C Programming Essentials
Unit 4: System Programming

CHAPTER 11: FILE AND I/O

DR. ERIC CHOU

IEEE SENIOR MEMBER

Objectives

- A file is a container in computer storage devices used for storing data.
- In this chapter, you will learn about file handling in C. You will learn to handle standard I/O in C using fprintf(), fscanf(), fread(), fwrite(), fseek() etc. with the help of examples.

LECTURE 1

Files

Why files are needed?

- When a program is terminated, the entire data is lost. Storing in a file will preserve your data even if the program terminates.
- If you have to enter a large number of data, it will take a lot of time to enter them all.
- However, if you have a file containing all the data, you can easily access the contents of the file using a few commands in C.
- You can easily move your data from one computer to another without any changes.

Types of Files

When dealing with files, there are two types of files you should know about:

1. Text files
2. Binary files

Text files

- Text files are the normal .txt files. You can easily create text files using any simple text editors such as Notepad.
- When you open those files, you'll see all the contents within the file as plain text. You can easily edit or delete the contents.
- They take minimum effort to maintain, are easily readable, and provide the least security and takes bigger storage space.

Binary files

- Binary files are mostly the .bin files in your computer.
- Instead of storing data in plain text, they store it in the binary form (0's and 1's).
- They can hold a higher amount of data, are not readable easily, and provides better security than text files.

LECTURE 2

File Operations

What is a File

- A file is a collection of related data
- *"C"* treats files as a series of bytes
- Basic library routines for file I/O

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

File Operations

In C, you can perform four major operations on files, either text or binary:

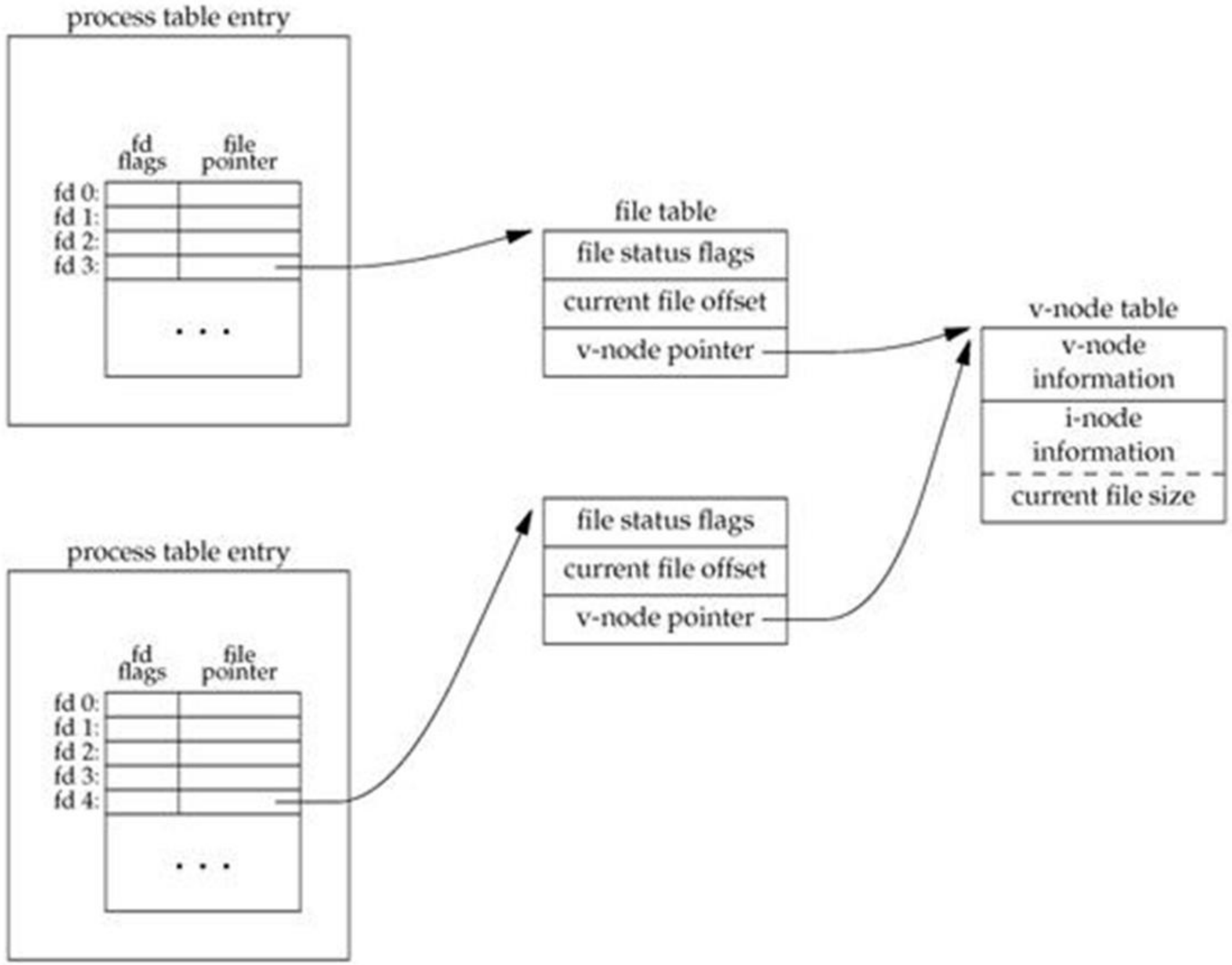
1. Creating a new file
2. Opening an existing file
3. Closing a file
4. Reading from and writing information to a file

