Unit 5

5.1 An Introduction to Pointers

Consider the following declaration:

```
int number;                    // step 1
```

Here we declare that we will have an integer valued variable named “number”. On the computer each variable is stored in some memory location. You can visualize the memory layout of a computer as a consecutive set of “cells”, each of which has an “address”. A cell is typically a byte. Say your computer has 64 KB of memory. What this means is that it has 64 X 1024 = 65536 bytes available. (1 KB is 1024 bytes.) On computer each memory location will have an address. For instance, the first location could have the address of “0” and the last location could have the address of “65535”. Size of an integer variable depends on the architecture of a computer, but 4 bytes is typical for an integer value. Let’s say your computer represents an integer in a 4 -byte memory location. The declaration above can be visualized as in Figure 1 below.

Address	Address + 1	Address + 2	Address + 3

Figure 1. An integer is represented as a consecutive set of 4 bytes.

A pointer is a variable that contains the address of another similar type variable.

With the definition of a pointer on mind, consider the following declaration:

```
int* pointerToNumber;          // step 2
```

What this declares is a variable that holds the address of an integer variable, in other words, it is *a pointer to an integer* . To link the pointer variable to the variable whose address it will hold, the following initialization is necessary:

```
pointerToNumber = &number;     // step 3
```

The “&” sign in front of a variable defines the “address” of that variable. Note that in steps 1, 2, and 3 we defined an integer variable, a pointer to an integer variable, and assigned the pointer to the address of the integer variable. This is a typical set of steps. See Figure 2 below for the relation between the two visually.

Now, let’s say you want to use the value represented by integer variable “number” in an arithmetic expression. There is a direct way by using the variable itself –as we are accustomed to- and an indirect way by using the pointer to the variable. Let’s say you want to calculate the divide the “number” by 10:
Direct way:

```
k = number / 10;
```

Indirect way:

```
k = *pointerToNumber / 10;
```

The notation *"*pointerToNumber"* means the value held in the address pointed by *"pointerToNumber"* variable. The notation *"*pointerToNumber"* (star sign in front of a pointer) is also called "dereferencing a pointer"; it yields the value held in the pointed address.

You might be asking yourself why we need pointers. Being able to have access to the address of any given variable (and later you will see access to the functions) is very powerful: primarily, this will allow us to change values of function arguments *within functions* and handle dynamical memory management. Remember the matrix multiplication example from the previous lecture or the homework you worked on to calculate the average and standard deviation of a given set of numbers. In these exercises we had to keep the size of arrays constant. In other words, we had to specify the number of array elements; we could not determine the number of elements at *run-time dynamically*. Now that we have introduced pointers, we can dynamically allocate memory for any number of elements. First, we need to learn the relationship between arrays and pointers.

Arrays and Pointers

In C there is a strong relationship between arrays and pointers.
Remember that an array is a consecutive set of elements in memory. For instance,
int a[10];
declares 10 integer elements in memory. Consider an integer pointer "*p*" defined as
int p;*

Pointer p can be used to access elements of array *a* if one initializes *p* as follows:

p = &a[0];

Note that what we do here is to set *p* to point to the address of the first element of array *a*. See Figure 3 below. (An alternative way to set *p* to the first element of array *a* is to just state "*p=a*"; in other words, array variables themselves are pointers in C.)

Now, consider the following:

*i = *p;*

This is equivalent to

i = a[0];
This is because *p* points to *a[0]* and **p* refers to the value held in *a[0]*.

Another example:

Say you set p as follows:

p = &a[5];
and

*i = *p;*

What *i* is set to now is the 5th element of array *a*.

Pointer arithmetic

If you use a pointer to access the elements of an array, there is a powerful way of traversing the array's elements.

