Derived Types and File Handling

- Overview of Pointers
- Structure declaration and Structure of Array
- Enumerated and Union Types
- Sequential and Binary Files : Creation, Merging and Updating
- Command Line Arguments
- Preprocessor Directives

Overview of Pointers

- every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator, which denotes an address in memory.
- Consider the following example, which prints the address of the variables defined −

```
#include <stdio.h>
int main ()
{
   int var1;
   char var2[10];
   printf("Address of var1 variable: %x\n", &var1 );
   printf("Address of var2 variable: %x\n", &var2 );
   return 0;
}
```

Pointers (cont.)

- What are Pointers?
- A **pointer** is a variable whose value is the address of another variable, i.e., direct address of the memory location.
- Like any variable or constant, you must declare a pointer before using it to store any variable address.

Pointers (cont.)

- The general form of a pointer variable declaration is –
  - **type *var-name;**
- Here, **type** is the pointer's base type; it must be a valid C data type and **var-name** is the name of the pointer variable.
- The asterisk * used to declare a pointer.
- However, in this statement the asterisk is being used to designate a variable as a pointer.
- Take a look at some of the valid pointer declarations –
  - **int *ip;**
  - **double *dp;**
  - **float *fp;**
  - **char *ch;**

Pointers (cont.)

```
#include <stdio.h>
int main ()
{
    int var = 20;
    int *ip;
    ip = &var;
    printf("Address of var variable: %x\n", &var );
    printf("Address stored in ip variable: %x\n", ip );
    printf("Value of *ip variable: %d\n", *ip );
    return 0;
}
```

Pointers (cont.)

- Reference operator (&) and Dereference operator (*)
- & is called reference operator. It gives you the address of a variable.
- * is called dereference operator. It gets you the value from the address.
- **Note:** The * sign when declaring a pointer is not a dereference operator. It is just a similar notation that creates a pointer.

Pointers (cont.)

```
#include <stdio.h>
int main()
{
    int* pc;
    int c; c=22;
    printf("Address of c:%u\n",&c);
    printf("Value of c:%d\n\n",c);
    pc=&c;
    printf("Address of pointer pc:%u\n",pc);
     printf("Content of pointer pc:%d\n\n",*pc);
    return 0;
}
```

Structure

- Arrays allow to define type of variables that can hold several data items of the same kind.
- Similarly **structure** is another user defined data type available in C that allows to combine data items of different kinds.
- Structures are used to represent a record.
- Suppose you want to keep track of your books in a library.
- You might want to track the following attributes about each book −
- Title
- Author
- Subject
- Book ID

Structure declaration

- To define a structure, you must use the **struct** statement.
- The struct statement defines a new data type, with more than one member.
- The format of the struct statement is as follows –

```
struct [structure tag]
{
   member definition;
   member definition;
   ...
   member definition;
} [one or more structure variables];
```

Structure declaration (cont.)

- The **structure tag** is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition.
- At the end of the structure's definition, before the final semicolon, you can specify one or more structure variables but it is optional.
- Here is the way you would declare the Book structure –

```
struct Books
{
   char title[50];
   char author[50];
   char subject[100];
   int book_id;
} book;
```

Accessing Structure Members

- To access any member of a structure, we use the **member access operator (.)**.
- The member access operator is coded as a period between the structure variable name and the structure member that we wish to access.
- You would use the keyword **struct** to define variables of structure type.

Accessing Structure Members (cont.)

```
#include <stdio.h>
#include <string.h>
struct Books
{
char title[50];
char author[50];
char subject[100];
int book_id;
};
int main( )
{
struct Books Book1;
//struct Books Book[5];  //structure of array

strcpy( Book1.title, "C Programming");
strcpy( Book1.author ,"Nuha Ali“);
strcpy( Book1.subject ,"C Programming Tutorial“);
Book1.book_id = 6495407;

printf( "Book 1 title : %s\n", Book1.title);
printf( "Book 1 author : %s\n", Book1.author);
printf( "Book 1 subject : %s\n", Book1.subject);
printf( "Book 1 book_id : %d\n", Book1.book_id);

return 0;
}
```

Enumeration

- An enumeration is a user-defined data type that consists of integral constants.
- To define an enumeration, keyword **enum** is used.
  - enum flag { const1, const2, ..., constN };
- Here, name of the enumeration is **flag**.
- And, **const1**, **const2**,...., **constN** are values of type **flag**.
- By default, **const1** is 0, **const2** is 1 and so on.

```
enum suit
{
    club = 0,
    diamonds = 10,
    hearts = 20,
    spades = 3,
};
```

Enumeration Example

```
#include <stdio.h>
enum week { sunday, monday, tuesday, wednesday,
   thursday, friday, saturday };
int main()
{
   enum week today;
   today = wednesday;
   printf("Day %d",today+1);
   return 0;
}
```

Union

- A **union** is a special data type available in C that allows to store different data types in the same memory location.
- You can define a union with many members, but only one member can contain a value at any given time.
- To define a union, you must use the **union** statement.
- The union statement defines a new data type with more than one member for your program.

