

Ukazatele, paměťové třídy, volání funkcí

Jan Faigl

Katedra počítačů
Fakulta elektrotechnická
České vysoké učení technické v Praze

Přednáška 05

BAB36PRGA – Programování v C

Přehled témat

- Část 1 – Ukazatele a dynamická alokace
Modifikátor `const` a ukazatele

Dynamická alokace paměti

S. G. Kochan: kapitoly 8 a 11

- Část 2 – Paměťové třídy a volání funkcí
Výpočetní prostředky a běh programu

Rozsah platnosti proměnných

Paměťové třídy

S. G. Kochan: kapitola 8 a 11

- Část 3 – Zadání 4. domácího úkolu (HW4)

Modifikátor `const` a ukazatele

Dynamická alokace paměti

Část I

Část 1 – Ukazatele a dynamická alokace

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 1 / 47

Modifikátor `const` a ukazatele Dynamická alokace paměti

Modifikátor typu `const`

- Uvedením klíčového slova `const` můžeme označit proměnnou jako konstantu.

Překladač nás kontroluje, zdali se snažíme hodnotu proměnné změnit.

- Definovat konstantu můžeme např.

```
const float pi = 3.14159265f;
```

- Symbolická konstanta

```
#define PI 3.14159265
```

- je pojmenování literálu, ve zdrojovém souboru je výkyt `PI` textově nahrazen literálem.

Přípomínka

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 5 / 47

Modifikátor `const` a ukazatele Dynamická alokace paměti

Příklad – Konstantní ukazatel

- Hodnotu konstantního ukazatele nelze po inicializaci měnit.
- Zápis `int *const ptr`; můžeme číst zprava doleva:

- `ptr` – proměnná, která je;
- `*const` – konstantním ukazatelem;
- `int` – na proměnnou typu `int`.

```
1 int v = 10;
2 int v2 = 20;
3 int *const ptr = &v;
4 printf("v: %d *ptr: %d\n", v, *ptr);
6 *ptr = 11; /* We can modify addressed value */
7 printf("v: %d\n", v);
9 ptr = &v2; /* IT IS NOT ALLOWED! */
```

lec05/const_pointers.c

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 8 / 47

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 2 / 47

Modifikátor `const` a ukazatele Dynamická alokace paměti

Ukazatele na konstantní proměnné a konstantní ukazatele

- Klíčové slovo `const` můžeme zapsat před jméno proměnné nebo před `*` (typ/).

- Dostáváme 3 možnosti jak definovat ukazatel s `const`.

(a) `const int *ptr`; – ukazatel na konstantní proměnnou.

- Nemůžeme použít pointer pro změnu hodnoty proměnné.

(b) `int *const ptr`; – konstantní ukazatel (`const` před jménem proměnné a mezi `*`).

- Pointer nemůžeme nastavit na jinou adresu než tu při inicializaci.

(c) `const int *const ptr`; – konstantní ukazatel na konstantní hodnotu.

- Kombinuje předchozí dva případy.

lec05/const_pointers.c

Další alternativy zápisu (a) a (c) jsou

- `const int *` lze též zapsat jako `int const *`;

*const je stále před *.*

- `const int * const` lze též zapsat jako `int const * const`.

const může být vlevo nebo vpravo od jména typu.

- Nebo komplexnější definice, např. `int ** const ptr`; – konstantní ukazatel na ukazatel na `int`.

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 6 / 47

Modifikátor `const` a ukazatele Dynamická alokace paměti

Příklad – Konstantní ukazatel na konstantní proměnnou

- Hodnotu konstantního ukazatele na konstantní proměnnou nelze po inicializaci měnit a ani nelze prostřednictvím takového ukazatele měnit hodnotu adresované proměnné.

- Zápis `const int *const ptr`; čteme "zprava doleva":

- `ptr` – proměnná, která je;
- `*const` – konstantním ukazatelem;
- `const int` – na proměnnou typu `const int`.

```
1 int v = 10;
2 int v2 = 20;
3 const int *const ptr = &v;
5 printf("v: %d *ptr: %d\n", v, *ptr);
7 ptr = &v2; /* IT IS NOT ALLOWED! */
8 *ptr = 11; /* IT IS NOT ALLOWED! */
```

lec05/const_pointers.c

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 9 / 47

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 3 / 47

Modifikátor `const` a ukazatele Dynamická alokace paměti

Příklad – Ukazatel na konstantní proměnnou (hodnotu)

- Prostřednictvím ukazatele na konstantní proměnnou nemůžeme tuto proměnnou měnit.

```
1 int v = 10;
2 int v2 = 20;
4 const int *ptr = &v; // ptr cannot be used to modify v
5 printf("*ptr: %d\n", *ptr);
7 *ptr = 11; /* IT IS NOT ALLOWED! */
9 v = 11; /* We can modify the original variable */
10 printf("*ptr: %d\n", *ptr);
12 ptr = &v2; /* We can assign new address to ptr */
13 printf("*ptr: %d\n", *ptr);
```

lec05/const_pointers.c

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 7 / 47

Modifikátor `const` a ukazatele Dynamická alokace paměti

Konstantní ukazatel (na konstantní hodnotu)

Příklad	Konstantní hodnota	Konstantní ukazatel	Popis „Čtu zprava doleva.“
<code>char *ptr</code>	Ne	Ne	„ <code>ptr</code> je ukazatel (<code>*</code>) na hodnotu <code>char</code> .“
<code>const char *ptr</code>	Ano	Ne	„ <code>ptr</code> je ukazatel na hodnotu <code>char</code> konstantní.“
<code>char const *ptr</code>	Ano	Ne	„ <code>ptr</code> je ukazatel na konstantní hodnotu <code>char</code> .“
<code>char* const ptr</code>	Ne	Ano	„ <code>ptr</code> je konstantní ukazatel na hodnotu <code>char</code> .“
<code>const char *const ptr</code>	Ano	Ano	„ <code>ptr</code> je konstantní ukazatel na hodnotu <code>char</code> konstantní.“

- Konstantní ukazatel je proměnná, jejíž hodnotu nemohu měnit. Ukazatel odkazuje na (stejně) paměťové místo, které mohu případně měnit.
- Konstantní hodnotu nemohu měnit. Tedy nemohu měnit obsah paměťového místa, na které odkazuje ukazatel (jehož adresa je uloženo v proměnné typu ukazatel).

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 10 / 47

Modifikátor const a ukazatele Dynamická alokace paměti

Ukazatel na funkci

- Implementace funkce je umístěna někde v paměti a podobně jako na proměnnou v paměti může ukazatel odkazovat na paměťové místo s definicí funkce.
- Můžeme definovat **ukazatel na funkci** a dynamicky volat funkci dle aktuální hodnoty ukazatele.
- Součástí volání funkce jsou předávané argumenty, které jsou též součástí typu ukazatele na funkci, resp. typu argumentů.
- Funkce (a volání funkce) je identifikátor funkce a `()`, tj. `typ_návratové_hodnoty funkce(argumenty funkce)`;
- Ukazatel na funkci definujeme jako `typ_návratové_hodnoty (*ukazatel)(argumenty funkce)`;

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 11 / 47

Modifikátor const a ukazatele Dynamická alokace paměti

Příklad – Ukazatel na funkci 1/2

- Používáme dereferenční operátor `*` podobně jako u proměnných.


```
double do_nothing(int v); /* function prototype */
double (*function_p)(int v); /* pointer to function */
function_p = do_nothing; /* assign the pointer */
(*function_p)(10); /* call the function */
```
- Závorky `(*function_p)` „pomáhají“ číst definici ukazatele.

Můžeme si představit, že závorky reprezentují jméno funkce. Definice proměnné ukazatel na funkci se tak v zásadě neliší od prototypu funkce.
- Podobně je volání funkce přes ukazatel na funkci identické běžnému volání funkce, kde místo jména funkce vystupuje jméno ukazatele na funkci.