Working with files

- When working with files, you need to declare a pointer of type file. This declaration is needed for communication between the file and the program.

```
FILE *fptr;
```

File Descriptor / File Handler



File Types

- Text (ASCII) files
- Binary files
- Special (device) files

  stdin - standard input *(open for reading)*

  stdout - standard output *(open for writing)*

  stderr - standard error (*open for writing)*

Opening a file - for creation and edit

- Opening a file is performed using the `fopen()` function defined in the `stdio.h` header file.
- The syntax for opening a file in standard I/O is:

```
ptr = fopen("fileopen","mode");
```

- For example,

```
fopen("E:\\cprogram\\newprogram.txt","w");
fopen("E:\\cprogram\\oldprogram.bin","rb");
```

Basics About Files

Files must be opened and closed

```
#include <stdio.h>
. . .
FILE  * myFile;
myFile = fopen ("C:\\data\\myfile.txt", "r");   //
 Name, Mode (r: read)
if ( myFile == NULL ){                       // (w: write)
     /* Could not open the file */
     ...
}
. . .
fclose ( myFile );
```

Note: status = fclose(file-variable)

status = **0** if file closed successfully- Error otherwise.

Opening a file - for creation and edit

- Let's suppose the file `newprogram.txt` doesn't exist in the location `E:\cprogram`. The first function creates a new file named `newprogram.txt` and opens it for writing as per the mode 'w'.

- The writing mode allows you to create and edit (overwrite) the contents of the file.

- Now let's suppose the second binary file `oldprogram.bin` exists in the location `E:\cprogram`. The second function opens the existing file for reading in binary mode `'rb'`.

- The reading mode only allows you to read the file, you cannot write into the file.

Operations with Files

- Reading (r)
  - sequential
  - random

- Writing (w)
  - sequential
  - random
  - appending (a)

- fopen() revisited

```
FILE *fOut;
fOut = fopen("c:\\data\\log.txt",  "w" );
```

Opening Modes in Standard I/O

Mode	Meaning of Mode	During Inexistence of file
r	Open for reading.	If the file does not exist, fopen() returns NULL.
rb	Open for reading in binary mode.	If the file does not exist, fopen() returns NULL.
w	Open for writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
wb	Open for writing in binary mode.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.

Opening Modes in Standard I/O

Mode	Meaning of Mode	During Inexistence of file
a	Open for append. Data is added to the end of the file.	If the file does not exist, it will be created.
ab	Open for append in binary mode. Data is added to the end of the file.	If the file does not exist, it will be created.
r+	Open for both reading and writing.	If the file does not exist, fopen() returns NULL.
rb+	Open for both reading and writing in binary mode.	If the file does not exist, fopen() returns NULL.
w+	Open for both reading and writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
wb+	Open for both reading and writing in binary mode.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
a+	Open for both reading and appending.	If the file does not exist, it will be created.
ab+	Open for both reading and appending in binary mode.	If the file does not exist, it will be created.

End-line Character

- Teletype Model 33 *(long time ago...) used 2 characters at the end of line.*
  - RETURN character
  - LINE FEED character
- Computer age
  - UNIX: LINE FEED at the end of line: "\n"
  - MS-DOS/Windows: both characters: "\n\r"
  - Apple: RETURN at the end of line: "\r"

Closing a File

- The file (both text and binary) should be closed after reading/writing.
- Closing a file is performed using the `fclose()` function.

```
    fclose(fptr);
```

- Here, fptr is a file pointer associated with the file to be closed.

