# Lecture 05
# Pointers ctd..

***Note: some notes here are the same as ones in lecture 04***

## 1 Introduction

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..

## 2 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?)

## 3 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);
```

## 4 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]

## 5 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);
```

## 6 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];
```

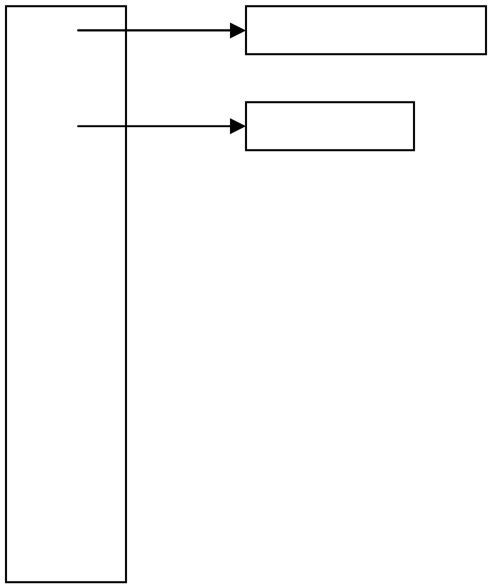
each cell in the array A[i] is a char* and so it can point to a character. You should initialize all the pointers (or char*) to NULL with

```
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.



## 7 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
```

## 8 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)

## 9 Introduction to Strings

Learning how to manipulate strings is quite important in any programming language. In Java string is an object and inherits all its object properties. However, in C string is an object with no inherited properties (such as length). First we will begin with the concept of a pointer or address. We will discuss in detail what pointers mean shortly, but for now we want to start with a definition as follows.

```
char* s;
```

The above statement simply indicates that s is a pointer to (or address of) a character. A String is simply defined as an array of characters and s is the address of the first character (byte) of the string. In C, a string is just an array of characters that does not have any inherited properties. **A valid C string ends with the null character '\0' [slash zero].** Therefore the amount of memory required for a C string is 1 + length of the actual string. Failure to make sure that a string ends with '\0' may result in unpredictable behavior in your code. Please note that some IO library functions automatically add a null character to the end of each string.

## 10 Initializing a String

A constant string s can be simply initialized as

```
char* s = “guna\0”;
```

However no memory is allocated for s in the stack. If we try to write to s, then it may cause a segmentation fault since memory has not been allocated explicitly.
For example,

**fscanf(stdin,”%s”,s);** would cause a problem

If we need to read into a memory location, it is important to allocate memory first and then copy the string to the location. To allocate a block of memory to hold a string, we use the **malloc** function from <stdlib.h>. To read more about malloc type:

```
% man malloc
```

The **malloc(int n)** returns a pointer to (or an address of) a block of n bytes. Note that a string with n characters requires n+1 bytes (n for the string AND 1 byte to store the ‘\0’ character). Therefore, to store the input string “guna”, we would require 5 characters. The following code allocates 5 characters to store the string “guna” + ‘\0’.

```
char *S = malloc(5*sizeof(char));
strcpy(S,”guna”);
```

It is important to note that malloc allocates memory inside what is called the **“dynamic heap”** and unless memory is explicitly freed using **free** function (we will discuss this later. A very important topic), the malloced block stays even after leaving the scope of the code.

Alternatively we can also write

```
char s[5];
strcpy(s,”guna”);
```

In this case, 5 bytes is allocated from the run time stack and s no longer available once it is out of scope of the variable s.

## 11 Reading a String from a file Stream

We can create a file stream using an input file as follows:

```
FILE* fp = fopen(“myfile.txt”,”r”);
```

The file is now open for “r” only and fp (FILE* or FILE pointer) can be used to read input from the file. To read from a file we can use fscanf. You can find more about fscanf by typing man fscanf at the unix prompt:

```
% man fscanf
```

```
NAME
       scanf,  fscanf, sscanf, vscanf, vsscanf, vfscanf - input format con-
       version

