Lecture 04
Introduction to pointers

A pointer is an address in the memory. One of the unique advantages of using C is that it provides direct access to a memory location through its address. A variable declared as int x has the address given by &x. ***& is a unary operator*** that allows the programmer to access the address of a single variable declared. The following simple program shows you how to find the address of an allocated variable.

```
#include <stdio.h>
int main(int argc, char* argv[]){
    int x = 10;
    printf(“The address of %d is %x \n”, x, &x);
    return 0;
}
```

For example if a variable x has the address bf9b4bf4 that means the first byte of the 4 bytes allocated for the variable x is bf9b4bf4.

Quiz: What are the addresses of the other 3 bytes?

An array variable defined as int A[3]; has an address of the first location given by

```
printf(“The address of A is %x \n”, A);
```

Note that & is not used to find the address of the array. That’s because the name of the array is in fact a pointer to the array (or the address of the first element A[0] of the array). The address of next element of array A is A+1. However, the meaning of A+1 depends on the type of A. For example, if A is an array of chars, then A+1 can be obtained by adding 1 to the address of A. However, if A is an array of ints, then A+1 must be obtained by adding 4 bytes to A as the array holds the type int.

Hence we have the following understanding of arrays.
A = &A[0], A+1 = &A[1], … etc..

Dereferencing Pointers

Given the address of a variable, the address can be dereferenced to find the actual content of that location. For example, consider the following code

```
int x = 10;
printf(“%d \n”, *(&x));
```

The output of the code is equivalent to printing the value of x. Compiler will interpret *(&x) as the content stored at 4 bytes starting at address of x. (why 4 bytes?)

Pointer Variables

We now know how to define standard variables of types char, int, double etc. C also allow users to define variables of type pointer(or address). A pointer or address variable to an int is defined as:

```
int* ptr;
```

The * can be placed anywhere between int and ptr. Compiler will consider ptr to be an address of a variable of int type. Therefore any dereferencing of the ptr variable will cause the program to look for 4 bytes of memory. Similarly we can define double*, char*, long*, void* etc. Memory can be viewed as an array of bytes and once a variable is declared, the value of the variable are stored in the memory as follows.

```
int x = 0x2FFF
```

Suppose following is an array of bytes. Then x is stored (little endian) as

		0 0	0 0	2 f	f f				
ff01	ff02	ff03	ff04	ff05	ff06	ff07	ff08	ff09	ff0A

Note that array addresses are given above. We use simple numbers to denote addresses although addresses are 32-bits or 64-bits

Now we can assign the address of a variable declared to a pointer as follows.

```
int x = 0x2fff;
int* xptr = &x;
```

Dereferencing xptr will now give access to the value of the variable. Dereferencing is done using the unary operator *. For example the following line of code will print x using its direct reference and its pointer reference.

```
printf("The value of x is %d or %d\n", x, *xptr);
```

Allocating Dynamic Memory

Java allocates memory with its "new" statement. But C does not have a "new" statement (C++ does) to allocate memory. One of the ways C allocates dynamic memory is with the malloc function. The malloc function prototype

```
void* malloc(size_t size)
```

Allows programmer to specify how much memory is needed for the variable or data structure at run time. For example,

malloc(4) returns a pointer (or address) to a 32-bit block of memory. So if we would like to allocate memory for 100 integers, we can write

```
int* A = (int*)malloc(100*sizeof(int));
```

Note that malloc returns void* and it is type casted to int* to be type compatible. You can now consider A as a **dynamic array** of ints and use A just like an array. For example,

```
for (i=0; i<100; i++)
    A[i] = random();
```

will assign 100 random numbers to the array.

Equivalently we can write

```
for (i=0; i<100; i++)
    *(A+i) = random();
```

Here the A+I refers to the address of A[i] and dereferencing &A[i] gives the content of A[i]

Reading Strings

A C string is an array of characters ending with a NULL character ‘\0’. You can allocate static memory for a string with

```
char s[50];
```

this will allocate 50 bytes for the string and the maximum size of the string that can be stored in this array is 49. However, this is not efficient when string size is a variable (like in a dictionary). Therefore we can allocate memory using malloc. So if you know the size of a string, say n bytes, you can allocate memory using,

```
char* s = (char*)malloc(n+1);
```

You still need to copy the characters into S. So suppose we read a string into a temp array as follows.

```
char tmp[50];
fscanf(stdin, “%s”, tmp);
```

and suppose we need to copy the string to a permanent location. So we do the following.

```
char* name = malloc(strlen(tmp)+1);
strcpy(name, tmp);
```

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.