FILE I/O

fopen()	opens a file
fclose()	closes a file
fputc()	writes a character to a file
fgetc()	reads a character from a file
fputs()	writes a string to a file
fgets()	reads a string to a file
fseek()	change file position indicator
ftell()	returns to file position indicator
fprintf()	similar to printf(), but to a file instead of console
fscanf()	similar to scanf(), but to a file instead of console
remove()	deletes the file
fflush()	flushes the file pipe

Useful File I/O Functions

fopen(), fclose()	-- open/close files
fprintf (myFile, "format...", ...)	-- formatted I/O
fscanf (myFile, "format...", ...)	
fgets(), fputs()	-- for line I/O
fgetc(), fputc()	-- for character I/O
feof()	-- end of file detection, when reading

Reading and writing to a text file

- For reading and writing to a text file, we use the functions `fprintf()` and `fscanf()`.
- They are just the file versions of `printf()` and `scanf()`. The only difference is that `fprint()` and `fscanf()` expects a pointer to the structure **`FILE`**.

Example 1: Write to a text file

Demo Program: fwrite1.c

- This program takes a number from the user and stores in the file `program.txt`.
- After you compile and run this program, you can see a text file `program.txt` created in C drive of your computer. When you open the file, you can see the integer you entered.

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

int main(){
  int num;
  FILE *fptr;

  // use appropriate location if you are using MacOS or Linux
  fptr = fopen("C:program.txt","w");
  if(fptr == NULL) {
    printf("Error!");
    exit(1);
  }

  printf("Enter num: ");
  scanf("%d",&num);
  fprintf(fptr,"%d",num);
  fclose(fptr);
  return 0;
}
```

Example 2: Read from a text file

Demo Program: fread1.c

- This program reads the integer present in the program.txt file and prints it onto the screen.
- If you successfully created the file from Example 1, running this program will get you the integer you entered.
- Other functions like `fgetchar()`, `fputc()` etc. can be used in a similar way.

```
Value of n=35
```

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

int main(){
  int num;
  FILE *fptr;

  if ((fptr = fopen(“program.txt","r")) == NULL){
     printf("Error! opening file");

     // Program exits if the file pointer returns NULL.
     exit(1);
  }

  fscanf(fptr,"%d", &num);
  printf("Value of n=%d", num);
  fclose(fptr);
  return 0;
}
```

LECTURE 1

Binary Files

Binary and Random I/O

Binary I/O

```
readSize = fread(dataPtr, 1, size, myFile);
//size of data read, if < size then encountered an error.
writeSize = fwrite(dataPtr, 1, size, myFile);
```

Positioning for random I/O

```
fseek(myFile, 0, SEEK_SET);
fseek(myFile, 10, SEEK_CUR);
fseek(myFile, 0, SEEK_END);
```

Buffered v.s. Unbuffered I/O

//no immediate write to file, instead buffer data and then flush after program finished

Buffered I/O may improve performance

Buffered I/O is with f...() functions

- fopen(), fwrite()

Unbuffered I/O

- open(), write()

Streams v.s. Records

Stream - a file interpreted as a stream of bytes

Record set - a file interpreted as a set of records, structures

- The structures can be of the same size, or
- each record can be of different size

Example: En/De-Crypter

```
int main ( int argc, char * argv[] ) {
 FILE *in, *out;
 in = fopen ( argv[1], "rb");
 out = fopen ( argv[2], "wb");
 if ( ! in || ! out ){
         printf( "Error opening files ! \n" );
         return -1;
 }
 while( ! feof ( in ) ){
         ch = fgetc ( in );
         fputc ( (ch ^ 0xFF) , out );      //UTF-16 vs UTF-8 (Unicode Byte Order mark)
 }                                         //Unicode Transformation Format
 return 0;
}
```

Reading and writing to a binary file

- Functions `fread()` and `fwrite()` are used for reading from and writing to a file on the disk respectively in case of binary files.