Consider

int p; int a[10];*

p = &a[0];

which set pointer *p* to the first element (indexed by 0) of array *a*. Now, say you want to refer to element "i" of array a. You can do this directly as
a[i];
or indirectly by using the pointer p as
**(p + i);*

Note that in the indirect way *(p+i)* takes you to the address of element *i* and *"*(p+i)"* gives you the value stored in address *(p+i).*

Important note: Adding "*i*" to a pointer does not mean adding "*i*" bytes to the address held by the pointer. If the size of the variable maintained in the address is "*k*", what this means is to add *i * k* to the address maintained by the pointer. For instance, if you are pointing to an integer array and your machine represents integers in 4 bytes, **(p + 3)* means the 3rd element of the array and the address is calculated internally as

*address of element 0 + 3 * 4*

In other words, the expression *(p+3) will take you to element 3 by jumping 12 bytes from the beginning address of the array. We will see more examples of this later when we cover structures.

5.2 Application of Pointer

Pointer is used for different purpose. Some of the pointer application is listed below:
1. Call by Reference
2. Accessing Array element
3. Dynamic Memory allocation
4. Some other pointer application in different data structure, such as tree, such as linked list, tree, graph etc.

5.2.1 Call by Reference (Pointer application)

```
#include<stdio.h>
#include<conio.h>
```

```
void main()
{
int a,b;
clrscr();
printf("-------Call By Reference---------");
printf("\nEnter the value of a and b");
scanf("%d%d",&a,&b);

printf("\nBefore swapping the values are\n");
printf("a=%d\nb=%d",a,b);

swap(&a,&b);

printf("\nAfter swapping the values are\n");
printf("a=%d\nb=%d",a,b);
getch();
}
swap(int  *a,int  *b)
{
int c;
c=*a;
*a=*b;
*b=c;
printf("\nIn function the values are\n");
printf("a=%d\nb=%d",*a,*b);
}
```

5.2.2 Accessing Array element (Pointer application)

```
#include<stdio.h>
void sendarr(int [],int n);
void main()
{
   int a[5]={4,5,3,2,6};
   int i,n=5;

  sendarr(a,n);

}
void sendarr(int a[],int n)
{
  int i;
   for(i=0;i<5;i++)
   printf("%d\t",*(a+i));
   }
```

5.2.3 Dynamic Memory allocation (Pointer application)

```
#include<conio.h>
```

```
#include<alloc.h>
void main()
{
int *aptr,n,i,m;
clrscr();
printf("how many number you want=") ;
scanf("%d",&n) ;

aptr=(int *) malloc(n*sizeof(int)) ;
if(aptr==NULL)
{
printf("memory not avialable\n");
}

printf("enter the element");
for(i=0;i<=n-1;i++)
{

scanf("%d",(aptr+i));
}

printf("\n Array elements are below\n");
for(i=0;i<=n-1;i++)
{
printf("%3d",*(aptr+i));
}
printf("\n how many more element  you wat to store =");
scanf("%d",&m);

aptr=(int *)realloc(aptr,m*sizeof(int));
if(aptr==NULL)
{
printf("memory not avialable\n");
exit(1);
}
printf("printf enter %d more elements\n",m) ;
for(i=n;i<=n+m-1;i++)
scanf("%d",(aptr+i));

printf("\n Elements are below\n\n");
for(i=0;i<=n+m-1;i++)
{
printf("%3d",*(aptr+i));
}
getch();
}
```

5.3 File Handling

In this topic you will come to know, how C programmers can create, open, close text or binary files for their data storage.
A file represents a sequence of bytes, does not matter if it is a text file or binary file. C programming language provides access on high level functions as well as low level (OS level) calls to handle file on your storage devices. This chapter will take you through important calls for the file management.

Opening Files

You can use the **fopen( )** function to create a new file or to open an existing file, this call will initialize an object of the type **FILE**, which contains all the information necessary to control the stream. Following is the prototype of this function call:

FILE *fopen(const char * filename, const char * mode);
Here, **filename** is string literal, which you will use to name your file and access **mode** can have one of the following values:

Mode Access mode	Description
“r”	Open an existing file for reading only.
“w”	Open a file for writing only. If the file does not exist create a new one. If the file exists it will be overwritten.
“a”	Open a file for appending only. If the file does not exist create a new one. New data will be added to the end of the file.
“r+”	Open an existing file for reading and writing
“w+”	Open a new file for reading and writing
“a+”	Open a file for reading and appending. If the file does not exist create a new