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 12 / 47

Modifikátor const a ukazatele Dynamická alokace paměti

Příklad – Ukazatel na funkci 2/2

- V případě funkce vracějící ukazatel postupujeme identicky.


```
double* compute(int v);
double* (*function_p)(int v);
function_p = compute;
```

----- substitute a function name
- Příklad použití ukazatele na funkci – `lec05/pointer_fnc.c`
- Ukazatele na funkce umožňují realizovat dynamickou vazbu volání funkce identifikované za běhu programu.

V objektové orientovaném programování je dynamická vazba klíčem k realizaci polymorfiamu.

Ukazatel na funkci se může hodit v implementaci HW4 povinné a bonusové zadání. Při vhodném návrhu programu je základní část společná, „jen“ zaměníme funkci pro porovnávání dvou řetězců s využitím Hammingovy nebo Levenštejnovy vzdálenosti. V případě obou funkcí může být vstup dva textové řetězce, případně včetně délky. Tedy můžeme jednoduše zaměnit ukazatel na funkci.

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 13 / 47

Modifikátor const a ukazatele Dynamická alokace paměti

Příklad použití ukazatele na funkci

- Vhodným využitím ukazatele na funkci je zajištění přístupu k datům pro jinak naprosto identický algoritmus, jako je řazení (funkce `qsort` z `stdlib.h`). *Zejména pro pole hodnot složeného typu.*

```
void qsort(void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *));
```

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 void print(int n, int array[n]);
4 int compare(const void *pa, const void *pb);
5 int main(void)
6 {
7     const int n = 10;
8     int array[n];
9     for (int i = 0; i < n; ++i) {
10        array[i] = rand() % 100;
11    }
12    print(n, array);
13    qsort(array, n, sizeof(array[0]), compare);
14    print(n, array);
15    return 0;
16 }
```

```
20 void print(int n, int array[n])
21 {
22     for(int i = 0; i < n; ++i) {
23         i > 0 ? printf(", ") : 0;
24         printf("%d", array[i]);
25     }
26     n > 0 ? putchar('\n') : 0;
27 }
28 {
29     const int a = *(int*)pa;
30     const int b = *(int*)pb;
31     return (a < b) - (a > b);
32 }
```

lec05/demo-pointer_fnc.c

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 14 / 47

Modifikátor const a ukazatele Dynamická alokace paměti

Definice typu – typedef

- Operátor `typedef` umožňuje definovat nový datový typ.
- Slouží k pojmenování typů, např. ukazatele, struktury a uniony.

Struktury a uniony viz přednáška 6.
- Například typ pro ukazatele na `double` a nové jméno pro `int`:


```
1 typedef double* double_p;
2 typedef int integer;
3 double_p x, y;
4 integer i, j;
```
- je totožné s použitím původních typů


```
1 double *X, *Y;
2 int i, j;
```
- Zavedením typů operátorem `typedef`, např. v hlavičkovém souboru, umožňuje systematické používání nových jmen typů v celém programu. *Viz např. <inttypes.h>.*
- Výhoda zavedení nových typů je především u složitějších typů jako jsou ukazatele na funkce nebo struktury.

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 15 / 47

Modifikátor const a ukazatele Dynamická alokace paměti

Dynamická alokace paměti

- Přidělení bloku paměti velikosti `size` lze realizovat funkcí `void* malloc(size)`; `Z knihovny <stdlib.h>`
 - Velikost alokované paměti je uložena ve správci paměti.
 - Velikost není součástí ukazatele.**
 - Návratová hodnota je typu `void*` – přetytování nutné/vhodné.
 - Je plně na uživateli (programátorovi), jak bude s pamětí zacházet.**
- Příklad alokace paměti pro 10 proměnných typu `int`.


```
1 int *int_array;
2 int_array = (int*)malloc(10 * sizeof(int));
```
- Operace s více hodnotami v paměťovém bloku je podobná poli.
 - Používáme pointerovou aritmetiku.
- Uvolnění paměti** `void free(pointer)`;
 - Správce paměti uvolní paměť asociovanou k ukazateli.
 - Hodnotu ukazatele však nemění!

Stále obsahuje předešlou adresu, která však již není platná.

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 17 / 47

Modifikátor const a ukazatele Dynamická alokace paměti

Příklad alokace dynamické paměti 1/3

- Alokace se nemusí nutně povést – testujeme návratovou hodnotu funkce `malloc()`.
- Pro vyplnění adresy alokované paměti předáváme proměnnou jako ukazatel na proměnnou typu ukazatel na `int`.


```
1 void* allocate_memory(int size, void **ptr)
2 {
3     // use **ptr to store value of newly allocated
4     // memory in the pointer ptr (i.e., the address the
5     // pointer ptr is pointed).
6     // call library function malloc to allocate memory
7     *ptr = malloc(size);
8     if (*ptr == NULL) {
9         fprintf(stderr, "Error: allocation fail");
10        exit(-1); /* exit program if allocation fail */
11    }
12    return *ptr;
13 }
```

lec05/malloc_demo.c

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 18 / 47

Modifikátor const a ukazatele Dynamická alokace paměti

Příklad alokace dynamické paměti 2/3

- Pro vyplnění hodnot pole alokovaného dynamicky nám postačuje předávat hodnotu adresy paměti pole.


```
1 void fill_array(int size, int* array)
2 {
3     for (int i = 0; i < size; ++i) {
4         *(array++) = random();
5     }
6 }
```
- Po uvolnění paměti odkazuje ukazatel stále na původní adresu, proto můžeme explicitně nulovat.

Předání ukazatele na ukazatele je nutné, jinak nemůžeme nulovat.

```
1 void deallocate_memory(void **ptr)
2 {
3     if (ptr != NULL && *ptr != NULL) {
4         free(*ptr);
5         *ptr = NULL;
6     }
7 }
```

lec05/malloc_demo.c

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 19 / 47

Modifikátor const a ukazatele Dynamická alokace paměti

Příklad alokace dynamické paměti 3/3

```
1 int main(int argc, char *argv[])
2 {
3     int *int_array;
4     const int size = 4;
5     allocate_memory(sizeof(int) * size, (void*)&int_array);
6     fill_array(int_array, size);
7     int *cur = int_array;
8     for (int i = 0; i < size; ++i, cur++) {
9         printf("Array[%d] = %d\n", i, *cur);
10    }
11    deallocate_memory((void*)&int_array);
12    return 0;
13 }
```

lec05/malloc_demo.c

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 20 / 47

Modifikátor const a ukazatele Dynamická alokace paměti

Příklad - Načítání textového řetězce 1/3

- Implementujete načtení libovolně dlouhého řádku ze `stdin`.
- Řádek je zakončen znakem nového řádku `'\n'`, který **není součástí načteného vstupu**.
- Reportujte chybové stavy `ERROR_IN = 100` a `ERROR_MEM = 101`.
- Po úspěšném načtení vstupu, reportujte velikost vstupu voláním funkce `strlen()` z `string.h`.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #ifndef INIT_SIZE
5 #define INIT_SIZE 128
6 #endif
7 enum {
8     ERROR_OK = EXIT_SUCCESS,
9     ERROR_IN = 100,
10    ERROR_MEM = 101,
11 };
12 char* read(int *error);
13 char* enlarge_string(size_t len, size_t
14 *capacity, char *str);
15
16 int main(int argc, char *argv[])
17 {
18     int ret = EXIT_SUCCESS;
19     char *str = read(&ret);
20     if (str) {
21         printf("Input string size %ld\n", strlen(str));
22         free(str);
23     } else {
24         fprintf(stderr, "ERROR: read return %d\n", ret);
25     }
26     return ret;
27 }
28
29 #include <stdio.h>
30 #include <stdlib.h>
31 #include <string.h>
32 static char* handle_str(char r, size_t l,
33 char *str, int *error);
34 char* read(int *error)
35 {
36     size_t capacity = INIT_SIZE;
37     size_t l = 0; // no. of read chars
38     char* str = malloc(capacity + 1);
39     int r = '\0';
40     while (
41         str
42         && *error == ERROR_OK
43         && (r = getchar()) != EOF
44         && r != '\n'
45     ) {
46         if (l == capacity) { // enlarge if need
47             // new address of str can be set
48             str = enlarge_string(l, &capacity, str);
49         }
50         //Is it correct? Can str be NULL?
51         str[l++] = r;
52     } // end while
53     str = handle_str(r, l, str, error);
54     return str;
55 }
56
57 char* handle_str(char r, size_t l, char *str, int *error)
58 {
59     if (str) {
60         if (r != '\n') { // end-of-line has not been read
61             *error = ERROR_IN; // report input error
62             free(str);
63             str = NULL;
64         } else {
65             str[l] = '\0'; // null terminating string
66         }
67     } else if (*error == ERROR_OK) { // str is NULL
68         *error = ERROR_MEM; // but error needs to be set
69     }
70     return str;
71 }
72
73 char* enlarge_string(size_t len, size_t *capacity, char *str)
74 {
75     char *t = realloc(str, *capacity * 2 + 1);
76     if (!t) {
77         free(str);
78         str = NULL; // indicate error
79     } else {
80         str = t;
81         *capacity *= 2;
82     }
83     return str;
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

