ECE 264 Advanced C Programming

Contents

1 Memory Allocation by Function

We can also allocate memory inside a function. When the function finishes, the memory space is returned to the caller. **The caller is responsible for releasing the memory.**

```
/* mallocfunc.c */
/* read two vectors, sort their elements together */
#include <stdlib.h>
#include <stdio.h>
int read2Vector(char * fileName, int * * vec1, int * * vec2)
{
  int numElem;
  int val1;
  int val2;
  int elemCnt = 0;
  FILE * fptr = fopen(fileName, "r");
  (*vec1) = 0; /* make sure it is invalid */
  (*vec2) = 0;
  if (fptr == NULL)
    {
      printf("cannot read file %s\n", fileName);
      return 0;
    }
  fscanf(fptr, "%d", & numElem);
  printf("%d elements\n", numElem);
  * vec1 = malloc(numElem * sizeof(int));
  * vec2 = malloc(numElem * sizeof(int));
  if (((* vec1) == NULL) || ((* vec2) == NULL))
    {
      printf("memory allocation fail\n");
```

```
      return -1;
    }
  while ((elemCnt < numElem) && (! feof(fptr)))
    {
      fscanf(fptr, "%d %d", & val1, & val2);
      (*vec1)[elemCnt] = val1;
      (*vec2)[elemCnt] = val2;
      elemCnt ++;
    }
  fclose(fptr);
  return numElem;
}

void print2Vector(int * vec1, int * vec2, int numElem)
{
  int elemCnt;
  for (elemCnt = 0; elemCnt < numElem; elemCnt ++)
    {
      printf("%d %d\n", vec1[elemCnt], vec2[elemCnt]);
    }
}

int main(int argc, char * argv[])
{
  int * v1ptr;
  int * v2ptr;
  int numElem;
  if (argc < 2)
    {
      printf("need file name and\n");
      return -1;
    }
  numElem = read2Vector(argv[1], & v1ptr, & v2ptr);
  if (numElem < 0) { return -1; }
  print2Vector(v1ptr, v2ptr, numElem);
  free(v1ptr);
  free(v2ptr);
  return 0;
}
```

Why do we need to use two asterisks for the arguments? Why do we need to use ampersands when calling `read2Vector`?

```
int read2Vector(char * fileName, int * * vec1, int * * vec2)
...
int * v1ptr;
int * v2ptr;
numElem = read2Vector(argv[1], & v1ptr, & v2ptr);
```

If `x` is an integer, `&x` is the address. If we want to change `x`'s value in a function, we need to pass its address to the function and use `* x = val;`. Remember an array is represented by the address of the first element. This address is stored as a *pointer*. That is the reason we declare `v1ptr` and `v2ptr` as pointers.

We do not know where they point to because `malloc` will allocate memory and tell us. In the earlier example (sortvector), we use

```
vecptr = malloc(2 * numElem * sizeof(int));
```

to assign where `vecptr` points to.

When a function allocates memory, we cannot pass `v1ptr` directly to `read2Vector` and modify `* vec1`. If we do so, we are changing the value stored at the location pointed by `vec1`. However, `vec1` is not pointing anywhere at this moment yet. This will cause the program to crash.

Instead, we have to modify where `vec1` points to. That means, modifying the value of `* vec1`. In order to modify `* vec1`, we have to pass `* * vec1` as the argument.

```
int read2Vector(char * fileName, int * * vec1, int * * vec2)
* vec1 = malloc(numElem * sizeof(int));
...
int * v1ptr;
int * v2ptr;
numElem = read2Vector(argv[1], & v1ptr, & v2ptr);
```

2 Multidimensional Arrays

We have seen this line many times

```
int main(int argc, char * argv[])
```

We know the first argument is an integer. What is the second argument? It is something related to a pointer since there is "*". It is also something related to an array because of "[]". What is it? It means `argv` is a *two-dimensional array* of characters.

