ECE 264 Advanced C Programming

Contents

1 Data Type

We have seen several types in C already, including integer, character, and string. In fact, C does not have real string. In C, **a string is an array of characters**. We will talk about arrays a littler later. Why do we need different data types in C? First, different types have different sizes. For example, `float` is single-precision floating point and `double` is double-precision floating point. The maximum value of `float` (4 bytes) is approximately 10^{38}. The maximum value of `double` (8 bytes) is approximately 10^{308}. If `double` can hold more numbers, why do we use `float`? **Performance**. Double-precision operations usually are much slower. If you do not need the precision, you can improve performance by using `float`. Another reason of different types is the operations. When two integers are added, they are added bit by bit. When two floating point numbers are added, they have to be aligned by their decimal points first. C does not explicitly define the size of each type, even though a character is usually one byte, or 8 bits. It is value is -128 to 127. An integer is usually 4 bytes, between -2 billion (approximately) to 2 billion (approximately). You can use `sizeof(int)` to find out the size (bytes) of an integer.

What is the *possible* result of the following statement?

```
double a = .... /* a very large number */
double b = .... /* a very small number */
double c = a + b - a; /* Is c the same as b? */
```

No, `c` may be different from `b` because `b` is lost in `a + b` (`a` is too large).

In C, each character is also an integer, based on it corresponding ASCII (American Standard Code for Information Interchange) value, for example, 'a' is is 97 and 'A' is 65. In C, a control character starts with backslash \ (also called escape character). A newline is '\n' and a tab is '\t'.

2 Array

Computer programs are very good at doing the same operations on different data. For example, adding two vectors, each with 20 elements.

```
/* array.c */
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
int main(int argc, char * argv[])
{
  const int vecSize = 20;
  int veca[vecSize];
  int vecb[vecSize];
  int vecSum[vecSize];
  int cnt;
  struct timeval currTime;
  gettimeofday(&currTime, NULL);
  srand(currTime.tv_usec);
  /* use the current microsecond to
     initialize the random number */
  for (cnt = 0; cnt < vecSize; cnt ++)
    {
      veca[cnt] = rand() % 1000; /* between 0 and 999 */
      vecb[cnt] = rand() % 1000;
    }
  for (cnt = 0; cnt < vecSize; cnt ++)
    {
      vecSum[cnt] = veca[cnt] + vecb[cnt];
    }
```

```
  for (cnt = 0; cnt < vecSize; cnt ++)
    {
      printf("%3d + %3d = %4d\n",
             veca[cnt], vecb[cnt], vecSum[cnt]);
    }
  return 0;
}
```

This program introduces many new concepts. First, we declare a constant using `const`. By adding `const`, the compiler will detect whether you accidentally modify the value. A major source of problems in programs is accidental modification. **Use the compiler to help you detect errors.** The next three lines declare and define three arrays of integers. An array is like a vector. Each array has 20 elements. In C, when an array is declared and defined, **the elements' values are not initialized.** It is your responsibility to initialize the elements. In fact, you should always initialize the elements. **Using uninitialized elements is a common mistake.** Worse, your program may run correctly sometimes (depending on the computers). The next line declares an integer counter.

We are going to assign the elements by using random numbers. To make the elements truly random, we set the seed of the random number generator by calling `srand`. The seed has to be something that is hard to predict. We will use the current microsecond as the seed. This is achieved by calling `gettimeofday`. This function returns a structure of two fields: second and microsecond. We will talk about structures later. Right now, just remember that this is a way to set the random number generator. The next two lines initialize the elements in the two vectors. This line says

1. initialize `cnt` to zero.
2. If `cnt`'s value is smaller than vecSize, execute the code inside the curly bracket { ... }. If the condition is false, jump to the code after the close bracket }.
3. after finishing one iteration, add one to `cnt`, go back to step 2.

The `for` line has three parts, separated by two semicolons. The part before the first semicolon is the initialization and executes only **once**. The part between the two semicolons is the condition and it is checked **before** running the code inside the bracket. If the condition is satisfied, the code inside the brackets is executed. If the condition is false, jump to the code after the pair of brackets. The third part is executed **after each iteration**. In this example, the value of `cnt` increases by one after each iteration.

In C,

```
cnt ++; /* plus plus */
```

means taking the current value of `cnt`, adding one, putting the new value back to `cnt`. If the original value is 7, the new value is 8.

Similarly,

```
cnt --; /* minus minus */
```

means taking the current value of `cnt`, subtracting one, putting the new value back to `cnt`.

You can also use the following statements

```
cnt += 1; /* same as cnt ++ */
cnt -= 1; /* same as cnt -- */
```

You can replace 1 by another number

```
cnt += 5; /* same as cnt ++ five times */
cnt -= 3; /* same as cnt -- three times */
```

The variable `cnt` is called the *array index*. In C, the **index starts from zero and ends at size - 1 (inclusive)**. It is **not** between 1 and size. This is a common mistake.