```

lec05/read.c

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 21 / 47

Modifikátor const a ukazatele Dynamická alokace paměti

Příklad - Načítání textového řetězce 2/4

```

31 // local function only for calling from read()
32 static char* handle_str(char r, size_t l,
33 char *str, int *error);
34 char* read(int *error)
35 {
36     size_t capacity = INIT_SIZE;
37     size_t l = 0; // no. of read chars
38     char* str = malloc(capacity + 1);
39     int r = '\0';
40     while (
41         str
42         && *error == ERROR_OK
43         && (r = getchar()) != EOF
44         && r != '\n'
45     ) {
46         if (l == capacity) { // enlarge if need
47             // new address of str can be set
48             str = enlarge_string(l, &capacity, str);
49         }
50         //Is it correct? Can str be NULL?
51         str[l++] = r;
52     } // end while
53     str = handle_str(r, l, str, error);
54     return str;
55 }
56
57 char* handle_str(char r, size_t l, char *str, int *error)
58 {
59     if (str) {
60         if (r != '\n') { // end-of-line has not been read
61             *error = ERROR_IN; // report input error
62             free(str);
63             str = NULL;
64         } else {
65             str[l] = '\0'; // null terminating string
66         }
67     } else if (*error == ERROR_OK) { // str is NULL
68         *error = ERROR_MEM; // but error needs to be set
69     }
70     return str;
71 }
72
73 char* enlarge_string(size_t len, size_t *capacity, char *str)
74 {
75     char *t = realloc(str, *capacity * 2 + 1);
76     if (!t) {
77         free(str);
78         str = NULL; // indicate error
79     } else {
80         str = t;
81         *capacity *= 2;
82     }
83     return str;
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

```

lec05/read.c

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 22 / 47

Modifikátor const a ukazatele Dynamická alokace paměti

Příklad - Načítání textového řetězce 3/4

- Příklad vstupu programu `clang read.c -o read`.
- Vstup soubor `read-in-1.txt`.

```

./read <read-in-1.txt; echo $?
Input string size 11
0
hexdump -C read_in-1.txt
00000000 49 20 6c 69 6b 65 20 70 72 67 21 0a
0000000c                                     |I like prg!.|
                                         lec05/read_in-1.txt

```

- Vstup soubor `read-in-2.txt`.

```

./read <read_in-2.txt; echo $?
ERROR: read return 100
100
hexdump -C read_in-2.txt
00000000 49 20 6c 69 6b 65 20 70 72 67 21
0000000b                                     |I like prg!|
                                         lec05/read_in-2.txt

```

lec05/read.c

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 23 / 47

Modifikátor const a ukazatele Dynamická alokace paměti

Příklad - Načítání textového řetězce 4/4

- Generování náhodného vstupu.

```

cat /dev/urandom | env LC_ALL=C tr -dc 'a-zA-Z0-9' | fold -w 10485760 | head -n 1

```

- Omezení paměti programu.

```

clang read.c -o read
./create_rand_string.sh >10MB.txt
du -h 10MB.txt
10M 10MB.txt
./read <10MB.txt
Input string size 10485760

```

lec05/create_rand_string.sh

lec05/read.c

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 24 / 47

Výpočetní prostředky a běh programu Rozsah platnosti proměnných Paměťové třídy

Část II

Část 2 – Paměťové třídy, model výpočtu

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 25 / 47

Výpočetní prostředky a běh programu Rozsah platnosti proměnných Paměťové třídy

Paměť počítače s uloženým programem v operační paměti

- Posloupnost instrukcí je čtena z operační paměti.
- Flexibilita ve tvorbě posloupnosti.
- Architektura počítače se společnou pamětí pro data a program.
- von Neumannova architektura počítače
- sdílí program i data ve stejné paměti.
- Adresa aktuálně prováděné instrukce je uložena v tzv. čítači instrukcí (Program Counter PC).
- Mimoto architektura se sdílenou pamětí umožňuje, aby hodnota ukazatele odkazovala nejen na data, ale také například na část paměti, kde je uložen program (funkce).

Princip ukazatele na funkci.

PC → PAMĚŤ

Program: posloupnost instrukcí

Data: hodnoty proměnných

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 27 / 47

Výpočetní prostředky a běh programu Rozsah platnosti proměnných Paměťové třídy

von Neumannova architektura

V drtivě většině případů je program posloupnost instrukcí zpracovávající jednu nebo dvě hodnoty (uložené na nějakém paměťovém místě) jako vstup a generování nějaké výstupní hodnoty, kterou ukládá někam do paměti nebo modifikuje hodnotu PC (podmíněně řízení běhu programu).

- ALU - Aritmeticko logická jednotka (Arithmetic Logic Unit)

Základní matematické a logické instrukce

- PC obsahuje adresu kódu – při volání funkce tak jeho hodnotu můžeme uložit (na zásobník) a následně použít pro návrat na původní místo volání.

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 28 / 47

Výpočetní prostředky a běh programu Rozsah platnosti proměnných Paměťové třídy

Základní rozdělení paměti

- Přidělenou paměť programu můžeme kategorizovat na 5 částí.
- Zásobník – lokální proměnné, argumenty funkcí, návratová hodnota funkce. Spravováno automaticky.
- Halda – dynamická paměť (malloc(), free()). Spravuje programátor.
- Statická – globální nebo „lokální“ statické proměnné. Inicializováno při startu.
- Literály – hodnoty zapsané ve zdrojovém kódu programu, např. textové řetězce. Inicializováno při startu.
- Program – strojové instrukce. Inicializováno při startu.

