ECE 264 Advanced C Programming

2009/01/30

Contents

1 File Operations

Most of our programs, up to this point, do the same things over and over again. The sizes of the arrays are fixed. The values of the elements are fixed. These programs are not particularly exciting. To make these programs more useful, we must be able to handle different sizes of arrays, with different element values.

We can use `scanf` to input an integer value by a user. However, this is useful if the program needs only several numbers. If the program needs many numbers, any user will soon run out of patience. Moreover, a very high percentage of data are stored in computers already. It makes no sense to ask anyone to enter the data by hand again. What we need is the ability to read and write data using *files*. The following program writes six integers to a file and then reads the values back from the same file.

```
/* file1.c */
#include <stdio.h>
void writeFile(char * fileName, int * array, int numElem)
{
  FILE * fptr = fopen(fileName, "w");
  int cnt;
  if (fptr == NULL)
    {
      printf("cannot write file %s\n", fileName);
      return;
    }
  for (cnt = 0; cnt < numElem; cnt ++)
    {
      fprintf(fptr, "%d\n", array[cnt]);
    }
  fclose(fptr);
}

void readFile(char * fileName)
{
  FILE * fptr = fopen(fileName, "r");
  int val;
  if (fptr == NULL)
    {
      printf("cannot read file %s\n", fileName);
      return;
    }
  while (! feof(fptr))
    {
      fscanf(fptr, "%d", & val);
      printf("%d\n", val);
    }
  fclose(fptr);
}

int main(int argc, char * argv[])
{
  int array[] = {2, 6, 4, 2, 0, 9};
  if (argc < 2)
    {
      printf("need file name\n");
      return -1;
    }
  writeFile(argv[1], array, sizeof(array) / sizeof(int));
  readFile(argv[1]);
  return 0;
}
```

C provides a type called `FILE`. We use `fopen` to open a file; the first argument is the file's name and the second argument is the mode. The following modes are supported

"r"	open to read
"w"	open to write
"a"	open to append
"rb"	open to read in binary mode
"wb"	open to write in binary mode
"ab"	open to append in binary mode
"r+"	both reading and writing

If `fopen` fails, it returns `NULL` (zero). This call may fail for many reasons, such as (1) the file does not exist or (2) the file exits but the user has not right to read or to write. You should **always check the returned value of `fopen` before doing anything related to the file.** If you do not check, the program may crash because it tries to read a file that does not exist. We can use `fprintf` to print values to a file. The first (additional) argument is a pointer to a file, obtained by calling `fopen` earlier. After we finish printing to the file, call `fclose` to flush the output and close the file. In most systems, writing to a file does not occur to the physical storage (such as a disk) immediately. Instead, the output is stored in a buffer (i.e. memory). This can significantly improve performance because the next write may occur soon. Writing immediately to a disk can slow down a program by thousands of times. Calling `fclose` ensures that all data in the buffer are flushed to the disk so that the data are not lost. You should **always call `fclose` when the file is no longer needed.** Reading a file is symmetric to writing a file. When we write, we use `fprintf`. When we read, we use `fscanf` with the first argument as the file pointer. The function `feof` (end-of-file) returns one if we have reached the end of the file.

In addition to `fprintf` and `fscanf`, there are many other functions to write to or to read from files. Function `fgets` reads a string of n (second argument) characters. (Check the manual "man fgets".) We can write or read one character each time by using `putc` and `getc`. The following program count the occurrence of character 'e' in a file.

```
/* counte.c
   count the occurrence of 'e' */
#include <stdio.h>
int main(int argc, char * argv[])
{
  FILE * fptr;
  int ch;
  int counter = 0;
  if (argc < 2)
    {
      printf("need file name\n");
      return -1;
    }
  fptr = fopen(argv[1], "r");
  if (fptr == NULL)
    {
      printf("open file fail\n");
      return -1;
    }
  while ((ch = getc(fptr)) != EOF)
    {
      if (ch == ’e’)
        { counter ++; }
    }
  fclose (fptr);
  printf("e appears %d times\n", counter);
  return 0;
}
```

Exercise (Potential Exam Question): Write a program that can count the occurrence of ′a′ followed by any one character followed by ′c′. The *regular expression* is `a?c`. The symbol ? means any character (and exactly one character). Do not count if ′a′ and ′c″ are in two different lines.

