ECE 264 Advanced C Programming

Contents

1 Stack and Heap Memory

When we talked about arrays, we said **an array is a pointer to the address of the first element**. When we use `malloc` to allocate memory, we can allocate a sufficient amount for multiple elements and create an array. The following two lines can server the same purpose as two arrays, each with 100 elements of integers.

```
  int b[100];
  int * c;
  c = malloc(100 * sizeof(int));
  b[8] = 11;
  c[72] = 6;
```

Does this mean arrays and pointers are equivalent? Consider the following example. What is the output of the program?

```
#include <stdlib.h>
#include <stdio.h>
int fa()
{
  int a = 15;
  return a;
}
int * fb()
{
  int b[100];
  int ind;
  for (ind = 0; ind < 100; ind ++)
    { b[ind] = ind; }
  return b;
}

int * fc()
{
  int * c;
  c = malloc(100 * sizeof(int));
  int ind;
  for (ind = 0; ind < 100; ind ++)
    { c[ind] = ind; }
  return c;
}

int main(int argc, char * argv[])
{
  int va;
  int * pb;
  int * pc;
  va = fa();
  pb = fb();
  pc = fc();
  printf("va = %d\n", va);
  printf("pb[68] = %d\n", pb[68]);
  printf("pc[68] = %d\n", pc[68]);
  /*
    free (pb);
    free (pc);
  */
  return 0;
}
```

The first function `fa` returns the value of a local variable `a`. The second function creates an array of 100 elements, assigns values, and then returns the array. The third function **allocates** memory for an array of 100 elements, assigns values, and then returns the pointer. Do you expect to see

```
va = 15
pb[68] = 68
pc[68] = 68
```

The difference between `fb` and `fc` is that `b` is a **local** array. When the function `fb` returns, the local array is **no longer available**. Hence, `pb` points to an invalid address. When a function is called, the following actions are taken:

1. The return location is pushed to the call stack.
2. The values of the arguments are pushed to the call stack.
3. The local variables are pushed to the call stack.
4. The program starts executing the code in the called function.

When a function returns, the following actions are taken:

1. The local variables are popped from the call stack.
2. The values of the arguments are popped from the call stack.
3. The returned value from the function is assigned to the caller's variable (if applicable).
4. The program starts executing the code at the return location.

Why do we need to push the arguments and the local variables to the stack? Consider the following situations:

```
f1(int x) /* x1 */              f2(int x) /* x2 */
{                               {
   int a = 5; /* a1 */             int a = 10; /* a2 */
   f2(7);                          f3(9);
   /* value of a? */               /* value of a? */
   /* value of x? */               /* value of x? */
}                               }
```

When `f3` returns, `a` (`a2`) is 10 and `x` (`x2`) is still 7. When `f2` returns, `a` (a1) is 5. We need to keep the values of the caller's arguments and local variables before calling a function. Based on this understanding, we know that `b` is no longer valid after function `fb` returns. In fact, you would receive a compiler warning:

```
stackheap.c: In function 'fb':
stackheap.c:14: warning: function returns address of local variable
```

This is another reason why **you should not ignore warning messages.**

How is `c` different? Its memory is created by calling `malloc`. This memory does **not** reside inside the stack. Instead, it is somewhere else (called *heap*). This memory is **not** released when the function returns. Hence, `c` is still valid and `pc` points to a valid piece of memory. The output of this program is

```
va = 15
pb[68] = 143626240
pc[68] = 68
```

The value of `pb[68]` is not predictable; it is simply garbage. If you execute the program several times, you will likely see different values.

If `c` (and `pc`) is still valid after the function `fc` returns, does this cause memory leak? Yes. You should release the memory

```
free (pc);
```

at the end of `main`.

How about calling

```
free (pb);
```

at the end of `main`? The program will crash since `pb` is pointing to invalid memory.

Now, you should understand why you should **never** write a function like this

```
void fc()
{
  int * c;
  c = malloc(100 * sizeof(int));
}
```

The memory allocated to `c` is still **valid** after the function returns but **no longer accessible**. This memory is leaked.

Why is `va` 5? Isn't `a` popped from the stack? Yes, that is correct. However, one of the steps when a function returns is to copy the returned value. This is **valid when the returned type is a primitive type** (such as `int`, `char`, or `double`). When the returned type is a pointer, the **address** is returned to the pointer. If the address is invalid, the pointer points to an invalid location, such as the case of `pb`.