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 29 / 47

Výpočetní prostředky a běh programu Rozsah platnosti proměnných Paměťové třídy

Rozsah platnosti (scope) lokální proměnné

- Lokální proměnné mají rozsah platnosti pouze uvnitř bloku a funkce.
- Globální proměnné mají rozsah platnosti „kdekoliv“ v programu.
- Zastíněný přístup lze řešit modifikátorem `extern` (v novém bloku).

```

1 int a = 1; // globální proměnná
2 void function(void)
3 {
4     // zde a ještě reprezentuje globální proměnnou
5     int a = 10; // lokální proměnná, zastiňuje globální a
6     if (a == 10) {
7         int a = 1; // nová lokální proměnná a; přístup
8         // na původní lokální a je zastíněn
9         int b = 20; // lokální proměnná s platností pouze
10        // uvnitř bloku
11        a += b + 10; // proměnná a má hodnotu 31
12    } // konec bloku
13    // zde má a hodnotu 10, je to lokální proměnná z řádku 5
14    b = 10; // b není platnou proměnnou
15 }

```

http://www.tutorialspoint.com/cprogramming/c_scope_rules.htm

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 31 / 47

Výpočetní prostředky a běh programu Rozsah platnosti proměnných Paměťové třídy

Definice vs. deklarace proměnné – extern

- Definice proměnné je přidělení paměťového místa proměnné (dle typu). *Může být pouze jedna!*
- Deklarace "oznamuje", že je proměnná někde definována.

```

1 // extern int global_variable = 10; /* extern
2   variable with initialization is a
3   definition */
4 int global_variable = 10;
5 void function(int p);      lec05/extern_var.h
6
7 #include <stdio.h>
8 #include "extern_var.h"
9 static int module_variable;
10 void function(int p)
11 {
12     fprintf(stdout, "function: p %d global
13     variable %d\n", p, global_variable);
14 }
15                                     lec05/extern_var.c

```

```

1 #include <stdio.h>
2 #include "extern_var.h"
3 int main(int argc, char *argv[])
4 {
5     global_variable ++ 1;
6     function(1);
7     global_variable ++ 1;
8     function(1);
9     global_variable ++ 1;
10    function(1);
11    return 0;
12 }
13                                     lec05/extern-main.c

```

Vícenásobná definice končí chybou.

```

clang extern_var.c extern-main.c
/tmp/extern-main-619051.o:(.data+0x0): multiple
definition of 'global_variable'
/tmp/extern_var-24a8d1.o:(.data+0x0): first
defined here
clang: error: linker command failed with exit
code 1 (use -v to see invocation)

```

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 32 / 47

Výpočetní prostředky a běh programu Rozsah platnosti proměnných Paměťové třídy

Přidělování paměti proměnným

- Přidělením paměti proměnné rozumíme určení paměťového místa pro uložení hodnoty proměnné (příslušného typu) v paměti počítače.
- Lokálním proměnným a parametrům funkce se paměť přiděluje při volání funkce.
 - Paměť zůstane přidělena jen do návratu z funkce.
 - Paměť se automaticky alokuje z rezervovaného místa – zásobník (stack).
 - Při návratu funkce se přidělené paměťové místo uvolní pro další použití.
 - Výjimku tvoří lokální proměnné s modifikátorem `static`.
 - Z hlediska platnosti rozsahu mají charakter lokálních proměnných.
 - Jejich hodnota je však zachována i po skončení funkce / bloku.
 - Jsou umístěny ve statické části paměti.
- Dynamické přidělování paměti
 - Alokace paměti se provádí funkcí `malloc()`.
 - Nebo její alternativou podle použité knihovny pro správu paměti (např. s `garbage collectorem` – `boehm-gc`).
 - Paměť se alokuje z rezervovaného místa – `haldy` (heap).

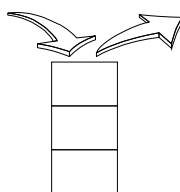
Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 33 / 47

Výpočetní prostředky a běh programu Rozsah platnosti proměnných Paměťové třídy

Zásobník

- Úseky paměti přidělované lokálním proměnným a parametrům funkce tvoří tzv. **zásobník** (stack).
- Úseky se přidávají a odebírají.
 - Vždy se odebere naposledy přidávaný úsek.
- Na zásobník se ukládá „volání funkce“.
 - Argumenty (parametry) jsou de facto lokální proměnné.
- Ze zásobníku se alokují proměnné parametrů funkce.

Opakovaným rekurzivním voláním funkce můžeme zaplnit velikost přiděleného zásobníku a program skončí chybou.



LIFO – last in, first out.

Na zásobník se také ukládá návratová hodnota funkce a také hodnota „program counter“ původně prováděné instrukce, před voláním funkce.

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 34 / 47

Výpočetní prostředky a běh programu Rozsah platnosti proměnných Paměťové třídy

Příklad rekurzivního volání funkce

- Vyzkoušejte si program pro omezenou velikost zásobníku.

```

1 #include <stdio.h>
2 void printValue(int v)
3 {
4     printf("value: %i\n", v);
5     printValue(v + 1);
6 }
7 int main(void)
8 {
9     printValue(1);
10 }
11                                     lec05/demo-stack_overflow.c

```

```

clang demo-stack_overflow.c
ulimit -s 10000; ./a.out | tail -n 3
value: 319816
value: 319817
Segmentation fault
ulimit -s 1000; ./a.out | tail -n 3
value: 31730
value: 31731
Segmentation fault

```

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 35 / 47

Výpočetní prostředky a běh programu Rozsah platnosti proměnných Paměťové třídy

Návratová hodnota funkce a kódovací styl return 1/2

- Předání hodnoty volání funkce je předepsáno voláním `return`.
 - Jak často umisťovat volání `return` ve funkci?


```

int doSomething() {
    if (cond1) {
        return 0;
    }
    if (!cond2) {
        return 0;
    }
    if (!cond3) {
        return 0;
    }
    ... some long code ...
    return 0;
}

```

<http://llvm.org/docs/CodingStandards.html>

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 36 / 47

Výpočetní prostředky a běh programu Rozsah platnosti proměnných Paměťové třídy

Návratová hodnota funkce a kódovací styl return 2/2

- Volání `return` na začátku funkce může být přehlednější.
 - Kódovací konvence může také předepisovat použití nejvýše jednoho volání `return`.
 - Má výhodu v jednoznačné identifikaci místa volání, můžeme pak například jednoduše přidat další zpracování výstupní hodnoty funkce.
- Dále není doporučováno bezprostředně používat `else` za voláním `return` (nebo jiným přerušením toku programu), např.


```

case 10:
    if (...) {
        ...
        return 1;
    } else {
        if (cond) {
            ...
            return -1;
        } else {
            break;
        }
    }

```

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 37 / 47

Výpočetní prostředky a běh programu Rozsah platnosti proměnných Paměťové třídy

Proměnné

- Proměnné představují vymezenou oblast paměti a v C je můžeme rozdělit podle způsobu alokace.
 - Statická alokace – provede se při definici `statické` nebo globální proměnné; paměťový prostor je alokovan při startu programu a nikdy není uvolněn.
 - Automatická alokace – probíhá automaticky v případě lokálních proměnných (nebo argumentů funkce); paměťový prostor je alokovan na **zásobníku** a paměť proměnné je automaticky uvolněna s koncem platnosti proměnné.
 - Např. po ukončení bloku funkce.
 - Dynamická alokace – není podporována přímo jazykem C, ale je přístupná knihovnými funkcemi.
 - Např. `malloc()` a `free()` z knihovny `<stdlib.h>` nebo `<malloc.h>`
 - http://gribblelab.org/CBootcamp/7_Memory_Stack_vs_Heap.html

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 39 / 47

Výpočetní prostředky a běh programu Rozsah platnosti proměnných Paměťové třídy

Proměnné – paměťová třída

- Specifikátory paměťové třídy (Storage Class Specifiers – SCS).
 - `auto` (lokální) – Definuje proměnnou jako dočasnou (automatickou). Lze použít pro lokální proměnné definované uvnitř funkce. Jedná se o implicitní nastavení, platnost proměnné je omezena na blok. Proměnná je v **zásobníku**.
 - `register` – Doporučuje překladači umístit proměnnou do registru procesoru (rychlost přístupu). Překladač může, ale nemusí vyhovět. Jinak stejně jako `auto`.
 - Zpravidla řešíme překladem s optimalizacemi.
 - `static`
 - Uvnitř bloku `{...}` – definujeme proměnnou jako statickou, která si **ponechává hodnotu i při opuštění bloku**. Existuje po celou dobu chodu programu. Je uložena v **datové oblasti**.
 - Vně bloku – kde je implicitně proměnná uložena v **datové oblasti** (statická) omezuje její viditelnost na modul.
 - `extern` – rozšiřuje viditelnost statických proměnných z modulu na celý program. Globální proměnné s `extern` jsou definované v **datové oblasti**.