Union (cont.)

union [union tag]
{
member definition;
member definition;
...
member definition;
} [one or more union variables];

- The **union tag** is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition.
- At the end of the union's definition, before the final semicolon, you can specify one or more union variables but it is optional.

Union Example

```
union MyData
{
int i;
float f;
char str[20];
} data;
```

- A variable of **Data** type can store an integer, a floating-point number, or a string of characters.
- It means a single variable, i.e., same memory location, can be used to store multiple types of data.
- You can use any built-in or user defined data types inside a union based on your requirement.
- The memory occupied by a union will be large enough to hold the largest member of the union.
- For example, in the above example, MyData type will occupy 20 bytes of memory space because this is the maximum space which can be occupied by a character string.

File Handling

- What is File?
- A **file** represents a sequence of bytes on the disk where a group of related data is stored. File is created for permanent storage of data.
- To declare a file: FILE ***fp**;

File types

- Sequential file – Text file
- A sequential file is a file in which records are stored one after another in some order.
- Text files contain ASCII codes of digits, alphabetic and symbols.

- Binary file
- A Binary file is similar to the text file, but it contains only large numerical data.
- Binary file contains collection of bytes (0's and 1's). Binary files are compiled version of text files.

Steps

- Naming a file
- Opening a file
- Reading data from file
- Writing data into file
- Closing a file

Sequential file operations

- fopen() - create a new file or open a existing file
- fclose() - closes a file
- getc() - reads a character from a file
- putc() - writes a character to a file
- fscanf() - reads a set of data from a file
- fprintf() - writes a set of data to a file
- getw() - reads a integer from a file
- putw() - writes a integer to a file

Defining and Opening a File

- Data structure of file is defined as FILE in the standard I/O function. So all files should be declared as type FILE.
- Before opening any file we need to specify for which purpose we open file, for example file open for write or read purpose.

```
FILE *fp;
pf=fopen("filename", "mode");
```

File Opening mode

S.No	Mode	Meaning	Purpose
1	r	Reading	Open the file for reading only.
2	w	Writing	Open the file for writing only.
3	a	Appending	Open the file for appending (or adding) data to it.
4	r+	Reading + Writing	New data is written at the beginning override existing data.
5	w+	Writing + Reading	Override existing data.
6	a+	Reading + Appending	To new data is appended at the end of file.

Input/Output Operation on files

S.No	Function	Operation	Syntax
1	getc()	Read a character from a file	getc(fp)
2	putc()	Write a character in file	putc(c, fp)
3	fprintf()	To write set of data in file	fprintf(fp, "control string", list)
4	fscanf()	To read set of data from file.	fscanf(fp, "control string", list)
5	getw()	To read an integer from a file.	getw(fp)
6	putw()	To write an integer in file.	putw(integer, fp)

Closing a File

- A file must be close after completion of all operation related to file. For closing file we need **fclose()** function.
- fclose(Filepointer);

```
#include <stdio.h>

void main()
{
FILE *fptr;
char name[20];
int age;
float salary;

/* open for writing */
fptr = fopen("emp.txt", "w");

if (fptr == NULL)
{
printf("File does not exists \n");
return;
}
```

```
printf("Enter the name \n");
scanf("%s", name);

fprintf(fptr, "Name = %s\n", name);

printf("Enter the age\n");
scanf("%d", &age);

fprintf(fptr, "Age = %d\n", age);

printf("Enter the salary\n");
scanf("%f", &salary);

fprintf(fptr, "Salary = %.2f\n",
     salary);

fclose(fptr);
}
```

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

void main()
{
FILE *fptr;
char filename[15];
char ch;

printf("Enter the filename to be opened
    \n");
scanf("%s", filename);

/* open the file for reading */
fptr = fopen(filename, "r");
if (fptr == NULL)
{
printf("Cannot open file \n");
exit(0);
}
ch = fgetc(fptr);
while (ch != EOF)
{
printf ("%c", ch);
ch = fgetc(fptr);
}
fclose(fptr);
}
```

