Data and File Structures Laboratory

Review of C – More Input/Output, File Handling, Header Files, Multi-file Programs

Malay Bhattacharyya

Assistant Professor

Machine Intelligence Unit
Indian Statistical Institute, Kolkata

July, 2018

More on getchar()

Reading a number from input using getchar():

```
if(isdigit(c))
x = x * 10 + c - '0';
```

More on getchar()

Taking a single character input from the user:

```
int c;
c = getchar();
```

taking a series of character inputs from the user:

```
int c;
while ((c = getchar()) != EOF) {
    ...
}
```

File handling

Opening a file: `fopen(filename, mode);`
filename: String (`char *`) containing name of file
mode: String specifying whether file is to be opened in read/write mode

File handling

Opening a file: `fopen(filename, mode);`
filename: String (`char *`) containing name of file
mode: String specifying whether file is to be opened in read/write mode

- `"r"`, `"w"`, `"a"`: read mode, write mode, append mode
- `"r+"`, `"w+"`, `"a+"`: read/write mode, write/read mode, read/append mode

File handling

Opening a file: `fopen(filename, mode);`
filename: String (`char *`) containing name of file
mode: String specifying whether file is to be opened in read/write mode

- `"r"`, `"w"`, `"a"`: read mode, write mode, append mode
- `"r+"`, `"w+"`, `"a+"`: read/write mode, write/read mode, read/append mode

Example:

```
FILE *fp;
if((fp = fopen("a.txt", "r")) == NULL)
    ERR_MESG("Error opening file");
```

File handling

Opening a file: `fopen(filename, mode);`
filename: String (`char *`) containing name of file
mode: String specifying whether file is to be opened in read/write mode

- `"r"`, `"w"`, `"a"`: read mode, write mode, append mode
- `"r+"`, `"w+"`, `"a+"`: read/write mode, write/read mode, read/append mode

Example:

```
FILE *fp;
if((fp = fopen("a.txt", "r")) == NULL)
    ERR_MESG("Error opening file");
```

Closing a file: `fclose(fp);`

File handling

Reading/writing text:

fgetc(fp): reads and returns the next character from `fp`, or EOF on end of file or error

Typical usage: while (EOF != (c = fgetc(fp))) ...

fgets(s, n, fp): reads at most `n-1` characters or one line (whichever is shorter), stores input in character array `s` and terminates `s` using '\0';
returns `s` or `NULL` on end of file (i.e., there is nothing to be read) or error

Typical usage: while (NULL != fgets(s, n, fp)) ...

fputc(c, fp): writes `c` to `fp`

fputs(s, fp): writes string `s` to `fp`

File handling

Reading / writing data:

fread((void *) buffer, sz, n, fp): reads `n` elements of data, each of size `sz` bytes from `fp`, stores them in `buffer`; returns number of elements read.

fwrite((void *) buffer, sz, n, fp): writes `n` elements of data from `buffer`, each of size `sz` bytes to `fp`; returns number of elements written.

Header files

Contents:

- Preprocessor directives and macros
- Constant declarations
- Type declarations (enum, typedef, struct, union, etc.)
- Function prototype declarations
- Global variable declarations
- Static function definitions (may contain)

Header files

Contents:

- Preprocessor directives and macros
- Constant declarations
- Type declarations (enum, typedef, struct, union, etc.)
- Function prototype declarations
- Global variable declarations
- Static function definitions (may contain)

Example: HelloWorld.h

```
#ifndef _HELLOWORLD_H_
#define _HELLOWORLD_H_
typedef unsigned int my_uint_t;
void printHelloWorld();
int iMyGlobalVar;
...
#endif
```

Multi-file programs

The motivations behind using multi-file programs are as follows:

1. Manageability
2. Modularity
3. Reusability
4. Abstraction

Multi-file programs

The motivations behind using multi-file programs are as follows:

1. Manageability
2. Modularity
3. Reusability
4. Abstraction

The general abstractions used in multi-file programs are as listed below.