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 40 / 47

Výpočetní prostředky a běh programu Rozsah platnosti proměnných Paměťové třídy

Příklad definice proměnných

- Hlavičkový soubor `vardec.h`

```

1 extern int global_variable;
2                                     lec05/vardec.h

```
- Zdrojový soubor `vardec.c`

```

1 #include <stdio.h>
2 #include "vardec.h"
3 static int module_variable;
4 int global_variable;
5 void function(int p);
6 int main(void)
7 {
8     int local;
9     function(1);
10    function(1);
11    function(1);
12    return 0;
13 }

```
- Výstup


```

1 func: p 1, lv 1, slv 1
2 func: p 1, lv 1, slv 2
3 func: p 1, lv 1, slv 3

```

Uvedený příklad demonstruje různé definice proměnných. V případě proměnné `global_variable` je její definice v modulu s funkcí `main()` diskutabilní. Modul `vardec.c` nebudeme linkovat s jiným program s vlastní (jinou) funkcí `main()`.

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 41 / 47

Výpočetní prostředky a běh programu Rozsah platnosti proměnných Paměťové třídy

Definice proměnných a operátor přiřazení

- Proměnné definujeme uvedením typu a jména proměnné.
 - Jména proměnných volíme malá písmena.
 - Vícelslovná jména zapisujeme s podtržítkem `_` nebo volíme tzv. *camelCase*.
<https://en.wikipedia.org/wiki/CamelCase>
- Proměnné definujeme na samostatném řádku.


```
1 int n;
2 int number_of_items;
```
- Příkaz přiřazení se skládá z operátoru přiřazení `=` a;
 - Levá strana přiřazení musí být **l-value – location-value, left-value** – musí reprezentovat paměťové místo pro uložení výsledku.
 - Přiřazení je výraz a můžeme jej tak použít všude, kde je dovolen výraz příslušného typu.

```
1 /* int c, i, j; */
2 i = j = 10;
3 if ((c = 5) == 5) {
4     fprintf(stdout, "c is 5 \n");
5 } else {
6     fprintf(stdout, "c is not 5\n");
7 }
```

lec05/assign.c

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 42 / 47

Diskutovaná témata

Shrnutí přednášky

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 45 / 47

hexdump Ukazatelová aritmetika NATO Abeceda NATO Abeceda („jinak“) Rotace textového řetězce

Kódovací příklad – hexdump – 1/4

- Implementujeme program, který vytiskne vstup načtený ze `stdin` na `stdout` v `hexa` formátu.
- Program vypíše na `stdout` nejvýše **16 hodnot** na řádek, oddělených čárkou.
- Na `stderr` vypíše na začátku "HEXDUMP:\n" a na konci "HEXDUMP END\n".

```
$ cat hw01-2.out
$ clang hexdump.c -o hexdump && ./hexdump < hw01-2.out
HEXDUMP:
44, 65, 73, 69, 74, 6b, 6f, 76, 61, 20, 73, 6f, 75, 73, 74, 61
76, 61, 3a, 20, 33, 37, 35, 39, 20, 2d, 31, 30, 30, 30, 0a
53, 65, 73, 74, 6e, 61, 63, 74, 6b, 6f, 76, 61, 20, 73, 6f, 75
73, 74, 61, 76, 61, 3a, 20, 65, 61, 66, 20, 66, 66, 66, 66, 64
38, 66, 30, 0a, 53, 6f, 75, 63, 65, 74, 3a, 20, 33, 37, 35, 39
20, 2b, 20, 2d, 31, 30, 30, 30, 30, 20, 3d, 20, 2d, 36, 32, 3d
31, 0a, 52, 6f, 7a, 64, 69, 6c, 3a, 20, 33, 37, 35, 39, 20, 2d
20, 2d, 31, 30, 30, 30, 20, 3d, 20, 31, 33, 37, 35, 39, 0a
53, 6f, 75, 63, 69, 6e, 3a, 20, 33, 37, 35, 39, 20, 2a, 20, 2d
31, 30, 30, 30, 20, 3d, 20, 2d, 33, 37, 35, 39, 30, 30, 30
30, 0a, 50, 6f, 64, 69, 6c, 3a, 20, 33, 37, 35, 39, 20, 2f, 20
2d, 31, 30, 30, 30, 20, 3d, 20, 3d, 20, 30, 0a, 50, 72, 75, 64, 65
72, 3a, 20, 2d, 33, 31, 32, 30, 2e, 35, 0a
HEXDUMP END
$
```

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 49 / 47

Část III

Část 3 – Zadání 4. domácího úkolu (HW4)

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 43 / 47

Diskutovaná témata

Diskutovaná témata

- Ukazatele a modifikátor `const`
- Dynamická alokace paměti
- Ukazatel na funkci
- Paměťové třídy
- Volání funkcí

■ Přístě: Struktury a union, přesnost výpočtu a vnitřní reprezentace číselných typů.

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 46 / 47

hexdump Ukazatelová aritmetika NATO Abeceda NATO Abeceda („jinak“) Rotace textového řetězce

Kódovací příklad – hexdump – 2/4

- Program napíšeme nejdříve kreativně, ale s počtem hodnot na řádek `MAX_WIDTH`.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #ifndef MAX_WIDTH
4 #define MAX_WIDTH 16
5 #endif
6 #endif
7
8 int main(void)
9 {
10     int ret = EXIT_SUCCESS;
11     int n = 0; // initialize
12     int c; // we do not need to init.
13     fprintf(stderr, "HEXDUMP:\n");
14
15     while ((c = getchar()) != EOF) {
16         if (n >= MAX_WIDTH) {
17             n = 0;
18             putchar('\n');
19         }
20         if (n > 0) {
21             printf(", ");
22         }
23         printf("%02x", c);
24         n += 1;
25     } //end while loop
26     if (n > 0) {
27         putchar('\n');
28     }
29     fprintf(stderr, "HEXDUMP END\n");
30     return ret;
31 }
```

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 50 / 47

Zadání 4. domácího úkolu HW4

Téma: **Caesarova šifra** Povinné zadání: **3b**; Volitelné zadání: **není**; Bonusové zadání: **5b**

- Motivace:** Získat zkušenosti s dynamickou alokací paměti. Implementovat výpočetní úlohu optimalizačního typu.
- Cíl:** Osvojit si práci s dynamickou alokací paměti.
- Zadání:** <https://cw.fel.cvut.cz/wiki/courses/bab36prga/hw/hw4>
 - Načtení dvou vstupních textů a tisk dekodované zprávy na výstup.
 - Zakódovaný text i (špatně) odposlechnutý text mají stejné délky.
 - Nalezení největší shody dekodovaného a odposlechnutého textu na základě hodnoty posunu v Caesarově šifře.
 - Optimalizace hodnoty Hammingovy vzdálenosti.
https://en.wikipedia.org/wiki/Hamming_distance
- Volitelné zadání** rozšiřuje úlohu o uvažování chybějících znaků v odposlechnutém textu, což vede na využití Levenštejnovy vzdálenosti.
https://en.wikipedia.org/wiki/Levenshtein_distance
- Termín odevzdání:** **06.04.2024, 23:59:59 PDT.**
- Bonusová úloha:** **24.05.2024, 23:59:59 CEST.**

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 44 / 47

hexdump Ukazatelová aritmetika NATO Abeceda NATO Abeceda („jinak“) Rotace textového řetězce

Část V

Appendix

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 47 / 47

hexdump Ukazatelová aritmetika NATO Abeceda NATO Abeceda („jinak“) Rotace textového řetězce

Kódovací příklad – hexdump – 3/4

- Program upravíme redukcí počtu zanoření vyjmutím (dekompozice).