The next `for` block takes each pair of elements from the two vectors, adds them, and stores the result in `vecSum`. Finally, the last `for` block prints the result. We can specify the number of digits used for each field by using "%3d". In this case, 3 digits are used for each element in `veca` or `vecb`. This is sufficient because each element is between 0 and 999. Each element in `vecSum` can be as large as 1998 so we give four digits. The output of one execution is

```
255 + 420 =  675
355 + 672 = 1027
658 + 461 = 1119
925 + 765 = 1690
423 + 791 = 1214
 68 + 282 =  350
937 +  79 = 1016
405 + 917 = 1322
125 + 394 =  519
664 + 885 = 1549
```

```
276 + 346 =  622
361 +  75 =  436
984 + 682 = 1666
966 + 989 = 1955
384 + 187 =  571
281 + 640 =  921
607 + 636 = 1243
664 + 266 =  930
449 + 590 = 1039
383 + 872 = 1255
```

If you execute the same program several times, you will see different numbers because we use a random number generator. In this example, we do not use the values of `argc` nor `argv`. In a C program, a function does **not have** to use the arguments. When you compile a program, the compiler will give a warning message.

3 Array and Function

If a program can only add 20 elements of random values, the program is not particularly interesting. We are going to create a function that can add arrays of different sizes.

```
/* array2.c */
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
int addVector(int * va, int * vb, int * vsum, int size)
{
  int cnt;
  for (cnt = 0; cnt < size; cnt ++)
    {
      vsum[cnt] = va[cnt] + vb[cnt];
    }
  return 0;
}
int main(int argc, char * argv[])
{
  const int vecSize = 20;
  int veca[vecSize];
  int vecb[vecSize];
  int vecSum[vecSize];
```

```
  int cnt;
  struct timeval currTime;
  gettimeofday(&currTime, NULL);
  srand(currTime.tv_usec);
  for (cnt = 0; cnt < vecSize; cnt ++)
    {
      veca[cnt] = rand() % 1000;
      vecb[cnt] = rand() % 1000;
    }
  addVector(veca, vecb, vecSum, vecSize);
  for (cnt = 0; cnt < vecSize; cnt ++)
    {
      printf("%3d + %3d = %4d\n",
             eca[cnt], vecb[cnt], vecSum[cnt]);
    }
  return 0;
}
```

The function `addVector` takes four arguments: the first two are the input arrays, the third is the output array, and the last is the size of the arrays. What do these asterisks mean in the arguments? Before explaining this, we need to understand arrays and pointers in C.

4 Array and Pointer

In C, **an array's name represents the starting location of a continuous piece of memory**, as illustrated in the following figure. In other words, the name of the array is a **pointer** to a piece of memory. The size of this piece of memory is the product of the number of elements and the size of each element (remember we mentioned different data types have different sizes?).

The pointer itself also occupies memory space. **When we pass an array to a function, we pass that pointer to the function and the function can modify the elements.** In C, a pointer is declared with an asterisk.

```
    int * va; /* a pointer to an integer or an array of integer */
```

C's arrays do not know their own sizes; therefore, we also have to pass the size into the function as the last parameter. Remember the index's range is between 0 and size - 1 (inclusive). **Giving an index outside the allowed range is a common mistake.** This mistake (in most cases) **cannot** be detected by a compiler and can cause a program to crash at run-time.

C does not automatically initialize variables. When you declare a variable or a pointer

```
int va;
int * pa;
```

you **must always** initialize their values before using them. **Uninitialized variables are common sources of errors.** Moreover, the program's behavior is **unpredictable**.

```
/* Do NOT do this */
int * pa;  /* not initialized */
if (pa == 0) /* result unpredictable */
{
   ...
}
```

because `pa` has not been initialized. The condition is meaningless.

5 Pointer and Address

A pointer does not have to be the beginning of an array. A pointer can point to a variable.

```
int val1 = 5;
int val2 = 10;
int * ptr;
printf("%d %d\n", val1, val2);
ptr = & val1;  /* ptr points to the address of val1 */
* ptr = 31;    /* change the value of the address pointed by ptr */
printf("%d %d\n", val1, val2);
ptr = & val2;
* ptr = 97;
printf("%d %d\n", val1, val2);
```

The output of this section of code is

```
5 10
31 10
31 97
```

In this example, two variables are assigned the values of 5 and 10. Then, `ptr` points to the address of `val1`. In C, & means the address of a variable. When we want to change the value of the address pointed by `ptr`, we need to add asterisk * ptr. Then `ptr` points to the address of `val2` and modifies the value. Can we write this?

```
ptr = 86;
```

The answer is both yes and no. You will receive a warning message but the program can execute. What does this line do? It assigns an address (value 86) to `ptr`. This is all right. However, if you want to change the value at this location

```
* ptr = 67;
```

The program will crash (for example, **"Segmentation fault"** because the program is not allowed to modify the value at this particular memory address). In general, a program does not control which memory address it can use. It has to obtain the address by using &.

The `printf` function displays the value of a variable. Is there a way to input the value from a keyboard? Yes, of course.

```
int v;
scanf("%d", & v);
```

When a program reaches this line, the program will stop and wait for the user to enter a value (and hit the Enter key). Then, `v` has the value entered by the user.