Writing to a binary file

To write into a binary file, you need to use the fwrite() function. The functions take four arguments:

1. address of data to be written in the disk
2. size of data to be written in the disk
3. number of such type of data
4. pointer to the file where you want to write.

```
fwrite(addressData, sizeData, numbersData, pointerToFile);
```

Example 3: Write to a binary file using fwrite()

Demo Program: bwrite1.c

- In this program, we create a new file `program.bin` in the C drive.
- We declare a structure `threeNum` with three numbers - `n1`, `n2` and `n3`, and define it in the main function as num.
- Now, inside the for loop, we store the value into the file using `fwrite()`.
- The first parameter takes the address of `num` and the second parameter takes the size of the structure `threeNum`.
- Since we're only inserting one instance of `num`, the third parameter is `1`. And, the last parameter `*fptr` points to the file we're storing the data.
- Finally, we close the file.

```c
#include <stdio.h>
#include <stdlib.h>
struct threeNum{
  int n1, n2, n3;
};
int main(){
  int n;
  struct threeNum num;
  FILE *fptr;
  if ((fptr = fopen("program.bin","wb")) == NULL){
     printf("Error! opening file");
     // Program exits if the file pointer returns NULL.
     exit(1);
  }
  for(n = 1; n < 5; ++n){
    num.n1 = n;
    num.n2 = 5*n;
    num.n3 = 5*n + 1;
    fwrite(&num, sizeof(struct threeNum), 1, fptr);
  }
  fclose(fptr);
  return 0;
}
```

Reading from a binary file

- Function `fread()` also take 4 arguments similar to the `fwrite()` function as above.

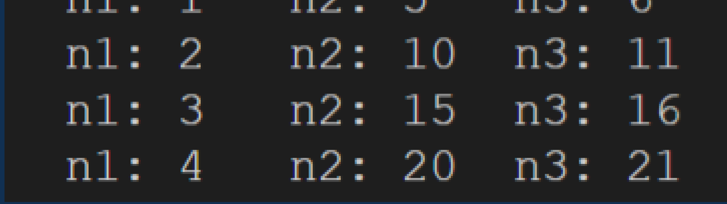
```
fread(addressData, sizeData, numbersData, pointerToFile);
```

Example 4: Read from a binary file using fread()

Demo Program: bread1.c

- In this program, you read the same file `program.bin` and loop through the records one by one.
- In simple terms, you read one `threeNum` record of `threeNum` size from the file pointed by `*fptr` into the structure num.
- You'll get the same records you inserted in Example 3.

```c
#include <stdio.h>
#include <stdlib.h>
struct threeNum{
  int n1, n2, n3;
};

int main(){
  int n;
  struct threeNum num;
  FILE *fptr;

  if ((fptr = fopen("program.bin","rb")) == NULL){
     printf("Error! opening file");
     // Program exits if the file pointer returns NULL.
     exit(1);
  }

  for(n = 1; n < 5; ++n){
    fread(&num, sizeof(struct threeNum), 1, fptr);
    printf("n1: %d\tn2: %d\tn3: %d\n", num.n1, num.n2, num.n3);
  }
  fclose(fptr);
  return 0;
}
```



```
n1: 1    n2: 5    n3: 6
n1: 2    n2: 10   n3: 11
n1: 3    n2: 15   n3: 16
n1: 4    n2: 20   n3: 21
```

LECTURE 3

Random Access

Getting data using fseek()

If you have many records inside a file and need to access a record at a specific position, you need to loop through all the records before it to get the record.

This will waste a lot of memory and operation time. An easier way to get to the required data can be achieved using `fseek()`.

As the name suggests, `fseek()` seeks the cursor to the given record in the file.

Syntax of fseek()

```
fseek(FILE * stream, long int offset, int whence);
```

The first parameter stream is the pointer to the file. The second parameter is the position of the record to be found, and the third parameter specifies the location where the offset starts.

Different whence in fseek()

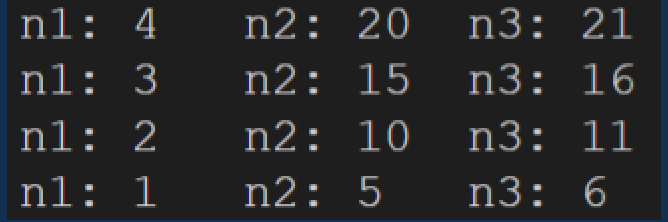
Whence	Meaning
SEEK_SET	Starts the offset from the beginning of the file.
SEEK_END	Starts the offset from the end of the file.
SEEK_CUR	Starts the offset from the current location of the cursor in the file.