```
8 void print_hex(int c, int *n);
9
10 int main(void)
11 {
12     int n = 0;
13     int c; // we do not need to init
14     fprintf(stderr, "HEXDUMP:\n");
15     while ((c = getchar()) != EOF) {
16         print_hex(c, &n);
17     } //end while loop
18     if (n > 0) {
19         putchar('\n'); // final EOL
20     }
21     fprintf(stderr, "HEXDUMP END\n");
22     return EXIT_SUCCESS; // always ok
23 }
24
15 void print_hex(int c, int *n)
16 {
17     if (*n >= MAX_WIDTH) {
18         *n = 0;
19         putchar('\n');
20     }
21     if (*n > 0) {
22         printf(", ");
23     }
24     printf("%02x", c);
25     *n += 1;
26 }
```

- Předávání ukazatele (adresy) proměnné `n` je nutné.
- Vyzkoušejte chování programu s předáváním hodnoty `n`.
`void print_hex(int c, int n);`

Jan Faigl, 2024 BAB36PRGA – Přednáška 05: Paměťové třídy 51 / 47

Kódovací příklad – hexdump – 4/4

- Program spustíme s přeměrováním `stdin` ze souboru, např. `hw01-2.out` a přeměrováním výstupu `stdout` do souboru `hex.out`.
- Při kompilaci definujeme počet hodnot na řádek `MAX_WIDTH=20` a program spustíme s výstupem `stdout` do souboru `hex.out`.

```
# clang hexdump-2.c -o hexdump.kk ./hexdump.c hw01-2.out > hex.out
HEXDUMP BDD
# cat hex.out
44, 65, 73, 69, 74, 69, 6f, 76, 61, 20, 73, 6f, 75, 73, 74, 61, 61, 63, 3a, 20,
33, 37, 38, 39, 20, 2a, 31, 20, 30, 30, 0a, 83, 65, 73, 74, 6a, 61, 63, 74
80, 6f, 70, 61, 20, 73, 6f, 75, 73, 74, 61, 76, 61, 3a, 20, 65, 61, 66, 20, 66
66, 66, 64, 64, 68, 66, 30, 0a, 83, 6f, 75, 63, 65, 74, 3a, 20, 33, 37, 35, 39
20, 2b, 20, 2a, 31, 30, 30, 30, 20, 2a, 2b, 2a, 36, 32, 3a, 31, 0a, 82, 6f
7a, 64, 69, 66, 8a, 20, 33, 37, 35, 39, 20, 2a, 20, 2a, 31, 30, 30, 30, 20
5a, 20, 31, 33, 37, 35, 39, 0a, 83, 6f, 75, 63, 69, 6a, 3a, 20, 33, 37, 35, 39
20, 2a, 20, 2a, 31, 30, 30, 30, 20, 2a, 2a, 33, 37, 35, 39, 30, 30, 30, 20
30, 0a, 50, 6f, 64, 69, 6a, 3a, 20, 33, 37, 35, 39, 20, 2f, 20, 2a, 31, 30, 30
30, 30, 20, 2a, 30, 0a, 50, 72, 75, 6f, 65, 72, 3a, 20, 2a, 33, 31, 32, 30
2a, 35, 0a
# cat hex.err
HEXDUMP BDD
```

- Obsah `stderr` není přeměrován, proto se vypíše na terminál.
- Obsah `stderr` je přeměrován, proto se nevypíše na terminál.

Vykoušejte program s přeměrováním binárního souboru na `stdin` našeho `hexdump`.

Kódovací příklad – NATO Abeceda – 1/4

- Implementujeme program, který převede vstupní text (ASCII, znaky A–Z a a–z) do NATO abecedy, ve které jsou písmena hláskována prostřednictvím následujících jmen.

- Alpha, Bravo, Charlie, Delta, Echo, Foxtrot, Golf, Hotel, India, Juliett, Kilo, Lima, Mike, November, Oscar, Papa, Quebec, Romeo, Sierra, Tango, Uniform, Victor, Whiskey, X-ray, Yankee, Zulu.

- V programu definujeme pole ukazatelů na textové literály s jednotlivými slovy.

- Programové otestujeme, že slova odpovídají počátečním písmenům A–Z.

```
# cat in.txt
I like PHP and programming in C.
# clang nato-alphabet.c.kk ./a.out < in.txt 2>/dev/null
India Lima India Kilo Echo Papa Romeo Golf Alpha
November Delta Papa Romeo Oscar Golf Romeo Alpha
Mike Mike India November Golf India November
Charlie
```

- Očekávaný výstup pro vstup `in.txt`.

- Implementujeme testovací funkce.

```
# static char words[] = { "Alpha", "Bravo", "Charlie", "Delta", "Echo", "Foxtrot", "Golf", "Hotel", "India", "Juliett", "Kilo", "Lima", "Mike", "November", "Oscar", "Papa", "Quebec", "Romeo", "Sierra", "Tango", "Uniform", "Victor", "Whiskey", "X-ray", "Yankee", "Zulu", NULL };
// it is an array of pointers to text literals
int count_words_array(char words[]); // Using array
int count_words(char **words); // Pointer to pointers
bool check_alphabet_words(char *words[]);

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <stdbool.h>
static const char * const words[] = { "Alpha", ... , NULL };
int count_words_array(const char * const words[]);
bool check_alphabet_words(const char * const words[]);
int main(void)
{
// Assert macro debug and development only, see -DDEBUG
assert(count_words_array(words) == 'Z' + 'A' + 1);
assert(check_alphabet_words(words));
int c;
while ((c = getchar()) != EOF) {
c = (c >= 'a' && c <= 'z') ? c - 'a' + 'A' : c;
if (c >= 'A' && c <= 'Z') {
printf("%c\n", words[c - 'A']);
}
}
char my_toupper(char c) // or use toupper() from <ctype.h>
{
if (c >= 'a' && c <= 'z') {
c = c - 'a' + 'A';
}
return c;
}
}
// Funkci my_toupper() můžeme nahradit použitím ternárního operátoru.
```

- V rámci zphlednění můžeme překlad (řádky 15–21) dát do samostatné funkce `void translate(const char * const words[])`.

Ukazatelová (pointerová) aritmetika

- S ukazately (pointery) lze provádět aritmetické operace + a – (přičítat nebo odčítat celé číslo).

- ukazatel = ukazatel stejného typu + (nebo –) a celé číslo (int).
- Nebo lze používat zkrácený zápis např. `ukazatel += 1` a unární operátory např. `ukazatel++`.

- Aritmetické operace jsou užitečné pokud ukazatel odkazuje na více položek daného typu (souvislý paměť).

- Např. pole položek příslušného typu;
- Dynamicky alokovaný souvislý blok paměti.

- Přičtením hodnoty celého čísla k pointeru „posouváme“ hodnotu pointeru na další prvek, např.

```
int a[10];
int *p = a;
int i = *(p+2); // odkazuje na hodnotu 3. prvku pole a

Podle typu ukazatele se hodnota adresy přísluše zvyší.
(p+2) je ekvivalentní adrese p + 2*sizeof(int).
Příklad použití viz lec04/pointers_and_array.c
```

Kódovací příklad – NATO Abeceda – 2/4

```
// array is terminated by NULL used for counting
static char *words[] = { "Alpha", ... , "Zulu", NULL };
// array-like variant
int count_words_array(char *words[])
{
int n = 0;
while (words[n] != NULL) {
sprintf(stderr, "DEBUG: %s\n", words[n]);
n++;
}
return n;
}
// pure pointer variant
int count_words(char **words)
{
int n = 0;
char **cur = words;
while (*cur) {
sprintf(stderr, "DEBUG: %s\n", *cur);
cur++;
}
return n;
}
bool check_alphabet_words(char *words[])
{
bool ret = true;
char c = 'A';
while (c <= 'Z') {
char *cur = words;
while (*cur) {
if (c != *cur[0]) { // the first letter needs to match
ret = false;
break;
} else {
cur++;
}
}
return ret;
}
}
// Můžeme použít const.
```

- Ukazatel je posloupnost prvků stejného typu (pole na `char` – textový řetězec).
- Hodnota `&words[0]` je identická adresa jako hodnota `words`.

