ECE 264 Advanced C Programming

Contents

1 Pointer, Address, and Value

Two essential components in a computer are the *processor* and the *memory*. The memory is organized into a list. Each element in the list has an *address* and a *value*. We, as programmers, can assign the values but we have no control of the addresses. The addresses are assigned by the operating systems (such as Windows or Linux). The following is one example of

```
int a = 5;
char b = 'e';
```

address	value	comment
0XBF800D40	5	/* a */
0XBF800D44	e	/* b, ASCII 101 */

here 0X means hexadecimal. When we use `a`, the compiler knows that we are using the *address* of 0XBF800D40. When we write

```
a = 7;
```

the compiler modifies the *value* at the *address* of 0XBF800D40. We can obtain the address of `a` by adding an ampersand `&` in front of `a`:

```
& a
```

returns 0XBF800D40. C uses *pass by value* when calling a function.

```
int add(int a, int b)
{
   return (a + b);
}
```

The arguments `a` and `b` take the values. When calling the function

```
int x = add(7, 9);
```

`a` has value 7 and `b` has value 9. When calling the function using variables

```
int s = 11;
int t = 32;
int x = add(s, t);
```

`a` has value 11 and `b` has value 32. The function `add` has **no** information about the *address* of `s` or `t`. Hence, it is **not** possible to change `s` or `t` inside the function and make the change visible to the caller. If we want to make the change visible to the caller, we have to use **pointers**.

```
void swap(int * p, int * q)
{
    ...
}
int x = 5;
int y = 11;
swap(& x, & y);
/* x = 11 and y = 5 now */
```

Calling `swap` using the *addresses* of `x` and `y` allows `swap` to directly modify the values stored at the addresses. Therefore, the change is visible after the function returns.

2 Selection Sort

With the swap function and the addresses of array elements, we can *sort* the elements of an array. Sorting is one of the most important steps in many programs. Sorting means *ordering* data based on a particular part (also called *key*) in each item. For example, we may sort the list of students based on their last names. Another example is sorting airplane tickets by prices or arrival time. You can sort emails by the arrival dates. *Selection sort* is a simple way to sort elements.

```
#include <stdio.h>
void swap(int * a, int * b)
{
  int temp = *a;
  (*a) = (*b);
  (*b) = temp;
}
void printArray(int * x, int n)
{
  int i;
  for (i = 0; i < n; i ++)
    { printf("%8d", x[i]);  }
  printf("\n");
}
int main(int argc, char * argv[])
{
  int x[] = {6, 7, 3, 2, 0, 9, -4, 1};
  int n = sizeof(x) / sizeof(int);
  printArray(x, n);
  int i1, i2, mInd;
  for (i1 = 0; i1 < n - 1; i1 ++)
    {
      mInd = i1;
      for (i2 = i1 + 1; i2 < n; i2 ++)
        {
          if (x[mInd] > x[i2])
            { mInd = i2; }
        }
      if (mInd != i1)
        {
          printf("\ni1 = %d, mInd = %d, x[i1] = %d, x[mInd] = %d\n",
                 i1, mInd, x[i1], x[mInd]);
          swap(&x[i1], &x[mInd]);
          printArray(x, n);
```

```
        }
    }
  printArray(x, n);
  return 0;
}
```

If you look carefully, in each iteration of the outer loop (`i1`), the i^{th} smallest value if moved to the i^{th} element. After the first (`i1` = 0) iteration, the smallest value (-4) is the first element. After the second (`i1` = 1) iteration, the second smallest value (0) is the second element. It is called "selection" sort because the smallest value is selected moving to the correction location. It contains two levels of iterations. The first goes from zero to the number of elements -1. The second goes from `i1 + 1` to the last element.

How should we change the program if we want the result in the descending order, instead of the ascending order? It is simple. We just change this line

```
    if (a[i] > a[j])
```

to

```
    if (a[i] < a[j])
```

```
       6       7       3       2       0       9      -4       1

i1 = 0, mInd = 6, x[i1] = 6, x[mInd] = -4
      -4       7       3       2       0       9       6       1

i1 = 1, mInd = 4, x[i1] = 7, x[mInd] = 0
      -4       0       3       2       7       9       6       1

i1 = 2, mInd = 7, x[i1] = 3, x[mInd] = 1
      -4       0       1       2       7       9       6       3

i1 = 4, mInd = 7, x[i1] = 7, x[mInd] = 3
      -4       0       1       2       3       9       6       7

i1 = 5, mInd = 6, x[i1] = 9, x[mInd] = 6
      -4       0       1       2       3       6       9       7

i1 = 6, mInd = 7, x[i1] = 9, x[mInd] = 7
      -4       0       1       2       3       6       7       9
      -4       0       1       2       3       6       7       9
```

3 String

What is a string? You can think of a string as a word or a sentence enclosed by double quotations. "Hello" is a string. "Good Morning" is another string. "We are studying C programming." is yet another string. A string can also include symbols, for example,

"If you add x and y (x + y), you will get z (x + y = z)."

"A string may include symbols, such as $#@%&—."