The following program generates two vectors and writes the elements into a file. The file′s name is one input argument and the size of the vector is another input argument.

```
/* genvector.c */
/* generate two vectors of the same size */
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>
void generate2Vector(char * fileName, int numElem)
{
  FILE * fptr = fopen(fileName, "w");
  int cnt;
  if (fptr == NULL)
    {
      printf("cannot write file %s\n", fileName);
      return;
    }
  fprintf(fptr, "%d\n", numElem);
  for (cnt = 0; cnt < numElem; cnt ++)
    {
      fprintf(fptr, "%d %d\n", rand() % 1000, rand() % 1000);
    }
  fclose(fptr);
}

int main(int argc, char * argv[])
{
  struct timeval currTime;
  gettimeofday(&currTime, NULL);
  srand(currTime.tv_usec);

  if (argc < 3)
    {
      printf("file name and number of elements\n");
      return -1;
    }
  generate2Vector(argv[1], (int)strtol(argv[2], (char **)NULL, 10));
  return 0;
}
/*
  output (one instance)
  17
  299 535
  553 941
  115 618
  112 114
  548 410
  922 390
  342 552
  162 140
  177 186
  257 960
  634 62
  969 788
  213 433
  903 883
  472 732
  619 252
  193 772
*/
```

The next program reads the elements, add each pair, and print the sums.

```
/* addvector.c */
/* add two vectors */
#include <stdlib.h>
```

```
#include <stdio.h>
void add2Vector(char * fileName)
{
  FILE * fptr = fopen(fileName, "r");
  int numElem;
  int val1;
  int val2;
  int sum;
  int elemCnt = 0;
  if (fptr == NULL)
    {
      printf("cannot read file %s\n", fileName);
      return;
    }
  fscanf(fptr, "%d", & numElem);
  printf("%d elements\n", numElem);
  while ((elemCnt < numElem) && (! feof(fptr)))
    {
      fscanf(fptr, "%d %d", & val1, & val2);
      sum = val1 + val2;
      printf("%d + %d = %d\n", val1, val2, sum);
      elemCnt ++;
    }
  fclose(fptr);
}

int main(int argc, char * argv[])
{
  if (argc < 2)
    {
      printf("need file name and\n");
      return -1;
    }
  add2Vector(argv[1]);
  return 0;
}
/*
  output (one instance)
  17 elements
  299 + 535 = 834
  553 + 941 = 1494
  115 + 618 = 733
  112 + 114 = 226
  548 + 410 = 958
  922 + 390 = 1312
  342 + 552 = 894
  162 + 140 = 302
  177 + 186 = 363
  257 + 960 = 1217
  634 + 62 = 696
  969 + 788 = 1757
  213 + 433 = 646
  903 + 883 = 1786
  472 + 732 = 1204
  619 + 252 = 871
  193 + 772 = 965
  193 + 772 = 965
*/
```

2 Array of Unknown Size: Dynamic Memory Allocation

The previous program can handle different sizes of vectors. This is a great improvement. However, sometimes we want to *keep the elements* for later use. For example, we may want to sort the elements. It will be very helpful if we can store the elements in arrays, since we already know how to sort elements in arrays. We are now ready to handle arrays whose sizes are not known when the program is written. The sizes are known when the program starts running. We are going to use `malloc` to allocate memory for the arrays.

```
/* sortvector.c */
/* read two vectors,
   sort their elements together */
#include <stdlib.h>
#include <stdio.h>

void swap(int *p, int *q)
{
  int   tmp;
  tmp = *p;
  *p = *q;
  *q = tmp;
}

void bubbleSort(int a[], int n)
```

```
{
  int i, j;
  for (i = 0; i < n - 1; ++i)
    /* same as i ++ */
    {
      for (j = i + 1; j < n; j++)
        {
          if (a[i] > a[j])
            {
              swap(&a[i], &a[j]);
            }
        }
    }
}

int main(int argc, char * argv[])
{
  int * vecptr;
  int elemCnt = 0;
  int numElem;
  int val1;
  int val2;
  FILE * fptr;
  if (argc < 2)
    {
      printf("need file name and\n");
      return -1;
    }
  fptr = fopen(argv[1], "r");
  if (fptr == NULL)
    {
      printf("cannot read file %s\n", argv[1]);
      return -1;
    }
  fscanf(fptr, "%d", & numElem);
  printf("%d elements\n", numElem);
  vecptr = malloc(2 * numElem * sizeof(int));
  if (vecptr == NULL)
    {
      printf("memory allocation fail\n");
      return -1;
    }
  while ((elemCnt < numElem) && (! feof(fptr)))
    {
      fscanf(fptr, "%d %d", & val1, & val2);
      vecptr[2 * elemCnt] = val1;
      vecptr[2 * elemCnt + 1] = val2;
      elemCnt ++;
    }
  fclose(fptr);
  bubbleSort(vecptr, 2 * numElem);
  for (elemCnt = 0; elemCnt < 2 * numElem; elemCnt ++)
    {
      printf("%d ", vecptr[elemCnt]);
    }
  printf("\n");
  free(vecptr);
  return 0;
}

/*
  17 elements
  62 112 114 115 140 162 177 186 193 213 252 257 299 342 390 410 433 472
  535 548 552 553 618 619 634 732 772 788 883 903 922 941 960 969
*/
```