```
int fill_words(char* alphabet, char *words[])
{
int ret = EXIT_SUCCESS;
char *cur = alphabet;
for (char c = 'A'; c <= 'Z'; ++c) {
assert(words[c - 'A'] == NULL); // nemá být nastaveno
cur = strchr(cur, c); // vyhledání řetězce začínající c
assert(cur); // písmeno c musí být v abecedě
*cur = '\0';
words[c - 'A'] = *cur; // nastavení a posun kurzoru
assert(words[c - 'A']); // it should be set now
return ret; // pragmaticky vždy EXIT_SUCCESS nebo assert.
}
}
// k testování správné inicializace datových struktur.
Makro slouží pro ladění, viz man assert.
```

Kódovací příklad – NATO Abeceda („jinak“) – 1/2

- Slova abecedy uložíme jako řetězec `alphabet` všech slov spojených bez mezery, do kterého budeme odkazovat na jednotlivá slova polem ukazatelů na textové řetězce (`words`).

- První písmeno slova abecedy používáme k indexaci, např. `C` harakter je odkazované ukazatelem `words['C' - 'A']`. První znak slova tak můžeme v abecedě `alphabet` nahradit znakem `'\0'` získáme textové řetězce.

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <stdbool.h>
static const char * const words[] = { "Alpha", ... , NULL };
int count_words_array(const char * const words[]);
bool check_alphabet_words(const char * const words[]);
int main(void)
{
// Ukazatel na textový literál. Literál nemůžeme změnit!
static char alphabet = "AlphaBravoCharlie...";
static char alphabet[] =
"AlphaBravoCharlieDeltaEchoFoxtrotGolfHotelIndia"
"JuliettKiloLimaMikeNovemberOscarPapaQuebecRomeo"
"SierraTangoUniformVictorWhiskeyX-rayYankeeZulu";
// pole ukazatelů na textové řetězce
static char *words['Z' - 'A' + 1] = { 0 };
int fill_words(char* str, char *words[]);
int main(void)
{
int ret = fill_words(alphabet, words);
if (ret) {
for (char c = 'A'; c <= 'Z'; ++c) {
sprintf(stderr, "DEBUG: %02d '%c' - %s\n",
c, c, words[c - 'A']);
}
}
return ret;
}
}
// Další rozšíření programu může být zpracování jiných znaků, než znaků abecedy 'A'-'Z' a 'a'-'z'.
```

- Další rozšíření programu může být zpracování jiných znaků, než znaků abecedy 'A'-'Z' a 'a'-'z'.

Příklad ukazatelové aritmetiky

- Ukazatel je proměnná jejíž hodnota je adresa v paměti.

- Ukazatelová aritmetika zohledňuje typ proměnné, její velikost v paměti.
- Přičtením hodnoty 1 k proměnné typu ukazatel je vypočtena adresa následujícího prvku, adresa je zvětšena o hodnotu odpovídající `sizeof()` příslušného typu.

```
int getLength(char *str)
{
int ret = 0;
while (str && str[ret] != '\0') {
ret++;
}
return ret;
}
int getLengthPtr(char *str)
{
int ret = 0;
while (str && *(str++) != '\0') {
ret++;
}
return ret;
}
int getLengthPtr2(char *str)
{
int ret = 0;
while (str && *(str++)) ret++;
return ret;
}
// Textový řetězec můžeme interpretovat jako pole znaků char[] nebo ukazatel char*.
lec04/string_length.c
```

- Textový řetězec můžeme interpretovat jako pole znaků `char[]` nebo ukazatel `char*`.

Kódovací příklad – NATO Abeceda – 3/4

- Můžeme použít `const`.

```
static const char * const words[] = { "Alpha", ... , NULL };
int count_words_array(const char * const words[]);
int n = 0;
while (words[n] != NULL) {
sprintf(stderr, "DEBUG: check %c for '%c'\n", *cur, c);
if (c != *cur[0]) { // the first letter needs to match
ret = false;
break;
} else {
cur++;
}
}
return ret;
}
int count_words(const char * const * const words)
{
int n = 0;
// cur je ukazatel na data typu konstantní
// ukazatel na konstantní textový řetězec
// (na konstantní ukazatel na konstantní hodnoty char).
const char * const * cur = words; // cur cheme měnit
while (*cur) {
cur++; // cur není konstantní ukazatel
}
return n;
}
#include <stdio.h>
#include <stdbool.h>
static const char * const words[] = { "Alpha", ... , NULL };
int count_words_array(const char * const words[]);
bool check_alphabet_words(const char * const words[]);
int main(void)
{
int ret = EXIT_SUCCESS;
sprintf(stderr, "DEBUG: size %i\n", sizeof(words));
int n = count_words_array(words);
printf(stderr, "DEBUG: no. of words: %i\n", n);
n = count_words(words[0]);
printf(stderr, "DEBUG: no. of words: %i\n", n);
bool checked = check_alphabet_words(words);
printf(stderr, "DEBUG: check_alphabet_words passed (%s)\n",
checked ? "OK" : "FAIL");
return ret;
}
// Další rozšíření programu může být zpracování jiných znaků, než znaků abecedy 'A'-'Z' a 'a'-'z'.
```

- Další rozšíření programu může být zpracování jiných znaků, než znaků abecedy 'A'-'Z' a 'a'-'z'.

Kódovací příklad – NATO Abeceda („jinak“) – 2/2

- Přidáme příklad znaků načítaných ze `stdin` a implementaci zphlednění.

```
static char alphabet[] =
"AlphaBravoCharlieDeltaEchoFoxtrotGolfHotelIndia"
"JuliettKiloLimaMikeNovemberOscarPapaQuebecRomeo"
"SierraTangoUniformVictorWhiskeyX-rayYankeeZulu";
static char *words['Z' - 'A' + 1] = { 0 };
int fill_words(char* str, char *words[]);
int main(void)
{
int ret = fill_words(alphabet, words);
if (ret) {
for (char c = 'A'; c <= 'Z'; ++c) {
printf(stderr, "DEBUG: %02d '%c' - %s\n",
c, c, words[c - 'A']);
}
}
// První písmeno slova abecedy.
return ret;
}
int c;
while ((c = getchar()) != EOF) {
c = (c >= 'a' && c <= 'z') ? c - 'a' + 'A' : c;
// volání funkce toupper() bude přehlednější!
if (c >= 'A' && c <= 'Z') {
printf("%c\n", c, words[c - 'A']);
}
}
return ret;
}
// Další rozšíření programu může být zpracování jiných znaků, než znaků abecedy 'A'-'Z' a 'a'-'z'.
```

- Další rozšíření programu může být zpracování jiných znaků, než znaků abecedy 'A'-'Z' a 'a'-'z'.

