som krävs senare.

- `void map_for_each(struct map* m,
 void (*exec)(key_t k, value_t v, int aux),
 int aux);`

Funktionen ska iterera igenom alla element i datastrukturen och anropa funktionen *exec* för varje värde som finns. Parameter 2 är en pekare till en funktion som tar 3 parametrar: en nyckel, ett värde och ett godtyckligt heltal. Den tredje parametern (*aux*) är det värde som skickas med till *exec* vid varje anrop. Exempelprogrammet använder detta för att veta vilken delmängd av nycklarna som ska skrivas ut.

Tittar vi på andra parametern ser vi att det är en pekare till en funktion som tar tre parametrar och returnerar *void*. Just denna syntax ser lite struligt ut i C, men tänk bort parenteserna runt ordet *exec* och stjärnan framför så ser du att det är en normal funktionsdeklaration man gjort om till pekare.

- `void map_remove_if(struct map* m,
 bool (*cond)(key_t k, value_t v, int aux),
 int aux);`

Funktionen ska fungera precis som `map_for_each`, men om *cond* returnerar *true* så ska elementet tas bort från datastrukturen. Detta är bra om endast vissa associationer ska tas bort, eller om extra steg för att exempelvis frigöra minne krävs innan borttagning. Exempelprogrammet använder detta för att frigöra minne innan elementen frigörs.

C.2 Implementation

Nedan följer två idéer för hur datastrukturen kan implementeras:

Alternativ 1

Det enklaste alternativet är att implementera `map` som en array med en förbestämd storlek. Detta gör det väldigt enkelt att hantera i C (man behöver inte tänka så mycket på minneshantering), men det har såklart problemet att den inte kan växa dynamiskt.

I och med att vi själva bestämmer vad för nyckel som insatta element får, så kan vi helt enkelt bestämma att ett element som är på plats 3 i arrayen får nyckeln 3. Detta gör uppslagning trivialt (titta på rätt plats i arrayen, med relevanta felkontroller innan). Insättning består då av att helt enkelt hitta en ledig plats i arrayen.

I C kan detta implementeras som följer:

```
/* symbolisk konstant för att enkelt kunna ändra storleken senare vid behov */  
#define MAP_SIZE 32
```

```
struct map
{
    value_t content[MAP_SIZE];
}
```

Alternativ 2

Om man inte vill begränsa storleken i förväg kan man i stället använda den länkade lista som finns i Pintos (uppgift X3). Nackdelen är att man måste tänka lite mer på minneshantering i och med att alla noder allokeras dynamiskt. Detta kan implementeras enligt följande:

```
struct map_element
{
    key_t    key; /* nyckeln */
    value_t  value; /* värdet associerat med nyckeln */
    /* list-element för att kunna sätta in i listan */
    struct list_elem elem;
}

struct map
{
    /* listan med alla lagrade element */
    struct list content;

    /* räknare för vilken nyckel som är nästa lediga */
    int next_key;
}
```

Vanliga fel och funderingar

Här finns några vanliga frågor gällande den här labben. Se också "Vanliga fel och funderingar" i Pintos-wikin för generella fel.

Minnesläcka i den givna main Programmet är inte helt färdigt (i och med att `map` inte är helt klar), så titta noga på kommentarerna kring `YOUR CODE` i källkoden, och hur programmet försöker frigöra minnet i slutet av programmet. Fundera på vilken del (`map`-strukturen, eller användaren av den) av programmet som har ansvaret för att frigöra minnet.