If you are going to handle binary files then you will use below mentioned access modes instead of the above mentioned:
"rb", "wb", "ab", "rb+", "r+b", "wb+", "w+b", "ab+", "a+b"

Closing a File

To close a file, use the fclose( ) function. The prototype of this function is:
int fclose(FILE *fp);
The **fclose( )** function returns zero on success, or **EOF** if there is an error in closing the file. This function actually, flushes any data still pending in the buffer to the file, closes the file, and releases any memory used for the file. The EOF is a constant defined in the header file **stdio.h**.
There are various functions provide by C standard library to read and write a file character by character or in the form of a fixed length string. Let us see few of the in the next section.
Writing a File
Following is the simplest function to write individual characters to a stream:
int fputc(int c, FILE *fp);
The function **fputc()** writes the character value of the argument c to the output stream referenced by fp. It returns the written character written on success otherwise **EOF** if there is an error. You can use the following functions to write a null-terminated string to a stream:
int fputs(const char *s, FILE *fp);
The function **fputs()** writes the string **s** to the output stream referenced by fp. It returns a non-

negative value on success, otherwise **EOF** is returned in case of any error. You can use **int fprintf(FILE *fp,const char *format, ...)** function as well to write a string into a file. Try the following example:

Make sure you have **/tmp** directory available, if its not then before proceeding, you must create this directory on your machine.

```
#include <stdio.h>
main()
{
  FILE *fp;

  fp = fopen("/tmp/test.txt", "w+");
  fprintf(fp, "This is testing for fprintf...\n");
  fputs("This is testing for fputs...\n", fp);
  fclose(fp);
}
```

When the above code is compiled and executed, it creates a new file **test.txt** in /tmp directory and writes two lines using two different functions. Let us read this file in next section.

Reading a File

Following is the simplest function to read a single character from a file:

```
int fgetc( FILE * fp );
```

The **fgetc()** function reads a character from the input file referenced by fp. The return value is the character read, or in case of any error it returns **EOF**. The following functions allow you to read a string from a stream:

```
char *fgets( char *buf, int n, FILE *fp );
```

The functions **fgets()** reads up to n - 1 characters from the input stream referenced by fp. It copies the read string into the buffer **buf**, appending a **null** character to terminate the string.

If this function encounters a newline character '\n' or the end of the file EOF before they have read the maximum number of characters, then it returns only the characters read up to that point including new line character. You can also use **int fscanf(FILE *fp, const char *format, ...)** function to read strings from a file but it stops reading after the first space character encounters.

```
#include <stdio.h>

main()
{
  FILE *fp;
  char buff[255];

  fp = fopen("/tmp/test.txt", "r");
  fscanf(fp, "%s", buff);
  printf("1 : %s\n", buff );

  fgets(buff, 255, (FILE*)fp);
  printf("2: %s\n", buff );

  fgets(buff, 255, (FILE*)fp);
  printf("3: %s\n", buff );
```

```
   fclose(fp);

}
```

When the above code is compiled and executed, it reads the file created in previous section and produces the following result:

```
1 : This
2: is testing for fprintf...
3: This is testing for fputs...
```

Let's see a little more detail about what happened here. First **fscanf()** method read just **This** because after that it encountered a space, second call is for **fgets()** which read the remaining line till it encountered end of line. Finally last call **fgets()** read second line completely.
Binary I/O Functions
There are following two functions, which can be used for binary input and output:
size_t fread(void *ptr, size_t size_of_elements, size_t number_of_elements, FILE *a_file);
size_t fwrite(const void *ptr, size_t size_of_elements, size_t number_of_elements, FILE *a_file);
Both of these functions should be used to read or write blocks of memories - usually arrays or structures.

