

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. Eftersom Pintos körs inuti en virtuell maskin krävs några extra steg för att starta GDB, 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 `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.

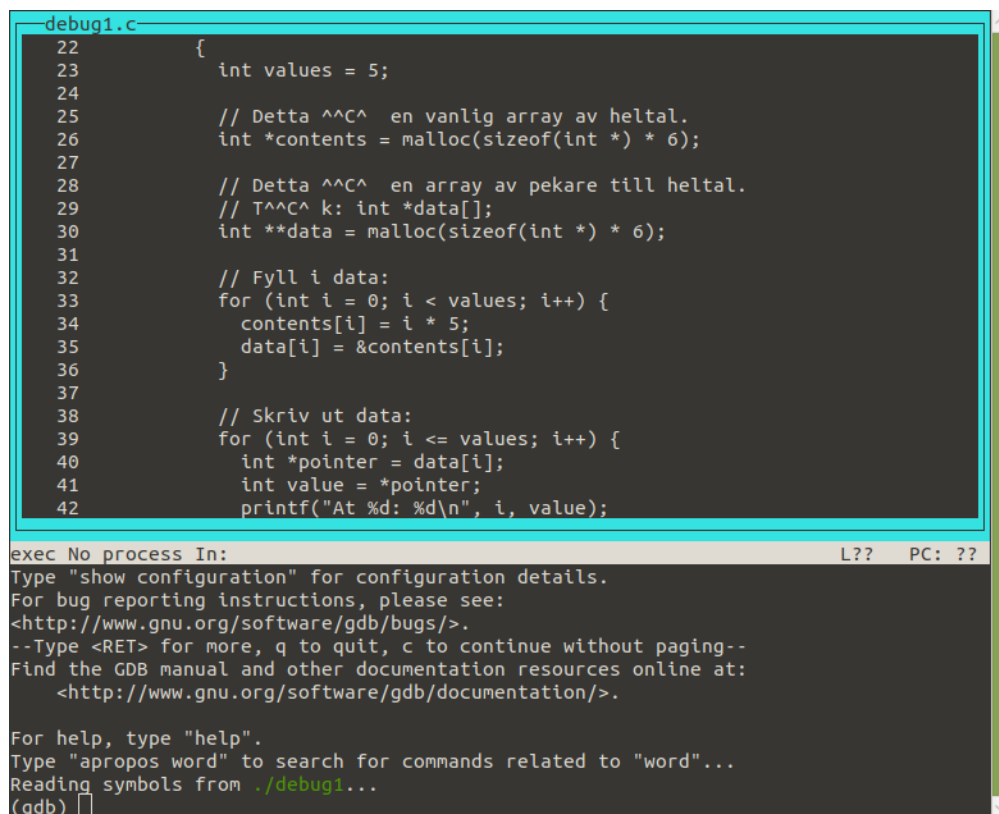
Notera: I den här delen används en version av funktionerna `malloc` och `free` som gör att programmet kraschar snabbare vid minnesfel. Detta gör felsökningen enklare i den här uppgiften. Detta betyder dock **inte** att du kan lita på att ditt program kraschar direkt ifall du råkar göra samma fel. Det 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
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
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2599
2600
2601
2602
2603

```

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      >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: 0x555555552a1
<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.
0x0000555555552a1 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 `0x0000555555552a1`. 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=0x555555556028 "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

Eftersom Pintos körs i en virtuell maskin kan vi inte bara köra **gdb -tui pintos**. Det skulle instruera GDB att felsöka det program som startar Pintos i en virtuell maskin. Vi måste i stället göra på ett litet annat sätt när vi startar Pintos och GDB för att kunna felsöka det som körs *inuti* den virtuella maskinen.

Nedan följer en kort beskrivning av hur detta sker. Det är inte en del av denna laboration, men det finns här för att belysa att uppstarten är lite annorlunda. 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 **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 --filesys-size=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 `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.