2 Divide and Conquer

Quiz and Discussion: Consider an array of 1000 distinct elements. For example,

```
a[0] = 9
a[1] = 17
a[2] = 28
a[3] = -4
a[4] = 1
a[5] = 113
...
```

How do you determine whether a number, say 1526, is an element of this array? Please write down the procedure (in English or in C code). On average, how many elements do you have to check before determining that the number is or is not in the array?

Quiz and Discussion (need volunteers to share their solutions): If these elements are **sorted**: `a[i] < a[j]` if $1 \leq i < j \leq 1000$, how do you determine whether a number is an element of this array?

```
a[0] = -4
a[1] = 1
a[2] = 9
a[3] = 17
a[4] = 28
a[5] = 113
...
```

Can you do it faster than the method presented earlier? Please write down the procedure (in English or in C code).

In the first case, we have no information about the **relationships** among the elements; hence, we have to check the elements one by one.

In the second case, we can check the element in the middle (`a[500]` or `a[499]`, does not really matter). If the element is larger than the number we are checking, we **can discard the second half of the array** because all elements are larger than the give number. Similarly, if `a[500]` is smaller than the number we are checking, we **can discard the first half of the array** because all elements are smaller than the give number.

The general principle is to

1. If the problem is simple enough, solve it directly.
2. Otherwise, divide the problem into some parts. These parts together is the original problem.
3. Determine whether we can discard one or some parts **without** doing anything additional in these parts
4. Process the remaining part (or parts)
5. If the remaining part (or parts) is still too large, repeat steps 1 - 3

This strategy is called **divide and conquer**.

In this example, the elements are divided into two parts. Because the elements are sorted, we can discard one part in a single step.

1. After checking one element (`a[500]`), we can discard half (500) of the elements. Only 499 elements are left.
2. After checking another element (`a[250]` or `a[750]`), we can discard another 250 elements. Only 249 elements are left.
3. After checking yet another element, we can discard another 125 elements. Only 124 elements are left.

Each step reduces the number of elements by half. The following is an example of the two search functions. The first one checks elements one by one.

```
/* The array is not sorted */
#include <stdio.h>
int search1(int data[], int size, int value)
/* return -1 if not found, index if found */
/* check each element until
   1. found
   2. no more element left
*/
{
  int ind;
  for (ind = 0; ind < size; ind ++)
    {
      if (data[ind] == value)
        {
          return ind;
        }
    }
  return -1; /* not found */
}
```

The next function discards half of the data in each step. This is called **binary search**.

```
/* the array is sorted */
#include <stdio.h>
int search2(int data[], int size, int value)
/* return -1 if not found, index if found */
{
  int index;
  int head = 0;
  int tail = size - 1;
  while (head <= tail)
    {
      index = (head + tail) / 2;
      printf("check index = %d\n", index);
      if (data[index] == value) /* found */
        { return index; }
      if (data[index] > value) /* discard the second half */
        {
          tail = index - 1;
        }
      else
        {
          head = index + 1;
        }
    }
  return -1; /* not found */
}
```

The following file contains the `main` function.

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
int search1(int data[], int size, int value);
int search2(int data[], int size, int value);

void print(int data[], int size)
```

```
{
  int ind;
  for (ind = 0; ind < size; ind ++)
    {
      printf("%d ", data[ind]);
    }
  printf("\n");
}

int compare(const void * p1, const void * p2)
/* comparison function needed by qsort */
{
  int v1 = * (const int *) p1;
  int v2 = * (const int *) p2;
  if (v1 > v2) { return 1; }
  if (v1 < v2) { return -1; }
  return 0;
}