In C, a string is nothing but an array of characters. Each element is `char`. The following is an example to create some strings.

```
/* string1.c */
#include <stdio.h>
int main(int argc, char * argv[])
{
  char str1[] = "Hello ECE264 Students";
  char * str2 = "This is another string.";
  char str3[] = {'a', 'b', 'x', 'y', '\0'};
  printf("str1 = %s\n", str1);
  printf("str2 = %s\n", str2);
  printf("str3 = %s\n", str3);
  return 0;
}
/*
  output:
  str1 = Hello ECE264 Students
  str2 = This is another string.
  str3 = abxy
*/
```

In C, printing a string uses `%s`. There are different ways to create strings. The first creates an array of `char`. The second uses a pointer to a *constant* array of `char` because `gcc` will create this array and assign the address to the pointer `str2`. The third creates another string. What is '\0'?

In C, each string **must** end with the special character '\0'. If you want to create a string "ECE264", you must have an array of 7 characters. The additional element stores the ending character '\0'. **It is a common mistake forgetting to add the ending character in a string.** C library has some functions to manipulate strings.

```
/* string2.c */
#include <string.h>
```

```
#include <stdio.h>
int main(int argc, char * argv[])
{
  char str1[] = "This is a string.";
  char * str2 = "Hello ECE264 Students";
  char str3[] = {'a', 'b', 'x', 'y', '\0'};
  char str4[] = "ECE264";
  char str5[80];
  printf("strlen(str1) = %d\n", strlen(str1));
  printf("strlen(str4) = %d\n", strlen(str4));
  printf("strcmp(str1, str2) = %d\n", strcmp(str1, str2));
  printf("strcmp(str1, str3) = %d\n", strcmp(str1, str3));
  printf("strcmp(str2, str3) = %d\n", strcmp(str2, str3));
  printf("strchr(str2, '4') = %s\n", strchr(str2, '4'));
  strcpy(str5, str2);
  printf("strcpy(str5, str2) = %s\n", str5);
  strcat(str5, str3);
  printf("strcat(str5, str3) = %s\n", str5);
  return 0;
}
/*
  output:
  strlen(str1) = 17
  strlen(str4) = 6
  strcmp(str1, str2) = 1
  strcmp(str1, str3) = -1
  strcmp(str2, str3) = -1
  strchr(str2, '4') = 4 Students
  strcpy(str5, str2) = Hello ECE264 Students
  strcat(str5, str3) = Hello ECE264 Studentsabxy
*/
```

- `strlen`: returns the length of the string, **without** counting the ending '\0'.
- `strcmp`: compares two strings based on the *dictionary order*. If the first string would appear before the second in a dictionary, the value is -1. If the two strings are the same, the value is zero. If the first would appear latter, the value is 1. The ASCII (American Standard Code for Information Interchange) value of 'T' is 84 and the value of 'a' is 97. Thus, the first string is smaller than the third one. In ASCII, A - Z are 65 - 90; a - z are 97 - 122.

  You can use `strncmp` with the third argument; this argument specifies the maximum number of characters to compare. If a string is shorter than this number,

`strncmp` compares up to only the ending character '\0'.

- `strchr` returns the address of the first appearance of the character in the second argument. It returns NULL if this character does not appear in the string. **You may find this function useful for PA1.**
- `strcpy` copies the second string to the first string. This function does **not** check whether the destination (first argument) has enough space. If the destination does not have enough space, the result is undefined (i.e. the program may crash).

  One solution is to copy as many characters as the space in the destination by using `strncpy`. This function has the third argument specifying the number of characters to copy. Suppose `buf` is an array of characters. The following code copies as many as allowed and **explicitly** adds the ending character to terminate the string.

  ```
  strncpy(buf, input, sizeof(buf) - 1);
  buf[sizeof(buf) - 1] = '\0';
  ```

  You must be very careful when you handle strings. Always terminates strings by adding '\0' when you are not sure.
- `strcat` concatenates the second string to the first string. C does **not** check whether the first string has enough space. You can use `strncat` to control the number of characters added to the destination (first) string.

One common security problem related to strings is called *buffer overflow attack*. This type attack gives ridiculously long inputs to crash a program or make it misbehave. The *Morris worm* in 1988 was one of the first large-scale attacks through the Internet using buffer overflow. Even though the worm did not intend to cause damage (such as erasing files), the worm grew so fast and shut down the Internet. Since then, numerous attackers have tried to use buffer overflow and in many cases succeeded because programmers forget to restrict the lengths of strings.

4 Modus Tollens

Consider this code segment

```
if (x == 0)
{
   y = 0;
}
```

After running this code, what can we say about x if we know y is not zero? We know that x must **not** be zero. This concept is called *modus tollens* (Latin).

What can we say about x if y is indeed zero? We do **not** know whether x is zero or not because y may be zero originally. Consider the following case:

```
int x = 3;
int y = 0;
if (x == 0)
{
   y = 0;
}
```

After this condition, y is zero even though x is **not** zero. This is a common mistake. If y is zero, you **cannot** say anything about x. If y is not zero, x must not be zero. When you debug a program and you know that y is zero, you **cannot** assume that x is also zero.

When x is not zero, we do **not** know whether y is zero or not.

```
int x = 3;
int y; /* can be zero or not zero */
if (x == 0)
{
   y = 0;
}
/* y may or may not be zero. We don't know. */
```

One way to understand this is the following: If it is raining, the street must be wet. If the street is dry, it must not be raining. If the street is wet, it may or **may not** be raining. The street can be wet for many reasons. It could be raining an hour ago. Maybe a water pipe bursts. Maybe a river overflows (heavy rain from upper stream or a dam is broken). Maybe it snowed yesterday and the snow is melting now. We do not know whether it is raining or not if the street is wet. We do know that it is **not** raining if the street is dry. Also, if it is not raining, we do not know whether the street is dry or wet.