SYNOPSIS
       #include <stdio.h>
       int scanf(const char *format, ...);
       int fscanf(FILE *stream, const char *format, ...);
       int sscanf(const char *str, const char *format, ...);

       #include <stdarg.h>
       int vscanf(const char *format, va_list ap);
       int vsscanf(const char *str, const char *format, va_list ap);
       int vfscanf(FILE *stream, const char *format, va_list ap);

DESCRIPTION
       The scanf family of functions scans input according to a  format  as
       described below.  This format may contain conversion specifiers; the
       results from such  conversions,  if  any,  are  stored  through  the
       pointer arguments.  The scanf function reads input from the standard
       input stream stdin, fscanf  reads  input  from  the  stream  pointer
       stream, and sscanf reads its input from the character string pointed
       to by str.
```

As an example, to read data from stdin,

**char s[10];**
**fscanf(stdin,"%s",s);**

Reading a string using fscanf is somewhat dangerous. It is possible that the input you enter may be longer than the memory allocated by the character array. For example, if you type something more than 9 characters in the above example, the program may segfault as enough memory have not been allocated for the string s. Consider program 3.1 below. Type the program and see what happens if you enter something significantly longer than 10. The behavior of the program is completely unpredictable.

```
/* Program 3.1 */
#include <stdio.h>
int main(int argc, char* argv[]){
  char S[10];
  fscanf(stdin,"%s",S);
  printf("The input is %s \n", S);
  return (EXIT_SUCCESS);
}
```

You need to be careful about managing memory for strings. This is especially true if you are reading strings of variable length and the size of the memory cannot be fixed in advance. One possible way to safely read strings is to use fgets function.

**char *fgets(char *s, int size, FILE *stream);**

*fgets() reads in at most one less than size characters from stream and stores them into the buffer pointed to by s. Reading stops after an EOF or a newline. If a newline is read, it is stored into the buffer. A '\0' is stored after the last character in the buffer.*

There is also another version, gets as follows

**char *gets(char *s);**

**However, DO NOT use gets since we do not know how many characters will be read from the stdin.**

***Warning:*** *Never use gets().Because it is impossible to tell without knowing the data in advance how many characters gets() will read, and because gets() will continue to store characters past the end of the buffer, it is extremely dangerous to use. It has been used to break computer security. Use fgets() instead. [**Source: UNIX Manual**]*

Before using fgets we need to make sure a buffer has been allocated to read in the string. For example

```
char buffer[50];
fgets(buffer, 40, stdin);
```

will read 39 characters into the buffer
(max buffer size 50).

## 12 Writing a String

If a string is properly read into a character array called buffer, then we can write the string to an output stream using fprintf as follows. (find out more about fprintf using **man fprintf** )

```
fprintf(stdout,"%s",buffer);
```

or use formatting such as

```
fprintf(stdout,"%20s",buffer); /*uses 20 spaces for string*/
```

Another useful function for string output is **sprintf**. This is particularly useful if you need to construct a string out of fixed and variable lengths, integers, floating points numbers etc. For example you can think of a CMU student course record in the format

```
S07,gunadean,Guna,Dean,SCS,CS,2,L,4,15111 ,1 ,,
```

Given the values of individual fields this can be created using sprintf. The prototype for sprintf is

```
int sprintf(char *str, const char *format, ...);
```

where 3 dots as the last argument indicates a variable length argument list(we will learn how to write such functions later in the course). An example of how to use sprintf is given below in program 3.2.

```
/* Program 3.2 */
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[]){
   char buffer[256];
   sprintf(buffer,"%s/%s%d.c",
           "/afs/andrew.cmu.edu", "myfile", 34);
   printf("%s\n",buffer);
   return (EXIT_SUCCESS);
}
```

The output produced by the above code is a string:

**"/afs/andrew.cmu.edu/myfile34.c"**

Some other library functions that can be used for string output are:

```
int puts(char* s);
int fputs(char *s, FILE* outfile);
```

## 13 Passing a String(s) to a function

A string or an array of strings can be passed as an argument to a function. Suppose for example we have a character buffer we need to pass to a function.