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

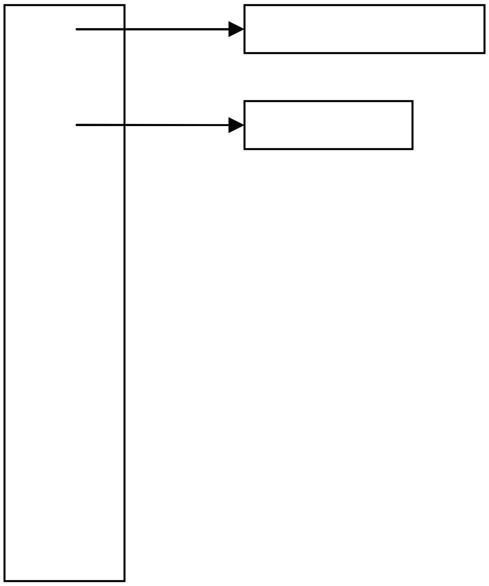
each cell in the array A[i] is a char* and so it can point to a character. You should initialize all the pointers (or char*) to NULL with

```
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 pointers

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)

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 flag

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 to be MAX_WORDS

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.

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));
```

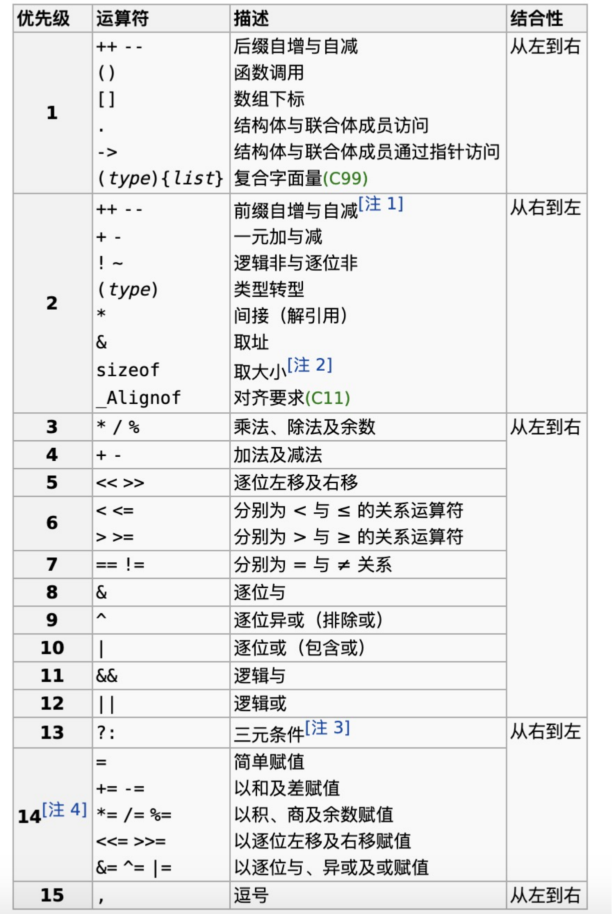
Answer: After loop is executed the ptr points to the last thing in the array (A[9]). Now *(ptr+1) tried to dereference the content at A[10], something that does not exists.

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.



优先级	运算符	描述	结合性
1	++ -- () [] . -> (*type*){*list*}	后缀自增与自减 函数调用 数组下标 结构体与联合体成员访问 结构体与联合体成员通过指针访问 复合字面量(C99)	从左到右
2	++ -- + - ! ~ (*type*) * & sizeof _Alignof	前缀自增与自减[注 1] 一元加与减 逻辑非与逐位非 类型转型 间接（解引用） 取址 取大小[注 2] 对齐要求(C11)	从右到左
3	* / %	乘法、除法及余数	从左到右
4	+ -	加法及减法	
5	<< >>	逐位左移及右移	
6	< <= > >=	分别为 < 与 ≤ 的关系运算符 分别为 > 与 ≥ 的关系运算符	
7	== !=	分别为 = 与 ≠ 关系	
8	&	逐位与	
9	^	逐位异或（排除或）	
10	|	逐位或（包含或）	
11	&&	逻辑与	
12	||	逻辑或	
13	?:	三元条件[注 3]	从右到左
14[注 4]	= += -= *= /= %= <<= >>= &= ^= |=	简单赋值 以和及差赋值 以积、商及余数赋值 以逐位左移及右移赋值 以逐位与、异或及或赋值	
15	,	逗号	从左到右