C has two functions to allocate memory: `malloc` and `calloc`. The former takes one argument and the latter takes two.

```
     malloc(n * sizeof(int));
     calloc(n, sizeof(int));
```

The former is more commonly used because `calloc` initializes the allocated memory to zero. In most cases, this is simply wasting time. In our program, after the memory is allocated, the elements are assigned using the values from the file. It is unnecessary to initialize the memory to zero.

When you call `malloc`, you should always check whether the allocation is successful. If it fails, take the appropriate steps to handle it. If you do not check whether the allocation is successful, the program will crash when it tries to use the memory.

After allocating memory, we can use it as the arrays that we have seen before. When this space is no longer needed, **call `free` to release the space.** A typical structure of using dynamically allocated memory is

```
    int * intPtr;
    ...
    intPtr = malloc(numElem * sizeof(int));
    ... /* use the array */
    free(intPtr);
```

Releasing memory isn't really a great deal in this very simple program. When your programs become more complex, **memory management can easily become a major source of mistakes.** Very often, we *reuse* a pointer by assigning it to point to somewhere else. If we do not release the memory before the assignment, that piece of memory can no longer be reached. This is called *memory leak*.

```
    int * intPtr;
    ...
    intPtr = malloc(numElem * sizeof(int));
    ... /* no free */
    intPtr = malloc(numElem * sizeof(int));
    /* memory leak, the previous memory space is lost */
```

For the operating system, the memory still belongs to the program. Memory leak is a silent killer of programs. The total amount of available memory space is finite. If a program leaks memory, the available memory gradually shrinks. Eventually, the operating system will refuse the allocate more memory (`malloc` returns 0) and the program will likely crash. Modern computers usually have large (virtual) memory space. It can take weeks for a program to run out of memory. A program may execute without any problem for weeks and then suddenly crash.

Fortunately, there are tools checking memory leak. In Linux, `valgrind` can check memory leak. Suppose `sortvector` is the name of the program, we can check whether there is memory leak by using this command

```
     valgrind --leak-check=yes ./sortvector data.in
```

If there is no problem, the output is something like this

```
==21608== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 12 from 1)
==21608== malloc/free: in use at exit: 0 bytes in 0 blocks.
==21608== malloc/free: 2 allocs, 2 frees, 488 bytes allocated.
==21608== For counts of detected errors, rerun with: -v
==21608== All heap blocks were freed -- no leaks are possible.
```

If we remove `free(vecptr);`, the report is

```
==21671== LEAK SUMMARY:
==21671==    definitely lost: 136 bytes in 1 blocks.
==21671==      possibly lost: 0 bytes in 0 blocks.
==21671==    still reachable: 0 bytes in 0 blocks.
==21671==         suppressed: 0 bytes in 0 blocks.
==21671== Reachable blocks (those to which a pointer was found) are not show
==21671== To see them, rerun with: --show-reachable=yes
```

Not surprisingly, we leak 17 elements of double, $8 \times 17 = 136$ bytes. **You should always check your program leaks memory.**

Some languages, such as Java, have built-in *garbage collection*. After a piece of memory is allocated and later becomes unreachable (called *garbage*), the languages will reclaim that piece of memory. Programmers do not have to explicitly release memory. C does **not** collect garbage because garbage collection can (1) slow down a program and (2) make a program's execution time less predictable.

3 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);
```

4 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 */
```