```
char buffer[50];
```

**Also assume that we have a function foo as follows:**
```
void foo(char A[] , int length){…}
```

**We can then pass buffer into foo as follows:**
```
foo(buffer,n); /* n is the length of the buffer */
```

Passing the address of the buffer allows direct manipulation of the content at the buffer location

Suppose that foo's job is to read a string of length n into buffer from stdin. Then it can be defined as

```
void foo(char[] s, int n){
   fgets(s,n,stdin);
}
```

## 14 Returning a String from a function

A string or the address of a string can be returned from a function. Consider the following foo function.

```
char* foo(){
   char* s;
   s = malloc(n);
   return s;
}
```

The function allocates memory for a string, and returns its address back to main.

## 15 Swapping Two Strings

Suppose we need to swap two strings. A simple swapstring function can be written as follows.

```
void swapstrings(char A[], char B[]){
    char tmp[10];
    strcpy(tmp,A);
    strcpy(A,B);
    strcpy(B,tmp);
}
```

Note that it is important to do **string copy,** instead of just assigning addresses of strings that would only result in a local change. Here is a wrongswapstrings.

```
void wrongswapstrings(char A[], char B[]){
    char tmp[10];
    tmp = A;
    A = B;
    B = tmp;
}
```

## Caution

## scanf("%s",buffer) and gets()

These functions are vulnerable to buffer overflows and may pose some security problems. Never use them in your programs. You can use fgets instead.

Unlike java C does not come with a extensive API for string operations. However, C library, <string.h> provides a decent collection of functions that can be used to accomplish any task. You can find more about <string.h> by typing:

## % man string.h

```
NAME
       string.h - string operations

SYNOPSIS
       #include <string.h>

DESCRIPTION
       Some  of  the functionality described on this reference page extends
       the ISO C standard. Applications shall define the  appropriate  fea-
       ture   test   macro   (see   the   System   Interfaces   volume   of
       IEEE Std 1003.1-2001, Section 2.2, The Compilation  Environment)  to
       enable the visibility of these symbols in this header.
```

## 16 String.h methods

C strings are supported by the string.h library. The following functions are available.

| **Copying** | |
|---|---|
| void *memcpy(void *dest, const void *src, size_t n); | The memcpy() function copies n bytes from memory area src to memory area dest. The memory areas may not overlap. |
| void *memmove(void *dest, const void * src, size_t n); | The memmove() function copies n bytes from memory area src to memory area dest. The memory areas may overlap. |
| char *strcpy(char *dest, const char *src); | The strcpy() function copies the string pointed to by src (including the terminating '\0' character) to the array pointed to by dest. The strings may not overlap, and the destination string dest must be large enough to receive the copy. |
| char *strncpy(char *dest, const char *src, size_t n); | The strncpy() function is similar, except that not more than n bytes of src are copied. Thus, if there is no null byte among the first n bytes of src, the result will not be null-terminated. |
| **Concatenation** | |
| char *strcat(char *dest, const char *src); | The strcat() function appends the src string to the dest string over-writing the '\0' character at the end of dest, and then adds a terminating '\0' character. The strings may not overlap, and the dest string must have enough space for the result. |
| char *strncat(char *dest, const char *src, size_t n); | The strncat() function is similar, except that only the first n characters of src are appended to dest. |
| **Comparison** | |
| int memcmp(const void *s1, const void *s2, size_t n); | The memcmp() function compares the first n bytes of the memory areas s1 and s2. It returns an integer less than, equal to, or greater than zero if s1 is found, respectively, to be less than, to match, or be greater than s2. |
| int strcmp(const char *s1, const char *s2); | The strcmp() function compares the two strings s1 and s2. It returns an integer less than, equal to, or greater than zero if s1 is found, respectively, to be less than, to match, or be greater than s2. |
| int strncmp(const char *s1, const char *s2, size_t n); | The strncmp() function is similar, except it only compares the first (at most) n characters of s1 and s2. |

| **Searching** | |
|---|---|
| char *strchr(const char *s, int c);<br><br>char *strrchr(const char *s, int c); | The strchr() function returns a pointer to the first occurrence of the character c in the string s.<br><br>The strrchr() function returns a pointer to the last occurrence of the character c in the string s. |
| char *strtok(char *s, const char *delim); | The strtok() function can be used to parse the string s into tokens. The first call to strtok() should have s as its first argument. Subsequent calls should have the first argument set to NULL. Each call returns a pointer to the next token, or NULL when no more tokens are found. |
| char *strstr(const char *haystack, const char *needle); | The strstr() function finds the first occurrence of the substring needle in the string haystack. The terminating '\0' characters are not compared. |
| **Other** | |
| void *memset(void *s, int c, size_t n); | The memset() function fills the first n bytes of the memory area pointed to by s with the constant byte c. |
| size_t strlen(const char *s); | The strlen() function calculates the length of the string s, not including the terminating '\0' character. |
| void *memchr(const void *s, int c, size_t n); | The memchr() function scans the first n bytes of the memory area pointed to by s for the character c. The first byte to match c (interpreted as an unsigned character) stops the operation. |

**Source: man unix**

## 17 Tokenizing a String

String tokenization is a very useful operation in many applications. For example, we may have to extract data from a comma separated file (CSV). In this case we need to extract tokens separated by comma delimiter. A String can be tokenized according to a delimiter. For example, let us assume we need to tokenize string S and the delimiter is the blank. The following code can be used to tokenize the string into tokens.