C program to write a File

```
#include<stdio.h>
#include<conio.h>
void main()
{
FILE *fp;
char fname[20];
int ch;
clrscr();

printf("Enter File name:");
scanf("%s",fname);

fp=fopen(fname,"w");

if(fp==NULL)
{
printf("File can not open\n");
}
else
{
printf("\nEnter text:\n");
while((ch=getchar())!=EOF)
{
fputc(ch,fp);
}
printf("File created Successfully.....\n");
}
fclose(fp);

getch();
}
```

C program to Read a File

```
#include<stdio.h>
#include<conio.h>
void main()
{
FILE *fp;
char fname[20];
int ch;
clrscr();

printf("Enter File name:");
scanf("%s",fname);

fp=fopen(fname,"r");

if(fp==NULL)
{
printf("File can not open\n");
}
else
{
while((ch=getc(fp))!=EOF)
{
printf("%c",ch);
}
printf("File Opened Successfully.....\n");
}
fclose(fp);

getch();
}
```

5.4 Standard C- Preprocessors

C – Processor is a program that processes our source program before it is passed to the compiler. Preprocessor commands (often known as directives) form what can almost be considered a language within C language.

*"The processor operates under the control of what is known as pre-*processor *or command line or directive"*

pre- processor directive:

#define	It defines the macro substitution .
#include	Specify the file to be included
#if	Test a compile time condition
#else	Specify alternative when #if test fails
#ifdef	Test whether micro is defined
#ifndef	Test whether micro is not defined
#Undef	Undefines a micro.

5.5 Defining and calling macros
Categories of Directive.
Macro Substitution Directives (Macro Expansion)

Simple Macro

```
#define a 20 #define b 30 void main()

{
clrscr();
printf("%d",(a+b));
getch();
}
```

Output is 50.

*Note: where a is known as **macro tamplet***
*20 is refers to **macro expention** in first line of above program .*

Another example

```
#define start
void main(){
#define clear_screen   clrscr();
#define print   printf("Hello World!!");
#define exit    getch();
}
start clear_screen
print exit
```

Output is Hello World!!

Augmented Macro

```
#define N 100
#define SQUARE(x) x*x

 void main()
{
int a=5; if(SQUARE(a)<N) printf("Hello"); else printf("World") ; getch();
}
```

Output is World

Nested Macro substitution

```
#define SQUARE(x)   x*x
#define CUBE(x)       (SQUARE(x)*(x))
#define SIXTH(x)      (CUBE(x)*CUBE(x))

void main()
{

printf("%d",SIXTH(2));
getch();

}
```

Output is 64

File Inclusion

This directive causes one file to be included in another.The preprocessor command for file inclusion looks like this:

#include "filename"

#include<filename>

The meaning of each of these forms is given below

# include<stdio.h>--- It searches one directory i.e specified list of directory.

# include"stdio.h"---- It searches two i.e specified list of directory & current directory.

Compiler Control Directive (Conditional Compilation)

if we want, have the compiler skip over part of a source code by inserting the pre-processing commands #ifdef and #endif, which have the general form:

```
#ifdef macroname

statement 1 ; statement 2 ; statement 3 ;
```

```
#endif
```

Suppose an organization has two different types of computers and you are expected to write a program that works on both the machines. You can do so by isolating the lines of code that must be different for each machine by marking them off with #ifdef.

For example:

```
main( )

{

#ifdef INTEL

code suitable for a Intel PC #else

code suitable for a Motorola PC #endif
code common to both the computers

}
```

When you compile this program it would compile only the code suitable for a Intel PC and the common code. This is because the macro INTEL has not been defined. Note that the working of #ifdef - #else - #endif is similar to the ordinary if - else control instruction of C.

5.6 Conditional compilation

Conditional compilation in c programming language: Conditional compilation as the name implies code is compiled if certain conditions hold true. Normally we use if keyword for checking some condition so we have to use something different so that compiler can determine whether to compile the code or not. The different thing is #if.
Now consider the following code to quickly understand the scenario:

Conditional compilation example in c language
Example 1

```
#include <stdio.h>

int main()
{
  #define COMPUTER "An amazing device"

  #ifdef COMPUTER
    printf(COMPUTER);
  #endif

  return 0;
}
```

Example 2

```
#include <stdio.h>

#define x 10

main()
{
   #ifdef x
     printf("hello\n");      //this is compiled as  x is defined
  #else
     printf("bye\n");       //this is not compiled
  #endif

  return 0;
}
```