Example 5: fseek()

Demo Program: bseek1.c

- This program will start reading the records from the file `program.bin` in the reverse order (last to first) and prints it.

```c
#include <stdio.h>
#include <stdlib.h>
struct threeNum{
  int n1, n2, n3;
};
int main(){
  int n;
  struct threeNum num;
  FILE *fptr;
  if ((fptr = fopen("program.bin","rb")) == NULL){
     printf("Error! opening file");
     // Program exits if the file pointer returns NULL.
     exit(1);
  }
  // Moves the cursor to the end of the file
  fseek(fptr, (long int) (-sizeof(struct threeNum)), SEEK_END);
  for(n = 1; n < 5; ++n){
    fread(&num, (long int) sizeof(struct threeNum), 1, fptr);
    printf("n1: %d\tn2: %d\tn3: %d\n", num.n1, num.n2, num.n3);
    fseek(fptr, (long int) (-2*sizeof(struct threeNum)), SEEK_CUR);
  }
  fclose(fptr);
  return 0;
}
```



```
n1: 4   n2: 20  n3: 21
n1: 3   n2: 15  n3: 16
n1: 2   n2: 10  n3: 11
n1: 1   n2: 5   n3: 6
```

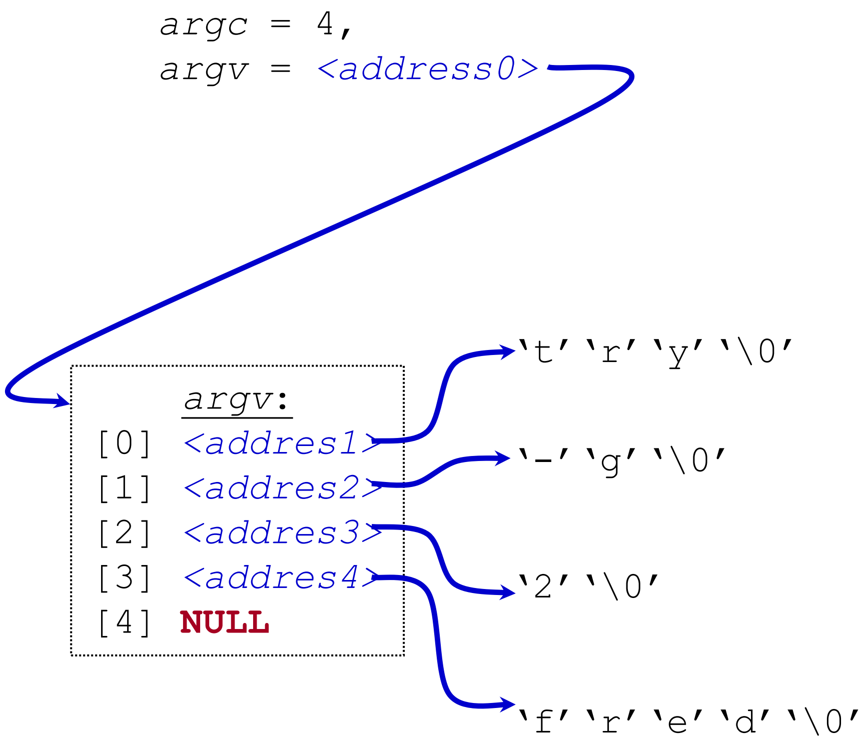
LECTURE 4

Command Line Arguments

Passing Command Line Arguments

- When you execute a program you can include arguments on the command line.
- The run time environment will create an argument vector.
  - `argv` is the argument vector
  - `argc` is the number of arguments
- Argument vector is an array of pointers to strings.
- a *string* is an array of characters terminated by a binary 0 (NULL or '\0').
- *argv[0]* is always the program name, so *argc* is at least 1.

```
./try -g 2 fred
```



```
argc = 4,
argv = <address0>

argv:
[0] <addres1>   't' 'r' 'y' '\0'
[1] <addres2>   '-' 'g' '\0'
[2] <addres3>   '2' '\0'
[3] <addres4>   'f' 'r' 'e' 'd' '\0'
[4] NULL
```

Supplement topic – I/O from console

- Reading from console
- During program execution
  - printf(), scanf(), putc(), getc()
- Just before execution starts (parameters passed to the program)

```
$ ./a.out 3 santa_singh banta_singh happy_singh
```

```
int main(int argc, char *argv[])
```