- Header files
- Implementation source files
- Application source file (contains the main() function)

Header files

Contents:

- Preprocessor directives and macros
- Constant declarations
- Type declarations (enum, typedef, struct, union, etc.)
- Function prototype declarations
- Global variable declarations
- Static function definitions (may contain)

Header files

Contents:

- Preprocessor directives and macros
- Constant declarations
- Type declarations (enum, typedef, struct, union, etc.)
- Function prototype declarations
- Global variable declarations
- Static function definitions (may contain)

Example: HelloWorld.h

```
#ifndef _HELLOWORLD_H_
#define _HELLOWORLD_H_
typedef unsigned int my_uint_t;
void printHelloWorld();
int iMyGlobalVar;
...
#endif
```

Implementation source files

Contents:

- Function body for functions declared in corresponding header files
- Statically defined and inlined functions
- Global variable definitions

Implementation source files

Contents:

- Function body for functions declared in corresponding header files
- Statically defined and inlined functions
- Global variable definitions

Example: HelloWorld.c

```
#include<stdio.h>
#include "HelloWorld.h"
void printHelloWorld(){
    iMyGlobalVar = 20;
    printf("Hello World\n");
    return;
}
```

Application source file

Contents:

- Function body for the main() function
- Acts as client for the different modules

Application source file

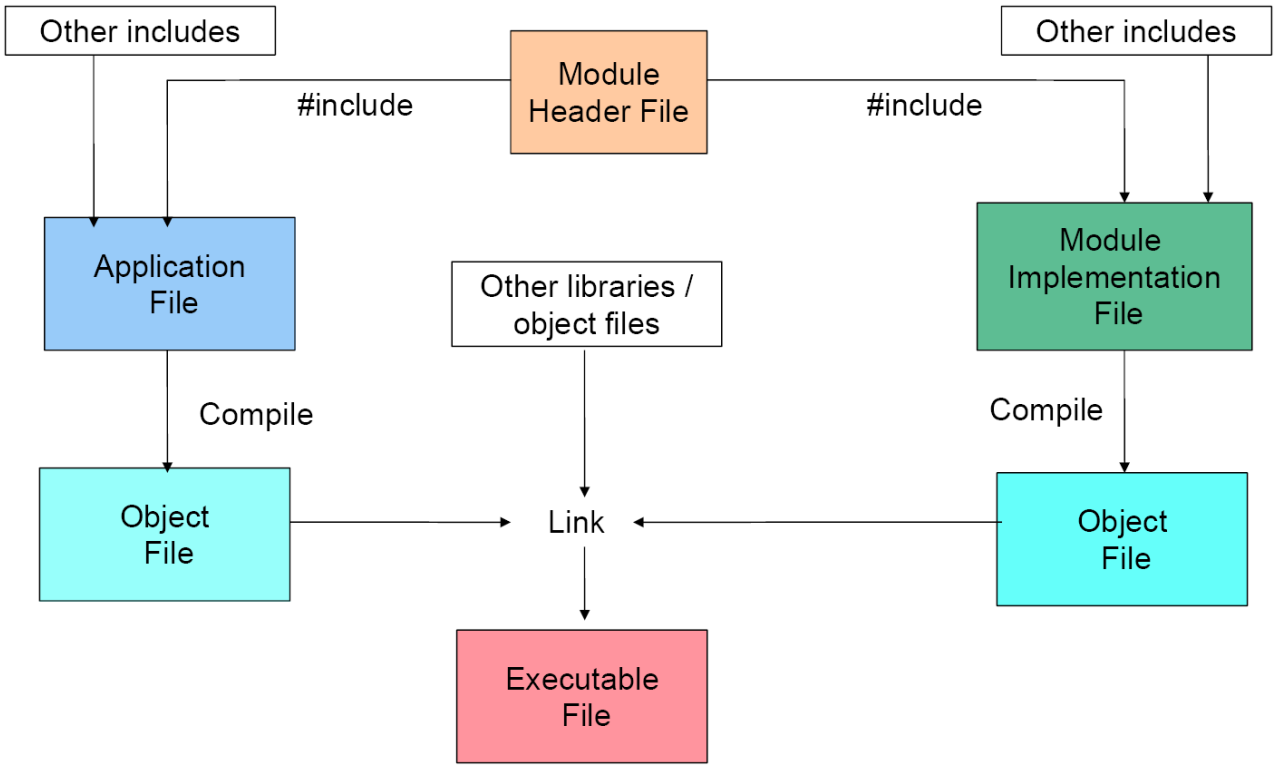
Contents:

