Lecture 05

Advanced pointers *, ** and ***

In this lecture

- 1D arrays revisited
- Array as a const pointer
- Dynamic arrays and resizing
- 2D arrays
- 2D Array Representation
- Arrays and pointers
- *, **, and ***
- Starting to think like a C programmer
- Further readings
- Exercises

1D Array Revisited

Unlike Java, C arrays are NOT objects. They do not have any inherited properties like length or do not contain methods like contains. C arrays are made up of primitives. A C array can be viewed as a contiguous memory block. The size of the block depends on the type of the array. For example, if we declare

```
char A[10];
```

an array of 10 characters would require 10 bytes of storage for data. On the other hand,

```
int A[10];
```

would require 10*sizeof(int) amount of storage.

In general, the sizeof(A) returns the number of bytes required to store the array.

Array as a const pointer

As stated above the name of the array is a const pointer to the first element of the array. Hence think of A as the address of A[0], A+1 as the address of A[1] etc. Assigning a value to a const pointer is illegal. For example,

```
int A[n];
int* ptr = A ;  /* is valid */
```

however

```
A = ptr;   /* is invalid as A cannot be changed */
```

Note that in the above example, the difference between A and ptr is that

ptr is a pointer to an integer or int*

and

A is a const pointer to an integer or const int*

Moreover, A is a static array managed by the compiler in the run time stack and ptr is just a pointer variables. Hence sizeof(A) returns total bytes required for A, while size(ptr) returns the number of bytes required for an address variable. We will discuss more about the similarities and differences between A and ptr later.

Dynamic Arrays and Resizing

Arrays by definition are static structures, meaning that size cannot be changed during run time. When an array is defined as

```
int A[n];
```

then A is considered a static array and memory is allocated from the run time stack for A. When A goes out of scope, the memory is deallocated and A no longer can be referenced.

C allows dynamic declaration of an array as follows.

```
int* A = (int*)malloc(sizeof(int)*n)
```

The above code declares a memory block of size n*sizeof(int) that can be accessed using the pointer A. For example, A can be initialized as follows:

```
int i;
for (i=0; i<n; i++)
    A[i] = 0;
```

Note that although A was declared as a pointer, A can be treated as an array. The difference between

int A[10] and int* A = malloc(10*sizeof(int))

is that latter is assigned memory in the dynamic heap (and hence must be managed by the programmer) and former is assigned memory from the run time stack (and hence managed by the compiler)

When defined dynamically as

```
int* A = (int*)malloc(sizeof(int)*n)
```

arrays can be resized easily. For example, if we want to double the size of the array, then we can do the following.

```
int* tmp = (int*)malloc(sizeof(int)*2*n)
int i;
for (i=0; i<n; i++)
  tmp[i] = A[i];

free(A);
A = tmp;
```

Note that the above code assigns a new block of memory, copy old content from A to tmp, free original memory allocated for A and reassign A pointer. Note that free(A) only frees the memory associated with A, but the variable A still exists.

2D arrays revisited

An array is a contiguous block of memory. A 2D array of size m by n is defined as

```
int A[m][n];
```

The number of bytes necessary to hold A is **m*n*sizeof(int).** The elements of A can be accessed using **A[i][j]** where i can be thought of as the row index and j can be thought of as the column index.

Now we take a look at how 2D arrays are store their elements. For example,

```
#define n 2
#define m 3
int A[n][m];
```

OR can be defined and initialized as

int A[2][3]={{1,2,3},{4,5,6}};

A[0][0]	A[0][1]	A[0][2]	A[1][0]	A[1][1]	A[1][2]

Here n represent the number of rows and m represents the number of columns. 2-D arrays are represented as a contiguous block of n blocks each with size m (i.e. can hold m integers(or any data type) in each block). The entries are stored in the memory as shown above. Type in the following program to see where the elements are stored.

```
Program_7_2:
#include <stdio.h>
#define n 2
#define m 3

int main(int argc, char* argv[]){
   int A[n][m]={{3,2,4},{7,1,9}},i=0,j=0;
   for (i=0;i<n;i++)
    for (j=0;j<m;j++)
     printf(“%x “,A[i]+j);
   printf(“\n”);
   for (i=0;i<n;i++)
    for (j=0;j<m;j++)
      printf("%d ",*(A[i]+j));
   printf("\n");

}
```

sample output

```
bfe3a1e0    bfe3a1e4    bfe3a1e8    bfe3a1ec    bfe3a1f0    bfe3a1f4
3               2           4           7           1           9
```

Another way to think of a 2D array is as follows. Suppose we define 2D array as

int A[][3] = {{1,2,3},{4,5,6}};

here we did not specify the number of rows, but by virtue of initialization on the right, A is assigned a block of 6 integers and the number of rows set to 2.