```
#include <stdio.h>

int main()
{
FILE *fileptr;
int count_lines = 0;
char filechar[40], chr;

printf("Enter file name: ");
scanf("%s", filechar);
fileptr = fopen(filechar, "r");
//extract character from file and
    store in chr
chr = getc(fileptr);
while (chr != EOF)
{
//Count whenever new line is
    encountered
if (chr == 'n')
{
count_lines = count_lines + 1;
}
//take next character from file.
chr = getc(fileptr);
}
fclose(fileptr); //close file.
printf("There are %d lines in %s in a
    file\n", count_lines, filechar);
return 0;
}
```

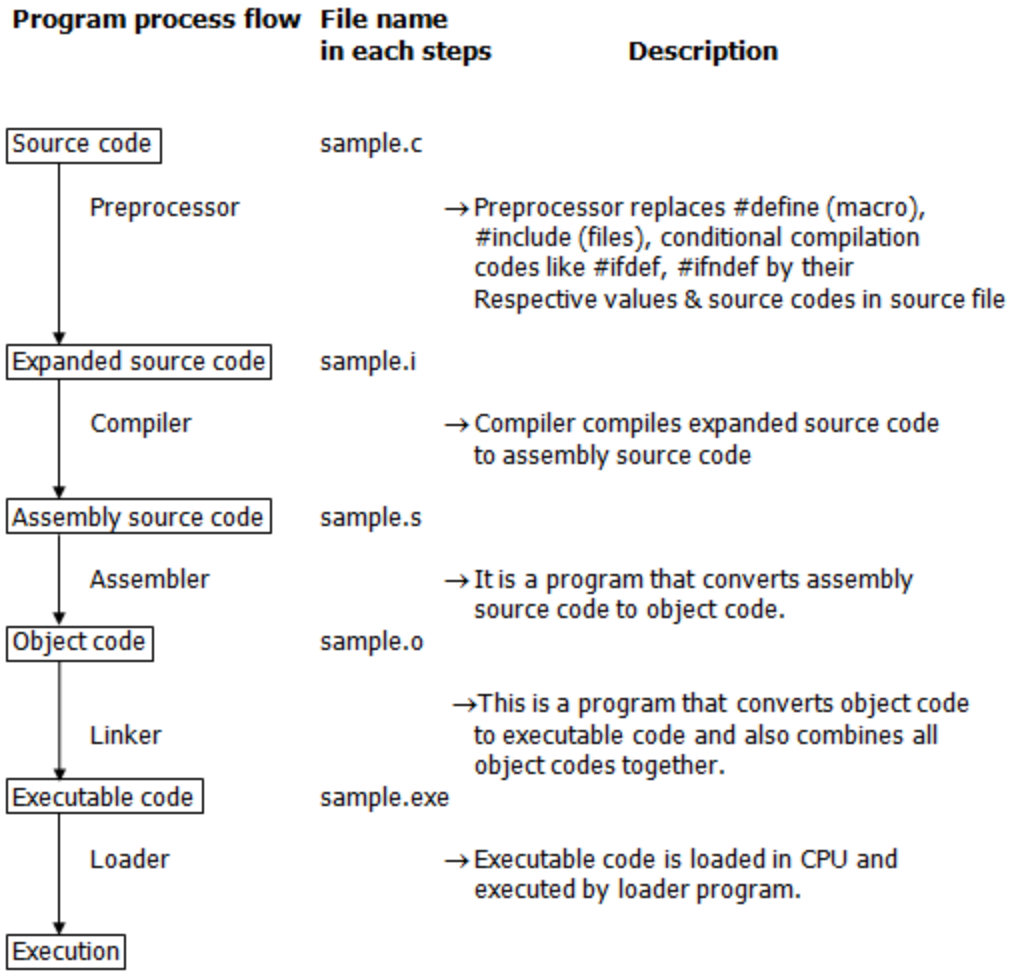
Binary file operations

- fopen() - create a new file or open a existing file
- fclose() - closes a file
- fread() - to read
- fwrite() – to write
- fseek() - set the position to desire point
- ftell() - gives current position in the file
- rewind() - set the position to the beginning point

Preprocessor Directives

- Before a C program is compiled in a compiler, source code is processed by a program called preprocessor. This process is called preprocessing.
- Commands used in preprocessor are called preprocessor directives and they begin with “#” symbol.
- The **C Preprocessor** is not a part of the compiler, but is a separate step in the compilation process.
- List of preprocessor directives:
  - Macro
  - Header file inclusion
  - Conditional compilation

- A program in C language involves into different processes. Below diagram will help you to understand all the processes that a C program comes across.



Command Line Arguments

- 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**
- Many times they are important for your program especially 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");
    }
}
```

- main() function of a C program accepts arguments from command line or from other shell scripts by following commands. They are,
- argc
- argv[]
- where,
- argc – Number of arguments in the command line including program name
  argv[] – This is carrying all the arguments
- In real time application, it will happen to pass arguments to the main program itself. These arguments are passed to the main () function while executing binary file from command line.
- For example, when we compile a program (test.c), we get executable file in the name “test”.
- Now, we run the executable “test” along with 4 arguments in command line like below.
- **./test this is a program**
- Where,
- argc = 5
  argv[0] = “test”
  argv[1] = “this”
  argv[2] = “is”
  argv[3] = “a”
  argv[4] = “program”
  argv[5] = NULL

Output

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

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

$./a.out
One argument expected

Cont.

- 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, 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 ''.

```
#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");
    }
}
```

Output

$./a.out "testing1 testing2"

Programm name ./a.out

The argument supplied is testing1 testing2