

CH-230-A

Programming in C and C++

C/C++


Tutorial 7

Dr. Kinga Lipskoch

Fall 2020

Another Function Pointer Example

```
1 #include <stdio.h>
2 void output(void) {
3     printf("%s\n", "Please enter a number:");
4 }
5 int sum(int a, int b) {
6     return (a + b);
7 }
8 int main() {
9     int x, y;
10    void (*fptr1)(void);
11    int (*fptr2)(int, int);
12    fptr1 = output;
13    fptr2 = sum;
14    fptr1(); // cannot see whether function or pointer
15    scanf("%d", &x);
16    (fptr1)(); // some prefer this to show it is pointer
17    (*fptr1)(); // complete syntax, same as above
18    scanf("%d", &y);
19    printf("The sum is %d.\n", fptr2(x, y));
20 }
```



Alternatives for Usage

```
1 int (*fct) (int, int);
2 /* define pointer to a fct */
3 int plus(int a, int b) {return a+b;}
4 int minus(int a, int b) {return a-b;}
5 int a=3; int b=4;
6 fct = &plus;
7 /* calling fct() same as calling plus() */
8 printf("fct(a,b):%d\n", fct(a,b)); /* 7 */
```



or

```
1 printf("fct(a,b):%d\n", (*fct)(a,b)); /* 7 */
2 fct = &minus;
3 /* calling fct() same as calling minus() */
4 printf("fct(a,b):%d\n", fct(a,b)); /* -1 */
```

Printing a List with Function Pointers

```
1 void foreach_list_simple(struct list *my_list,
2     void (*func)(int num)) {
3     struct list *p;
4     for (p = my_list; p != NULL; p = p->next) {
5         func(p->info);
6     }
7 }
8 void printnum(int num) {
9     printf("%d ", num);
10 }
11 int main() {
12     ...
13     foreach_list_simple(my_list, printnum);
14     return 0;
15 }
```

Summing Up a List with Function Pointers

```
1 void foreach_list(struct list *my_list,
2     void (*func)(int num, void *state),
3     void *state) 
4     struct list *p;
5     for (p = my_list; p != NULL; p = p->next) {
6         func(p->info, state);
7     }
8 }
9 void sumup(int num, void *state) {
10     int *p = (int *) state;
11     *p += num;
12 }
13 int main() {
14     ...
15     int sum = 0;
16     foreach_list(my_list, sumup, &sum); 
17     printf("sum=%d\n", sum); return 0;
18 }
```

Sorting with Function Pointers

- ▶ An array of lines (strings) can be sorted according to multiple criteria:
 - ▶ Lexicographic comparison of two lines (strings) is done by `strcmp()`
 - ▶ Function `numcmp()` compares two lines on the basis of numeric value and returns the same kind of condition indication as `strcmp` does
- ▶ These functions are declared ahead of the main and a pointer to the appropriate one is passed to the function `qsort` (implementing quick sort)

Function strcmp()

- ▶ strcmp() compares the two strings s1 and s2
- ▶ It returns an integer less than, equal to, or greater than zero if s1 is found, respectively, to be less than, to match, or be greater than s2

```
1 #include <stdio.h>
2 #include <string.h>
3 int main() {
4     char s1[30], s2[30];
5     scanf("%29s", s1);
6     scanf("%29s", s2);
7     // avoid buffer overflow on the strings
8     if (!strcmp(s1, s2)) {
9         printf("Both strings are equal!\n");
10    }
11    return 0;
12 }
```

Function numcmp()

- ▶ Function `strcmp()` compares two strings and returns `<0`, `0`, `>0`

- ▶ Here you see function `numcmp()`, which compares two strings on a leading numeric value, computed by calling `atof`

```
1 #include <stdlib.h>
2 /* numcmp: compare s1 and s2
   numerically */
3 int numcmp(char *s1, char *s2 ){
4     double v1, v2;
5     v1 = atof(s1);
6     v2 = atof(s2);
7     if (v1 < v2)
8         return -1;
9     else if (v1 > v2)
10        return 1;
11    else
12        return 0;
13 }
```



Further Refinement of the Sorting Problem

- ▶ You want to write a sorting function
- ▶ The sorting algorithm is the same, but the comparison function may be different (i.e., you want ordering by different keys, different data types, increasing/decreasing sequence)
- ▶ Can we have a pointer to a comparison function as parameter for the sort function and write the sort function only once, always calling it with different comparison functions?

Function Pointer as Function Argument

```
1 int my_sort(int *array, int n,  
2           int (*my_cmp) (int ,int)) {  
3     ...  
4     if ( my_cmp(array[i],array[i+1]) == 1)  {  
5         ...  
6     }  
7     ...  
8 }
```

Usage of Function Pointers as Function Arguments

```
1 int fct1(int a, int b) {  
2     ...  
3 }  
4 int *array, n;  
5 /* pass your function as argument */  
6 my_sort(array, n, &fct1);
```

Using the qsort() from stdlib.h

This version of the qsort is declared in stdlib.h:

```
1 void qsort(void *base ,  
2           size_t nmemb ,  
3           size_t size ,  
4           int (*compare)(const void *,  
5                           const void *));
```



User Supplied Comparison Function

```
1 int my_compare(const void *va, const void *vb) {  
2     const int* a = (const int*) va;  
3     const int* b = (const int*) vb;  
4     if (*a < *b) return -1;  
5     else if (*a > *b) return 1;  
6     else return 0;  
7 }
```



Calling qsort()

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4 #define NUM_ELEMENTS 50
5 int my_compare(const void *va, const void *vb) {
6     const int* a = (const int*) va;
7     const int* b = (const int*) vb;
8     if (*a < *b) return -1;
9     else if (*a > *b) return 1;
10    else return 0;
11 }
12 int main() {
13     srand(time(NULL)); // initialize random number generator
14     int arr[NUM_ELEMENTS];
15     int i;
16     /* fill array with random numbers */
17     for (i = 0; i < NUM_ELEMENTS; i++)
18         arr[i] = rand() % 1000;
19     qsort(arr, NUM_ELEMENTS, sizeof(arr[0]), my_compare);
20     for (i = 0; i < NUM_ELEMENTS; i++)
21         printf("%d\n", arr[i]);
22     return 0;
23 }
```

Why useful?

- ▶ Can use `qsort()` or other functions with your own data types (`struct`), just need to write the comparison function, but no need to duplicate the sorting function itself
- ▶ Change comparison function to reverse the order
- ▶ Change comparison function to sort by different key (member of your `struct`), e.g., sort by first name, last name, age, ...

Stacks (1)

- ▶ A **stack** is a container where items are retrieved according to the order of insertion
- ▶ For a stack, the element deleted from the set is the one most recently inserted
- ▶ It is called **Last-In First-Out policy**: **LIFO**


Stacks (2)

Abstract operations on a stack:

- ▶ `push(x, s)` insert item x at top of stack s
- ▶ `pop(s)` remove (and return) the top item of stack s
- ▶ `init(s)` create an empty stack
- ▶ `isFull(s)` determine whether stack is full
- ▶ `isEmpty(s)` determine whether stack is empty

Stacks (3)

- ▶ Easiest implementation uses an array with an index variable that represents top of stack

```
1 struct stack {  
2     unsigned int count;   
3     int array[10];      // Container  
4 };
```

- ▶ Linked list implementation is also possible
 - ▶ Advantage: no overflow