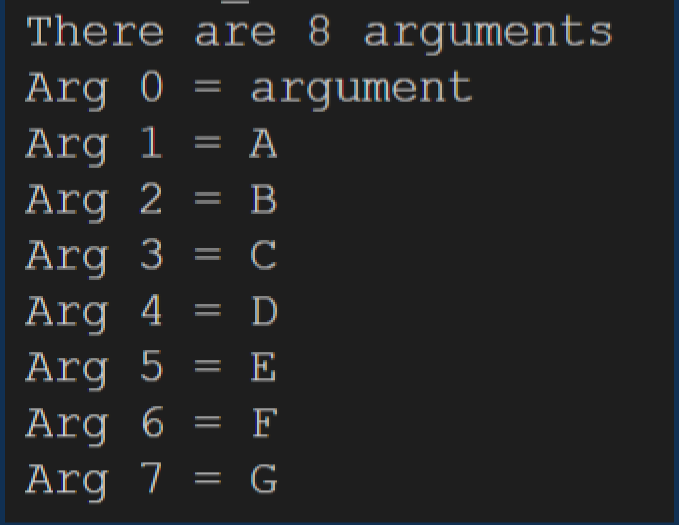
  - argc: number of arguments (in above case, 5)
  - argv: pointer to array of char pointers

Another Simple C Program

```
int main (int argc, char **argv) {
  int i;
  printf(“There are %d arguments\n”, argc);
  for (i = 0; i < argc; i++)
    printf(“Arg %d = %s\n”, i, argv[i]);

  return 0;
}
```

- Notice that the syntax is similar to Java
- What’s new in the above simple program?
  - of course you will have to learn the new interfaces and utility functions defined by the C standard and UNIX
  - Pointers will give you the most trouble

```c
#include <stdio.h>

int main (int argc, char **argv) {
  int i;
  printf("There are %d arguments\n", argc);
  for (i = 0; i < argc; i++)
    printf("Arg %d = %s\n", i, argv[i]);

  return 0;
}
```



```
There are 8 arguments
Arg 0 = argument
Arg 1 = A
Arg 2 = B
Arg 3 = C
Arg 4 = D
Arg 5 = E
Arg 6 = F
Arg 7 = G
```

LECTURE 5

Option Lines

Options on Argument line

- Normally, getopt is called in a loop. When getopt returns -1, indicating no more options are present, the loop terminates.
- A switch statement is used to dispatch on the return value from getopt. In typical use, each case just sets a variable that is used later in the program.
- A second loop is used to process the remaining non-option arguments.

Parsing program options using getopt

Function:

```
int getopt(int argc, char *const *argv, const char *options)
```

`argc`: argument count integer
`argv`: argument vector array of char
`options`: option string

Shared Global Variables (Class Variable)

int **opterr**

If the value of this variable is nonzero, then getopt prints an error message to the standard error stream if it encounters an unknown option character or an option with a missing required argument. This is the default behavior. If you set this variable to zero, getopt does not print any messages, but it still returns the character ? to indicate an error.

int **optopt**

When getopt encounters an unknown option character or an option with a missing required argument, it stores that option character in this variable. You can use this for providing your own diagnostic messages.