Arrays and Pointers

Arrays and pointers are closely related in C. In fact an array declared as

```
int A[10];
```

can be accessed using its pointer representation. The name of the array A is a constant pointer to the first element of the array. So A can be considered a **const int***.Since A is a constant pointer, A = NULL would be an illegal statement.

Other elements in the array can be accessed using their pointer representation as follows.

```
&A[0] = A
&A[1] = A + 1
&A[2] = A + 2
…..
&A[n-1] = A + n-1
```

If the address of the first element in the array of A (or &A[0]) is FFBBAA0B then the address of the next element A[1] is given by adding 4 bytes to A.
That is

&A[1] = A + 1 = FFBBAA0B + 4 = FFBBAA0F

And

&A[2] = A + 2 = FFBBAA0B + 8 = FFBBAA13

Note that the when doing address arithmetic, the number of bytes added depends on the ***type*** of the pointer. That is int* adds 4 bytes, char* adds 1 byte etc. You can type in this simple program to understand how a 1-D array is stored.

```
Program_5_1:
#include <stdio.h>
#define n 5

int main(int argc, char* argv[]){
   int A[n],i=0;
   for (i=0;i<n;i++)
     printf(“%x “,A+i);
   printf(“\n”);

}
```

sample
out put

```
bf802330   bf802334   bf802338   bf80233c   bf802340
```

Array of Pointers

C arrays can be of any type. We define array of ints, chars, doubles etc. We can also define an array of pointers as follows. Here is the code to define an array of n char pointers or an array of strings.

```
char* A[n];
```

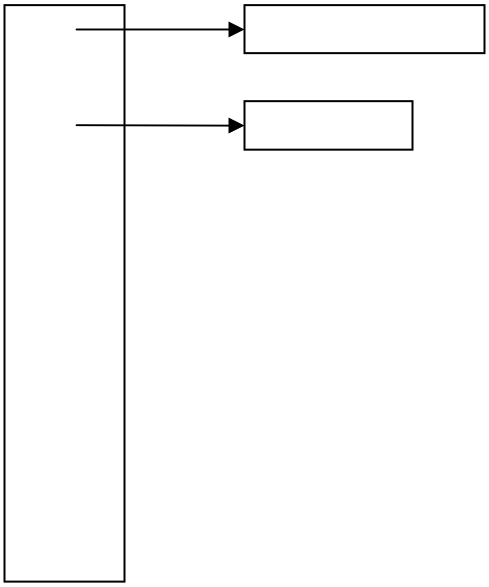
each cell in the array A[i] is a char* and so it can point to a character. You should initialize all the pointers (or char*) to NULL with

```
for (i=0; i<n; i++)
    A[i] = NULL;
```

Now if you would like to assign a string to each A[i] you can do something like this.

```
A[i] = malloc(length_of_string + 1);
```

Again this only allocates memory for a string and you still need to copy the characters into this string. So if you are building a dynamic dictionary (n words) you need to allocate memory for n char*'s and then allocate just the right amount of memory for each string.



Functions that take pointer arguments

Pointers or memory addresses can be passed to a function as arguments. This may allow indirect manipulation of a memory location. For example, if we want to write a swap function that will swap two values, then we can do the following.

```
void intswap(int* ptrA, int* ptrB){
    int temp = *ptrA;
    *ptrA = *ptrB;
    *ptrB = temp;
}
```

To use this function in the main, we can write the code as follows.

```
int A = 10, B = 56;
intswap(&A, &B);
```

note that the addresses of the variables A and B are passed into the intswap function and the function manipulates the data at those addresses directly. This is equivalent to passing values by "reference". It is really passing the

values that are addresses instead of copies of variables. However this can be dangerous since we are giving access to the original values.
One way to avoid this situation is to provide only "read" access to the data using a pointer. Consider the following function.

```
void foo(const int* ptr){

  /* do something */

}
```

The function takes the address of an integer variable, but is not allowed to change its content. It only has read privileges.

```
printf("%d", *ptr) is legal
scanf("%d", ptr) is illegal
```

Functions that Return pointers

Pointers can be returned from functions. For example, you can think of a function that allocates a block of memory and pass a pointer to that memory back to the main program.
Consider the following generic function that returns a block of memory.

```
void* allocate(short bytes){
    void* temp = malloc(bytes);
    return temp;
}
```

The function can be used in the main as follows.

```
int* A = (int*)allocate(sizeof(int)*100);
char* S = (char*)allocate(sizeof(char)*n+1);
```

since the function returns a void* it can be allocated for any pointer type, int*, double*, char* etc. However, you need to take great care in using the array. You must be aware of the segmentation of the array (4 byte blocks for int, 1 byte blocks for chars etc)

*, ** and ***

Starting to think like a C programmer

We have spent quite a bit of time now talking about C language. It is possible that so far your thinking was based on your first “computer” language Java. You may have been trying to think like a Java programmer and convert that thought to C. Now it is time to think like a C programmer. Being able to think directly in C will make you a better C programmer. Here are 15 things to remember when you start a C program from scratch.