```
char* tk = strtok(S,” “);
do {
      printf(“%s\n”, tk);
} while ((tk=strtok(NULL,” “)) != NULL);
```

Note that the first call to the strtok uses the original string S and the subsequent calls we are passing NULL. When there are no more tokens. Strtok returns NULL and the while loop ends.

## Exercises

3.1. A typical entry on CMU student records looks like this

**S07,gunadean,Guna,Dean,SCS,CS,2,L,4,15111 ,1 ,,**

The course data file is a file where each line contains the following information: semester, computer id, student's last name, student's first name, college, department, class, grade option, qpa scale, course and section. Write a function that takes a typical entry as a string and break down its fields into appropriate data types. Output a file containing just the Andrew ID's

3.2 What is wrong with the following code?

```
{char *s1 = "Hello, "; char *s2 = "world!"; char *s3 = strcat(s1, s2);}
```

3.3 What happens with this code? Please explain

```
char* s = "guna\0";
char buffer[20];
printf("%s is of length %d\n",s, strlen(s));
strcpy(s, buffer); // strcpy(dest,source)
```

3.4 Why is this function is bad? Find all problems you can think of.

```
char* badfunction(int n){
  char A[n];
  strncpy(A,"ghfhhfhhhfhhfdfasfff",n);
  return A;
}
```

3.5 Explain why calling this function would not swap two integers A and B.

```
 void intswap(int x, int y){
    int temp = x;
    x = y;
    y = temp;
 }

int main(){
 int A=10, B=20;
 intswap(A,B);
}
```

3.6 Explain why calling this function would not swap two strings A and B.

```
  void stringswap(char x[], char y[]){
     char temp[10] = x;
     x = y;
     y = temp;
  }

int main(){
  char A[10]=”guna\0”, B[10]=”me\0”;
  stringswap(A,B);
}
```

3.7 What is the output of the following. If there are errors please state the error type.

```
void Question9() {
char *string, *x;
string = (char *)malloc(20*sizeof(char));
strcpy(string, "Hello World.");
x=string;
for( ; *x != '\0'; x++) {
  printf("%c", *x);
}
printf("\n");
}
```