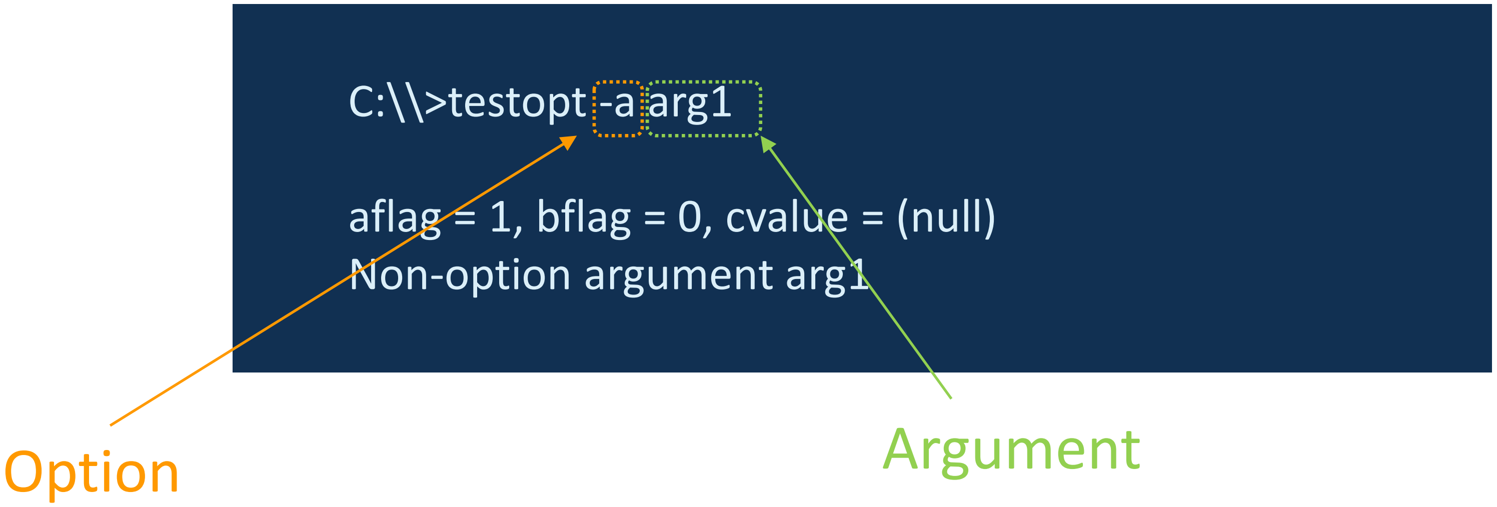
int **optind**

This variable is set by getopt to the index of the next element of the *argv* array to be processed. Once getopt has found all of the option arguments, you can use this variable to determine where the remaining non-option arguments begin. The initial value of this variable is 1.

*char ** **optarg**

This variable is set by getopt to point at the value of the option argument, for those options that accept arguments.

Argument and Options



```
C:\\>testopt -a arg1

aflag = 1, bflag = 0, cvalue = (null)
Non-option argument arg1
```

GETOPT

```
GETOPT (3) Linux Programme r’s Manual

        GETOPT (3)


        NAME
                getopt , getopt_long , getopt_long_only , optarg , optind , opterr , optopt -
                Parse command - line options

        SYNOPSIS
                # include <unistd .h>

                int getopt (int argc , char * const argv [], const char * optstring );

                extern char * optarg ;
                extern int optind , opterr , optopt ;

                # include <getopt .h>

                int getopt_long (int argc , char * const argv [],
                const char * optstring , const struct option * longopts , int * longindex );


```

```c
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main (int argc, char **argv){
 int aflag = 0;
 int bflag = 0;
 char *cvalue = NULL;
 int index;
 int c;
 opterr = 0;
 while ((c = getopt (argc, argv, "abc:")) != -1){
  switch (c){
   case 'a':
    aflag = 1;
    break;
   case 'b':
    bflag = 1;
    break;
   case 'c':
    cvalue = optarg;
    break;
   case '?':
    if (optopt == 'c')
     fprintf (stderr, "Option -%c requires an argument.\n", optopt);
    else if (isprint (optopt))
     fprintf (stderr, "Unknown option `-%c'.\n", optopt);
    else
     fprintf (stderr, "Unknown option character `\\x%x'.\n", optopt);
    return 1;
   default:
    abort ();
   }
 }
 printf ("aflag = %d, bflag = %d, cvalue = %s\n", aflag, bflag, cvalue);
 for (index = optind; index < argc; index++) printf ("Non-option argument %s\n", argv[index]);
```

```
C:\\>gcc testopt.c -o testopt

C:\\>testopt
aflag = 0, bflag = 0, cvalue = (null)

C:\\>testopt -a -b
aflag = 1, bflag = 1, cvalue = (null)

C:\\>testopt -ab
aflag = 1, bflag = 1, cvalue = (null)

C:\\>testopt -c foo
aflag = 0, bflag = 0, cvalue = foo

C:\\>testopt -cfoo
aflag = 0, bflag = 0, cvalue = foo

C:\\>testopt -a arg1
aflag = 1, bflag = 0, cvalue = (null)
Non-option argument arg1
```

```
C:\\>testopt -c foo arg1
aflag = 0, bflag = 0, cvalue = foo
Non-option argument arg1

C:\\>testopt -a -- -b
aflag = 1, bflag = 0, cvalue = (null)
Non-option argument -b

C:\\>testopt -a -
aflag = 1, bflag = 0, cvalue = (null)
Non-option argument -
```

LECTURE 6

Conclusion

Conclusion

- C, as a system programming, strongly related to operation system.
- In this chapter, we briefly discuss the file operations and the command line arguments and options.
- It serves as a very good guide for system programming