5.7 Command Line Argument / passing values to the compiler.

It is possible to pass some values from the command line to your C programs when they are executed. These values are called **command line arguments** and many times they are important for your program specially when you want to control your program from outside instead of hard coding those values inside the code.
The command line arguments are handled using main() function arguments where **argc** refers to the number of arguments passed, and **argv[ ]** is a pointer array which points to each argument passed to the program. Following is a simple example which checks if there is any argument supplied from the command line and take action accordingly:

```
#include <stdio.h>

int main( int argc, char *argv[] )
{
if( argc == 2 )
{
	printf("The argument supplied is %s\n", argv[1]);
}
else if( argc > 2 )
{
	printf("Too many arguments supplied.\n");
}

else
{
	printf("One argument expected.\n");
}
}
```

When the above code is compiled and executed with a single argument, it produces the following result.

```
$./a.out testing
The argument supplied is testing
```

When the above code is compiled and executed with a two arguments, it produces the following result.

```
$./a.out testing1 testing2
Too many arguments supplied.
```

When the above code is compiled and executed without passing any argument, it produces the following result.

```
$./a.out
One argument expected
```

It should be noted that **argv[0]** holds the name of the program itself and **argv[1]** is a pointer to the first command line argument supplied, and *argv[n] is the last argument. If no arguments are supplied, argc will be one, otherwise and if you pass one argument then **argc** is set at 2.

You pass all the command line arguments separated by a space, but if argument itself has a space then you can pass such arguments by putting them inside double quotes "" or single quotes ''. Let us re-write above example once again where we will print program name and we also pass a command line argument by putting inside double quotes:

```
#include <stdio.h>
int main( int argc, char *argv[] )
{
printf("Program name %s\n", argv[0]);
if( argc == 2 ) {
                    printf("The argument supplied is %s\n", argv[1]);
                            }
else if( argc > 2 )
{
                   printf("Too many arguments supplied.\n");
}
else
{printf("One argument expected.\n");}
}
```

When the above code is compiled and executed with a single argument separated by space but inside double quotes, it produces the following result.

```
$./a.out "testing1 testing2"
Program name ./a.out
The argument supplied is testing1 testing2
```

Questions

From 2013-14

1. Define double pointer with example.
2. Write short note on Macros with suitable example.
3. Explain the following: Preprocessor, Conditional Operator.
4. Write a program in C to copy content from one file to another file.
5. Write the difference between call by value and call by reference with suitable example.
   [UPTU 2013-14], [UPTU 2012-13], [UPTU 2008-09]

6. What is pointer arithmetic? Write advantage and disadvantage of using pointer variable. [UPTU 2012-13]
7. Write short notes on: Macros, Linked List, and Mathematical Function. [UPTU 2012-13]
8. Dynamic memory allocation. [UPTU 2011-12]
9. Macros. How they are different from C variables. Advantage of macro. Explain conditional compilation and how does it help the programmers. [UPTU 2011-12]
10. Pointers. Declare, initialize. Swap program. [UPTU 2011-12]
11. File program: Odd number in ODD file and even number in EVEN file. [UPTU 2011-12]
12. What is dynamic memory allocation? Explain *malloc()* with example. [UPTU 2008-09]
13. Write a program in C that takes ten integers from a file and write square of these integers into another file

From 2008-09

1. How macros are defined and called in C? Explain with example. [UPTU 2008-09]
2. What do you understand by pointer arithmetic? Explain. [UPTU 2007-08]

UPTU QUESTIONS OF C LANGUAGE (BEFORE 2007)

1. What are pointers in 'C'? How is a pointer initialized? Explain with a suitable example.
2. Using pointer, write a function that receives a character string and a character as argument and deletes all occurrences of this character in this string. Function should return the corrected string without space.
3. Write a C-program to display the address and the content of a pointer variable.
4. Discuss different modes in which a file can be opened by giving suitable examples.
5. Differentiate between: int abc [5] [10] and int * b [5];
6. Write a 'C' program using pointer to read in line of text, store it in memory and print the line backward.
7. Write a 'C' program to search and replace in a string using pointer and functions provided by the 'C' library.