1. include <stdio.h> in all your programs
2. Declare functions and variables before using them
3. increment and decrement with ++ and – operators.
4. Use x += 5 instead of x = x + 5
5. A string is an array of characters ending with a ‘\0”. Don’t ever forget the null character.
6. Array of size n has indices from 0 to n-1. Although C will allow you to access A[n] it is very dangerous.
7. A character can be represented by an integer (ASCII value) and can be used as such.
8. The unary operator & produces an address
9. The unary operator * dereference a pointer
10. Arguments to functions are always passed by value. But the argument can be an address of just a value
11. For efficiency, pointers can be passed to or return from a function.
12. Logical false is zero and anything else is true
13. You can do things like for(;;) or while(i++) for program efficiency and writability
14. Use /* .. */ instead of //
15. Always compile your program with –ansi –pedantic –Wall flags

Further Readings

1. K & R – 1.6, 1.7, 1.8, 1.9, 4.1, 5.1,5.3, 5.7
2. http://www.cs.cmu.edu/~thoffman/S09-15123/Chapter-2/Chapter-2.html#CHAP_2.2

Exercises

1. Write a function foo that takes a file name as a string, and reads each string in the file, allocate memory and create an array of strings (of multiple lengths) and return the address of the array back to the calling program. Assume the max size of the file is set in MAX_WORDS (#define MAX_WORDS = 100)

2. What could be a possible error in the following code?

```
int* foo(int n){
    int A[10], *x;
    strcpy(A,”guna”);
    x = A;
    return x;
 }
```

3. What can be wrong with the following code?

```
int A[10], i, *ptr;
for (i=0;i<10;i++)
    ptr = A + i;
printf(“%d “, *(ptr+1));
```

4. The C library string.h contains the function strcpy(dest,src) that copies src string to a dest string. Write a alternative version of the strcpy with the following prototype. The function returns 0 if successful and returns 1 if fails for some reason.

```
int mystrcpy(char* dest, const char* src){

}
```

   Is it possible to check inside the function, whether there is enough memory available in dest to copy src? Justify your answer.

5. What is the output of the following code?

```
void foo() {
  int x, *y;
  x=30;
  y=&x;
  y++;
  printf("x=%d y=%d\n", x, *y);
}
```

6. What is the output of the following code. If there is an error state the type of the error

```
void foo() {
  char *string, *x;
  string = (char *)malloc(20);
  strcpy(string, "Hello World.");
  x=string;
  for( ; *x != '\0'; x++) {
    printf("%c", *x);
  }
  printf("\n");
}

int main( int argc, char *argv[]) {
  foo();
  return 0;
}
```

ANSWERS

1. Write a function foo that takes a file name as a string, and reads each string in the file, allocate memory and create an array of strings (of multiple lengths) and return the address of the array back to the calling program. Assume the max size of the file to be MAX_WORDS

Answer:

```
char** foo(char* filename){
        char tmp[100];
        char* list[MAX_WORDS];
        int i = 0;
       FILE* fp = fopen(filename,”r”);
        while (fscanf(fp,”%s”,tmp)>0){
           list[i] = malloc(strlen(tmp)+1);
           strcpy(list[i++],tmp);
        }
        return list;
 }
```

2. What could be a possible error in the following code?

```
int* foo(int n){
    int A[10], *x;
    Strcpy(A,”guna”);
    x = A;
    return x;
 }
```

ANSWER: A is a local allocation of memory, and when x is returned A no longer exists. Therefore, any effort to dereference the address returned by x could cause errors.

3. What can be wrong with the following code?

```
int A[10], i, *ptr;
for (i=0;i<10;i++)
    ptr = A + i;
printf(“%d “, *(ptr+1));
```

Answer: After loop is executed the ptr points to the last thing in the array (A[9]). Now *(ptr+1) tried to dereference the content at A[10], something that does not exists.

4. The C library string.h contains the function strcpy(dest,src) that copies src string to a dest string. Write an alternative version of the strcpy with the following prototype. The function returns 0 if successful and returns 1 if fails for some reason.

```
int mystrcpy(char* dest, const char* src){
     int i = 0;
     while (i<strlen(src) && src[i] != ‘\0’) {
         dest[i] = src[i++];
     dest[i] = ‘\0’;
     return 0;
}
```

Is it possible to check inside the function, whether there is enough memory available in dest to copy src? Justify your answer.

Answer: The passed argument dest is a copy of the address of the memory available to dest. However, the mystrcpy does not know anything about the max size available to copy src. So obviously the code above is dangerous since we could overwrite some memory not allocated to us.