Why do we need multi-dimensional arrays? Consider an example when you want to list the travel (flight) time between pairs of cities.

To \ From	Boston	Chicago	New York	San Francisco
Boston	-	150	60	270
Chicago	140	-	130	240
New York	60	140	-	290
San Francisco	260	230	280	-

(The flight time may be unsymmetrical due to wind.)

It is naturally expressed by a two-dimensional array. If you want to express the location, you may want to use a three-dimensional array for x, y, and z directions.

The following are examples to declare and define multi-dimensional arrays

```
/* mdarray1.c */
int a[100];
/* 1 dimensional, 100 elements */

double b[10][6];
/* 2 dimensional, 60 = 10 x 6 elements */

int c[5][3][2];
/* 3 dimensional, 30 = 5 x 3 * 2 elements */

double v[5][5][5];
/* 3 dimensional, 125 = 5 * 5 * 5 elements */
```

The following example shows how to allocate memory for a 2-dimensional array (i.e. matrix). When we allocate a high-dimensional array, we have to do it **one dimension at a time**.

```
/* arraypointer.c */
/* This example is related to Sec 12.6 of ABoC. */
#include <stdlib.h>
#include <stdio.h>

void printMatrix(int * * ma, int row, int col)
```

```
{
  int rcnt;
  int ccnt;
  for (rcnt = 0; rcnt < row; rcnt ++)
    {
      for (ccnt = 0; ccnt < col; ccnt ++)
        {
          printf("%5d ", ma[rcnt][ccnt]);
        }
      printf("\n");
    }
  printf("\n");
}

int main(int argc, char * argv[])
{
  int numRow;
  int numCol;
  int row;
  int col;
  int * * matrix;
  if (argc < 3)
    {
      printf("need numRow and numCol\n");
      return -1;
    }
  numRow = (int)strtol(argv[1], (char **)NULL, 10);
  numCol = (int)strtol(argv[2], (char **)NULL, 10);
  if ((numRow <= 0) || (numCol <= 0))
    {
      printf("both numbers must be positive\n");
      return -1;
    }
  matrix = malloc(numRow * sizeof(int *));
  if (matrix == 0)
    {
      printf("memory allocation fail\n");
      return -1;
    }
  for (row = 0; row < numRow; row ++)
    {
      matrix[row] = malloc(numCol * sizeof(int));
```

```
      if (matrix[row] == 0)
        {
          printf("memory allocation fail\n");
          return -1;
        }
    }
  for (row = 0; row < numRow; row ++)
    {
      for (col = 0; col < numCol; col ++)
        {
          matrix[row][col] = row + col;
        }
    }
  printMatrix(matrix, numRow, numCol);

  for (row = 0; row < numRow; row ++)
    {
      for (col = 0; col < numCol; col ++)
        {
          matrix[row][col] *= 2;
        }
    }
  printMatrix(matrix, numRow, numCol);

  for (row = 0; row < numRow; row ++)
    {
      free(matrix[row]);
    }
  free(matrix);
  return 0;
}
```

Each matrix is a pointer of arrays. Since each array is also a pointer, a matrix is declared as pointers to pointers

```
      int * * matrix;
```

This program first creates an array of **pointers**:

```
        matrix = malloc(numRow * sizeof (int *));
```

Notice `int *` inside `sizeof`. Then it allocates an array for each pointer. Each row points to a piece of memory for storing the elements:

```
for (row = 0; row < numRow; row ++)
  {
    matrix[row] = malloc(numCol * sizeof (int));
  }
```

Each element is initialized to the sum of the row and the column indexes.

```
matrix[row][col] = row + col;
```

Before the program ends, we need to release the memory. Each row is released first and then the pointers to the rows are released. **Memory allocation and release are usually symmetric.**

Exercise How to allocate space to store a list of strings? Your PA1 probably needs this.

The following means `argv` is an array of strings. Since each string is an array of characters, `argv` is a 2-dimensional array.

```
int main(int argc, char * argv[])
```