6 Reuse Function

We have created a function to add two arrays. We can reuse the function to add arrays of different sizes since the size is given into the function as an argument.

```
/* array3.c */
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
int addVector(int * va, int * vb, int * vsum, int size)
{
  int cnt;
  for (cnt = 0; cnt < size; cnt ++)
    {
      vsum[cnt] = va[cnt] + vb[cnt];
    }
  return 0;
}
int main(int argc, char * argv[])
{
  const int vecSize1 = 20;
  const int vecSize2 = 10;
  int veca[vecSize1];
  int vecb[vecSize1];
  int vecSum1[vecSize1];
  int vecc[vecSize2];
  int vecd[vecSize2];
  int vecSum2[vecSize2];
  int cnt;
  struct timeval currTime;
  gettimeofday(&currTime, NULL);
  srand(currTime.tv_usec);
  for (cnt = 0; cnt < vecSize1; cnt ++)
    {
      veca[cnt] = rand() % 1000;
      vecb[cnt] = rand() % 1000;
    }
  addVector(veca, vecb, vecSum1, vecSize1);
  /* call it first time */
  printf("vecSum1 :\n");
  for (cnt = 0; cnt < vecSize1; cnt ++)
    {
      printf("%3d + %3d = %4d\n",
```

```
            veca[cnt], vecb[cnt], vecSum1[cnt]);
    }

  for (cnt = 0; cnt < vecSize2; cnt ++)
    {
      vecc[cnt] = rand() % 1000;
      vecd[cnt] = rand() % 1000;
    }
  addVector(vecc, vecd, vecSum2, vecSize2);
  /* call it again */
  printf("vecSum2 :\n");
  for (cnt = 0; cnt < vecSize2; cnt ++)
    {
      printf("%3d + %3d = %4d\n",
            vecc[cnt], vecd[cnt], vecSum2[cnt]);

    }
  return 0;
}
```

In this example, `addVector` is called twice with different arrays of different sizes. Of course, we can create another function that initializes the values in the arrays:

```
/* array4.c */
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

void initVector(int *va, int size)
{
  int cnt;
  for (cnt = 0; cnt < size; cnt ++)
    {
      va[cnt] = rand() % 1000;
    }
}

int addVector(int * va, int * vb,
              int * vsum, int size)
{
  int cnt;
  for (cnt = 0; cnt < size; cnt ++)
    {
```

```
      vsum[cnt] = va[cnt] + vb[cnt];
    }
  return 0;
}

int main(int argc, char * argv[])
{
  const int vecSize1 = 20;
  const int vecSize2 = 10;
  int veca[vecSize1];
  int vecb[vecSize1];
  int vecSum1[vecSize1];
  int vecc[vecSize2];
  int vecd[vecSize2];
  int vecSum2[vecSize2];
  int cnt;
  struct timeval currTime;
  gettimeofday(&currTime, NULL);
  srand(currTime.tv_usec);
  initVector(veca, vecSize1);
  initVector(vecb, vecSize1);
  addVector(veca, vecb, vecSum1, vecSize1);
  printf("vecSum1 :\n");
  for (cnt = 0; cnt < vecSize1; cnt ++)
    {
      printf("%3d + %3d = %4d\n",
             veca[cnt], vecb[cnt], vecSum1[cnt]);
    }

  initVector(vecc, vecSize2);
  initVector(vecd, vecSize2);
  addVector(vecc, vecd, vecSum2, vecSize2);
  printf("vecSum2 :\n");
  for (cnt = 0; cnt < vecSize2; cnt ++)
    {
      printf("%3d + %3d = %4d\n",
             vecc[cnt], vecd[cnt], vecSum2[cnt]);
    }
  return 0;
}
```

We add a new function `initVector`. It does not return anything so the function's return type is `void`. You can probably guess that a function does not have to return `int`. A

function can also return `char`, `float`, or `double`.

Here is an important way to reduce the possibility of mistakes in a program. **A good programmer detects and removes similar code by creating functions.** If you find that you are writing code in several places and the code is identical or somewhat similar, you should create a function so that the code is in a single place. Why? Because you have to change only one place later, if any change is needed. **Assume that you will change the same programs many times — because you will.** Consider this example, if you want each element to be between -50 and 9590, you need to change only one place in `initVector`. This may not seem much improvement. However, as your program becomes larger and more complex, you will quickly lose track of how many places to change. **A program with several similar pieces of code is likely to have mistakes.** When you need to change the code, you will likely change some places but forget to change the other places. As a result, the program will behave in a strange way. In some test cases, the program is correct (executing the code you have recently changed). In some other cases, the program is wrong (executing the code you forgot to change). This type of mistakes is very hard to find and very time-consuming to correct. It is better to prevent the problems all together from the beginning. **Do not copy-paste code.** You should create a function instead.

Copy-paste code in multiple places is a common source for mistakes. When you want to change something, you have to change all places. If you forget to change all places, the program is incorrect. Copy-paste code seems to allow you to create a lot of code quickly, but you will spend many hours finding and fixing problems later.

In rare cases, you may copy-paste code for performance reasons (calling a function may slightly slow down a program). However, such cases are rare and you should not worry about it in ECE 264. You need to **write a correct program before making it fast.**