int main(int argc, char * argv[])
{
  int numElement = 0;
  srand(time(0));
  if (argc > 1)
    {
      numElement = (int)strtol(argv[1], (char **)NULL, 10);
    }
  if (numElement < 100) { numElement = 100; }
  int * data = malloc(numElement * sizeof(int));
  int ind;
  /* initialize the elements */
  for (ind = 0; ind < numElement; ind ++)
    {
      data[ind] = numElement - ind;
    }
  print(data, numElement);
  int value = rand() % (5 * numElement);
  printf("search %d, index = %d\n\n", value,
         search1(data, numElement, value));
  qsort(data, numElement, sizeof(int), compare);
  /* qsort (quicksort) is provided by C but we have to provide a
     comparison function. Please check the manual for details */
  print(data, numElement);
  printf("search %d, index = %d\n", value,
         search2(data, numElement, value));
  /* The index may be different from search1 because the elements have
     been sorted. */
  free (data);
  return 0;
}
```

If there are 7 elements (from `a[0]` to `a[6]`), we need at most three steps

1. check `a[3]`
2. check `a[1]` or `a[5]`
3. check `a[0]` or `a[2]` or `a[4]` or `a[6]`

If there are 15 elements, we need at most 4 steps. When there are $2^n - 1$ elements, we need at most n steps. This is the definition of **logarithm**.

$$\log_2 n = j, \text{ if } 2^j = n. \tag{1}$$

Therefore, $\log_2 4 = 2$, $\log_2 8 = 3$, $\log_2 16 = 4$, and $\log_2 32 = 5$. Sometimes, we write

$$\lg n = \log_2 n. \tag{2}$$

The logarithm function grows very slowly as n increases. For example, $\lg 1,048,576 = 20 \approx \lg 1,000,000$. **If you can solve a problem with n elements using only $\lg n$ steps, the solution is usually considered very efficient.**

To make divide and conquer efficient, you should divide the original problem into **non-overlapping** parts so that you can discard some parts quickly. **Divide and conquer** is, arguably, the **most important strategy in computing** because it allows us to handle very large sets of data within very short time.

Using binary search, we can use at most n comparisons to determine whether a number is in an array of $2^n - 1$ elements. To help you appreciate the strength of this relationship, please consider this example. A typical computer can compare 1,000 integers within a millisecond. This allows us to determine whether an integer is in an array of $2^{1,000} - 1$ elements. How large is this array? It has approximately 10^{301} elements. How large is this number indeed? An estimate shows that there are 10^{58} atoms in this solar system and 10^{80} atoms in this universe. Even if you used one atom to store one array element, you would run out atoms by a very far distance of 10^{221}. If you could have this array, however, you would be able to perform a binary search within a millisecond on your desktop computers.

3 Divide and Conquer with Recursion

How do we implement divide and conquer in programs? There are many ways, one shown in binary search earlier. Another way to implement divide and conquer is using **recursion, a function that calls itself**. The following is the general format of a recursive function:

```
function(data)
{
   if (data simple enough)
   {
       solve the problem directly; /* conquer */
       return;
   }
   /* else not needed since return is used inside if */
   divide the data into smaller parts; /* divide */
   discard the parts that are no longer needed;
   for each part of the remaining data
   {
       function(part);
   }
}
```

We have seen functions many times. Have you ever thought about a *function calling itself*? Why would we do that? Consider this definition of *factorial*

$$f(n) = \begin{cases} 1 & \text{if } n = 1 \\ n \cdot f(n-1) & \text{otherwise.} \end{cases} \tag{3}$$

Where does factorial come from? Why do we care about it? It comes from *permutation*. Suppose you have five students and want to assign five seats for an exam. How many possible assignments do you have? The first student has five choices. The second has four choices. The third has three choices. There are totally $5 \times 4 \times 3 \times 2 \times 1 = 120$ options. This is where factorial comes from. We can implement factorial by recursion:

- We know how to solve the simplest case, $f(1) = 1$.
- We know how to break a more complex case into simpler cases, $f(n) = n \cdot f(n-1)$.
- We solve the problem by *recursively* dividing the problem into smaller and simpler cases and eventually reach the simplest case.
- With the solution of the simplest case, we go back to gradually build the solution of the more complex case and eventually solve the original problem.

```c
#include <stdlib.h>
#include <stdio.h>
unsigned int factorial1(unsigned int n)
{
  /* using recursion */
  if (n == 1)
    { return 1; }
  return (n * factorial1(n - 1));
}

unsigned int factorial2(unsigned int n)
{
  /* without recursion */
  int product = 1;
  int cnt;
  for (cnt = 1; cnt <= n; cnt ++)
    { product *= cnt; }
  return product;
}

int main(int argc, char * argv[])
{
  unsigned int n;
  if (argc < 2)
    {
      printf("need a number\n");
      return -1;
    }
  n = strtol(argv[1], (char **)NULL, 10);
  printf("f(%d) = %d = %d\n", n,
         factorial1(n), factorial2(n));
  return 0;
}
```