Kódovací příklad – Rotace textového řetězce – 1/4

- Implementujeme program, který načte ze `stdin` dva textové řetězce (dva řádky zakončené `'\n'`) a pokusí se najít rotaci (posunutí – `offset`) druhého řádku tak, aby odpovídal prvnímu řádku.
- Oba řádky (řetězce) předpokládáme, že jsou stejně dlouhé.
- Chybu dynamické alokace program indikuje návratovou hodnotou 129, chybu vstupu hodnotou 100, jinak vrací `EXIT_SUCCESS`.
- Délka řetězců je až do maximálního hodnoty `size_t`, posunutí pouze do `INT_MAX`.
- V případě neúspěšné dynamické alokace program ukončujeme voláním `exit(129)`;

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <limits.h> // for INT_MAX
5 #ifndef INIT_LEN
6 #define INIT_LEN 8
7 #endif
8 enum { ERROR_OK = EXIT_SUCCESS, ERROR_IN = 100, ERROR_MEM = 129 };
9
10 void* my_realloc(void* ptr, size_t size,
11               const char *file, const int line);
12
13
14
15
16
17
18
19
20
21
22 int main(void)
23 {
24     int ret = ERROR_OK;
25     char *l1 = read_line();
26     char *l2 = read_line();
27     size_t n1, n2;
28
29     if (l1 && l2 && (n1 = strlen(l1)) == (n2 = strlen(l2))) {
30         fprintf(stderr, "DEBUG: l1[%lu]: \"%s\"\n", n1, l1);
31         fprintf(stderr, "DEBUG: l2[%lu]: \"%s\"\n", n2, l2);
32         int offset = get_offset(l1, n1, l2, n2);
33         fprintf(stdout, "Matching offset %d\n", offset);
34         offset >= 0 && print_offset(l2, n2, offset); // call print_offset only if offset >= 0
35     } else {
36         fprintf(stderr, "ERROR: Wrong input!\n");
37         ret = ERROR_IN;
38     }
39
40     free(l1); // free(ptr) - If ptr is NULL no action occurs.
41     free(l2); // See man free.
42     return ret;
43 }

```

```

44 void* my_realloc(void* ptr, size_t size,
45               const char *file, const int line)
46 {
47     void* ret = realloc(ptr, size);
48     if (ret) {
49         fprintf(stderr, "ERROR: Cannot realloc %lu bytes -- called
50 at %s:%d\n", size, file, line);
51         free(ptr);
52         exit(ERROR_MEM);
53     }
54     return ret;
55 }

```

- Volání `realloc()` alokuje nebo přealokuje paměť.
- Funkci předáváme soubor a číslo řádku, kde funkci `my_realloc()` voláme, pro indikaci, kde došlo k chybě.

Kódovací příklad – Rotace textového řetězce – 2/4

```

14 char* read_line(void) // read a line from stdin, terminated by '\n' return as null-terminated string
15 char* shift(int offset, const char* src, size_t n, char *dst); // src and dst are strings at least n long (+1 for '\0')
16
17 int get_offset(const char *s1, size_t n1, const char *s2, size_t n2); // offset - max INT_MAX; strings - up to can size_t
18
19 int print_offset(const char *s, size_t n, int offset);
20
21 int main(void)
22 {
23     int ret = ERROR_OK;
24     char *l1 = read_line();
25     char *l2 = read_line();
26     size_t n1, n2;
27
28     if (l1 && l2 && (n1 = strlen(l1)) == (n2 = strlen(l2))) {
29         fprintf(stderr, "DEBUG: l1[%lu]: \"%s\"\n", n1, l1);
30         fprintf(stderr, "DEBUG: l2[%lu]: \"%s\"\n", n2, l2);
31         int offset = get_offset(l1, n1, l2, n2);
32         fprintf(stdout, "Matching offset %d\n", offset);
33         offset >= 0 && print_offset(l2, n2, offset); // call print_offset only if offset >= 0
34     } else {
35         fprintf(stderr, "ERROR: Wrong input!\n");
36         ret = ERROR_IN;
37     }
38
39     free(l1); // free(ptr) - If ptr is NULL no action occurs.
40     free(l2); // See man free.
41     return ret;
42 }

```

Kódovací příklad – Rotace textového řetězce – 3/4

```

14 char* read_line(void)
15 {
16     size_t capacity = INIT_LEN;
17     char *str = my_realloc(NULL, sizeof(char) * (INIT_LEN + 1)
18     , __FILE__, __LINE__); //+1 for '\0'
19     size_t len = 0;
20     int c;
21     while ((c = getchar()) != EOF && c != '\n') {
22         if (len == capacity) {
23             capacity += 2;
24             str = my_realloc(str, sizeof(char) * (capacity + 1),
25             , __FILE__, __LINE__); //+1 for '\0'
26         }
27         str[len++] = c;
28     }
29     if (len > 0) {
30         str[len] = '\0';
31     } else {
32         free(str);
33         str = NULL;
34     }
35     return str;
36 }

```

```

81 char* shift(int offset, const char* src, size_t n, char *dst)
82 {
83     for (size_t i = 0; i < n; ++i) { // n type is size_t !!!
84         dst[i] = src[(offset + i) % n];
85     }
86     return dst;
87 }
88
89 int get_offset(const char *s1, size_t n1, const char *s2, size_t n2)
90 { // we already checked that s1 && s2 && n1 == n2
91     int ret = -1;
92     int max_shift = INT_MAX < n2 ? INT_MAX : n2; // limits.h
93     char *s = my_realloc(NULL, sizeof(char) * (n2 + 1), __FILE__,
94     , __LINE__); //+1 for '\0'
95     for (int i = 0; i < max_shift; ++i) {
96         s = shift(i, s2, n2, s); // shift s2 to s and return s
97         if (strcmp(s1, s) == 0) { //strings matched
98             ret = i; // perfect match, exit the loop
99             break;
100         }
101     }
102     free(s); // s is dynamically allocated, release the memory
103     return ret;
104 }

```

- `read_line()` vrací `NULL` pouze pokud je načten prázdný řádek.
- Chyba alokace dynamické paměti ukončí program voláním `exit()` v naší funkci `my_realloc()`.
- Posuneme 2. řádek (`s`) a testujeme jestli je identický s 1. řádkem.
- Funkce `strcmp()` porovnává řetězce lexikograficky, proto vrací `int`.

Kódovací příklad – Rotace textového řetězce – 4/4

- K vytištění posunutého řetězce v samostatné funkci `print_offset()` alokujeme dynamickou paměť, kterou před ukončení funkce opět uvolníme.
- Program otestujeme pro ukázkový vstup.
- Vyzkoušejte si chování programu v kombinaci s `valgrind` pro detekci chybného přístupu k paměti, např. chybná alokace paměti pro posunutý řetězec.

```

100 int print_offset(const char *s, size_t n, int offset)
101 {
102     int ret = 1;
103     char *str = my_realloc(NULL, sizeof(char) * (n + 1),
104     , __FILE__, __LINE__); //+1 for '\0'
105     shift(offset, s, n, str);
106     fprintf(stderr, "DEBUG: shift: \"%s\"\n", str);
107     free(str);
108     return ret;
109 }

```

```

110 char *l1 = read_line();
111 char *l2 = read_line();
112 size_t n1, n2;
113 if (l1 && l2 && (n1 = strlen(l1)) == (n2 = strlen(l2))) {
114     fprintf(stderr, "DEBUG: l1[%lu]: \"%s\"\n", n1, l1);
115     fprintf(stderr, "DEBUG: l2[%lu]: \"%s\"\n", n2, l2);
116     int offset = get_offset(l1, n1, l2, n2);
117     fprintf(stdout, "Matching offset %d\n", offset);
118     offset >= 0 && print_offset(l2, n2, offset);
119 } else {
120     fprintf(stderr, "ERROR: Wrong input!\n");
121     ret = ERROR_IN;
122 }

```

```

123 for (size_t i = 0; i < n; ++i) {
124     dst[i] = src[(offset + i) % n];
125 }

```

```

$ valgrind ./shift < input.txt
...
==80708== Invalid write of size 1
==80708== at 0x202240: shift (shift.c:84)
==80708== by 0x202092: get_offset (shift.c:95)
==80708== by 0x201DF2: main (shift.c:36)

```