- Function body for the main() function
- Acts as client for the different modules

Example: app.c

```
#include<stdio.h>
#include "HelloWorld.h"
int main(){
    iMyGlobalVar = 10;
    printf("%d\n", iMyGlobalVar);
    printHelloWorld();
    printf("%d\n", iMyGlobalVar);
    return 0;
}
```

Associativity between different components



Compiling a simple multi-file program

Syntax:

```
gcc <file1.c> <file2.c> ... -o filename
```

Compiling a simple multi-file program

Syntax:

```
gcc <file1.c> <file2.c> ... -o filename
```

```
user@ws$ gcc HelloWorld.c app.c -o my_app
user@sw$ ./my_app
10
Hello World
20
user@ws$
```

Compiling a simple multi-file program

Syntax:

```
gcc <file1.c> <file2.c> ... -o filename
```

```
user@ws$ gcc HelloWorld.c app.c -o my_app
user@sw$ ./my_app
10
Hello World
20
user@ws$
```

Note: Source files are directly converted into executables.

Compiling a simple multi-file program

Syntax:

```
gcc -c <filename(s).c>
gcc <filename1.o> <filename2.o> [-o output]
```

Compiling a simple multi-file program

Syntax:

```
gcc -c <filename(s).c>
gcc <filename1.o> <filename2.o> [-o output]
```

```
user@ws$ gcc -c HelloWorld.c
user@ws$ gcc -c app.c
user@ws$ gcc HelloWorld.o app.o -o my_app
user@sw$ ./my_app
10
Hello World
20
user@ws$
```

Compiling a simple multi-file program

Syntax:

```
gcc -c <filename(s).c>
gcc <filename1.o> <filename2.o> [-o output]
```

```
user@ws$ gcc -c HelloWorld.c
user@ws$ gcc -c app.c
user@ws$ gcc HelloWorld.o app.o -o my_app
user@sw$ ./my_app
10
Hello World
20
user@ws$
```

Note: Source files are compiled into object files and multiple object files are linked to executables.

Problems – Day 4

1. Suppose you are playing a game in turn with the computer. Total 11 sticks are to be picked up but no more than 3 can be picked up at a time. Whoever picks the last one looses the game. Write a program to let the following happen.
   1. The computer wins if it has the first turn.
   2. The computer wins optimally irrespective of the turn.
2. Without using the getpass() function, write a program to take a password from the user and verify its strength. Mask the password text with character '?'.
   - If the counts of lowercase alphabets, uppercase alphabets, digits, and special characters contained in this is a prime number, then return STRONG.
   - Otherwise, return WEAK.
3. Consider that you have a pair of integers larger than the capacity of long long int. How will you add them? If the integers are larger than the capacity of primary memory of your machine then what to do?

Problems – Day 4

4. Given two files, write a program to find the frequency of each word present in one file in the other and vice versa. Print those words by the decreasing order of frequency.
5. Count the number of HTML tags used in the course webpage of DFS Lab.
6. Write a header file for the ease of dynamic memory allocation, deallocation and reallocation for one-dimensional and multi-dimensional arrays. Use it to write a program for swapping the contents of two files without using any additional file.