Computing Laboratory

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

Malay Bhattacharyya

Assistant Professor

Machine Intelligence Unit
Indian Statistical Institute, Kolkata

November, 2021

The getchar()

Taking a single character input from the user:

```
char c; // int c also works (use carefully)
c = getchar();
```

The getchar()

Taking a single character input from the user:

```
char c; // int c also works (use carefully)
c = getchar();
```

Taking a series of character inputs from the user:

```
char c; // int c also works (use carefully)
while ((c = getchar()) != EOF) {
    ...
}
```

More on getchar()

Reading a number from input using getchar():

```
int x;
char c;
while(isdigit(c = getchar()))
    x = x * 10 + c - '0';
```

Note: `isdigit()` is defined in the header file `ctype.h`.

Enumeration (enum)

We can use enum to declare new enumeration types and define their values.

```
enum MONTH{January, February, March, April, May, June,
  July, August, September, October, November, December};
int main(){
    enum MONTH mi;
    mi = August; // Index of August is assigned to mi
    printf("%d", mi); // Prints 7
    return 0;
}
```

Basics

Definition (What is a preprocessor?)

Before a C program is compiled by a compiler, the source code is processed by a program called preprocessor. This process is called preprocessing.

Definition (What are preprocessor directives?)

The commands used in preprocessor are defined as preprocessor directives and they begin with '#' symbol.

Predefined macros

```
#include<stdio.h>
int main(){
    printf("Current date: %s\n", __DATE__);
    printf("Current time: %s\n", __TIME__);
    printf("\# Lines in the code: %d\n", __LINE__);
    printf("File name: %s\n", __FILE__ );
    printf("C Standard: %d\n", __STDC_VERSION__);
    printf("Compilation was successful: %d\n", __STDC__);
}
```

Predefined macros

```
#include<stdio.h>
int main(){
    printf("Current date: %s\n", __DATE__);
    printf("Current time: %s\n", __TIME__);
    printf("\# Lines in the code: %d\n", __LINE__);
    printf("File name: %s\n", __FILE__ );
    printf("C Standard: %d\n", __STDC_VERSION__);
    printf("Compilation was successful: %d\n", __STDC__);
}
```

Note: The `__STDC_VERSION__` value 199409L signifies the C89 standard, 199901L signifies the C99 standard, and 201112L signifies the C11 standard.

#include

This preprocessor directive is used to include system-defined and user-defined files into the current file. There are two different ways of doing this as shown below.

```
#include <filename>
```

This is used to include system header files. It first searches for a file in a list of directories specified by you, then in a standard list of system directories.

```
#include "filename"
```

This is used include user-defined contents. It first searches for a file in the current directory, then in a standard list of system directories.

#define

There are two different types of macros as listed below.

1. **Object-like macros:**

```
#define e 2.71828
```

2. **Function-like macros:**

```
#define COMPARE(x, y) (((x) > (y)) - ((x) < (y)))
```

#define

What is the output of the following C program?

```
#include<stdio.h>
#define RECIPROCAL(x) 1/x
int main(){
    int x = 1, y = 2, z;
    z = (x+y) * RECIPROCAL(x+y);
    printf("%d", z);
    return 0;
}
```

#define (also known as macros)

Multi-line macros can be accommodated by inserting line separators at the end of each line.

```
#define ASCII(character) {\
    printf("The ASCII value of ");\
    printf("%c", character);\
    printf(" is ");\
    printf("%d", character);\
}
```

#undef

The definition of a macro can be dropped as follows.

```
#undef COMPARE(x, y)
```

#ifdef

We can check whether a macro is already defined as follows.

```
#ifdef COMPARE(x, y)
    // Code segment
#endif
```

Note: If the definition exists, it will execute the code segment.

#ifndef

We can check whether a macro is not yet defined as follows.

```
#ifndef COMPARE(x, y)
    // code segment
#endif
```

Note: If the definition does not exist, it will execute the code segment.

#if and #endif

```
#if <expression>
    // code segment
#endif
```

#else

```
#if <expression>
    // if code segment
#else
    // else code segment
#endif
```

#elif

```
#if expression
    // if code segment
#elif expression
    // elif code segment
#else
    // else code segment
#endif
```

#error

```
#include<stdio.h>
#ifndef __MATH_H
#error Remember to include math.h!!!
#else
void main(){
    float f = 0.7;
    printf("%0.2f", pow(f, 2));
}
#endif
```

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(NULL == (fp = fopen("a.txt", "r")))
    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(NULL == (fp = fopen("a.txt", "r")))
    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 text line by line:

fread(char **lineptr, n, fp): Reads an entire line (`n` elements of data) from `fp`, storing the text (including the newline and a terminating null character) in a `buffer` and storing the buffer address in `*lineptr`.
Returns number of elements read.

Note: Before calling `getline()`, you should place in `*lineptr` the address of a `buffer` (n bytes long), allocated with `malloc()`. If this buffer is long enough to hold the line, `getline()` stores the line in this buffer.

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:

- Pre-processor 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:

- Pre-processor 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
```

Looking into stdio.h (GNU)

```
#ifndef _STDIO_H_
#ifdef __cplusplus
extern "C" {
#endif
#define _STDIO_H_
#define _FSTDIO                    /* ``function stdio'' */
#define __need_size_t
#include <stddef.h>
#define __need___va_list
#include <stdarg.h>
struct __sFile
{
   int unused;
};
typedef struct __sFILE FILE;
```

Link: www.gnu.org/software/m68hc11/examples/stdio_8h-source.html

Looking into stdio.h (GNU)

```
#define __SLBF  0x0001      /* line buffered */
#define __SNBF  0x0002      /* unbuffered */
#define __SRD   0x0004      /* OK to read */
#define __SWR   0x0008      /* OK to write */
/* RD and WR are never simultaneously asserted */
#define __SRW   0x0010      /* open for reading & writing */
#define __SEOF  0x0020      /* found EOF */
#define __SERR  0x0040      /* found error */
#define __SMBF  0x0080      /* _buf is from malloc */
...
```

Link: www.gnu.org/software/m68hc11/examples/stdio_8h-source.html

Looking into stdio.h (GNU)

```
...
int     _EXFUN(printf, (const char *, ...));
int     _EXFUN(scanf, (const char *, ...));
int     _EXFUN(sscanf, (const char *, const char *, ...));
int     _EXFUN(vfprintf, (FILE *, const char *, __VALIST));
int     _EXFUN(vprintf, (const char *, __VALIST));
int     _EXFUN(vsprintf, (char *, const char *, __VALIST));
...
```

Link: www.gnu.org/software/m68hc11/examples/stdio_8h-source.html

A bit of memory organization

Whiling compiling and executing a C program, four different regions of memory are created. These are used as follows:

- A memory region that holds the executable code of the program.
- A memory region where global variables are stored.
- Stack: A memory region that holds the local variables, return addresses of function calls, and arguments to functions while a program is in execution. It also holds the CPU's current state.
- Heap: A memory region that is used by the dynamic memory allocation functions at run time.

Multi-file programs

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

1. Manageability
2. Modularity
3. Re-usability
4. Abstraction

Multi-file programs

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

1. Manageability
2. Modularity
3. Re-usability
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)

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

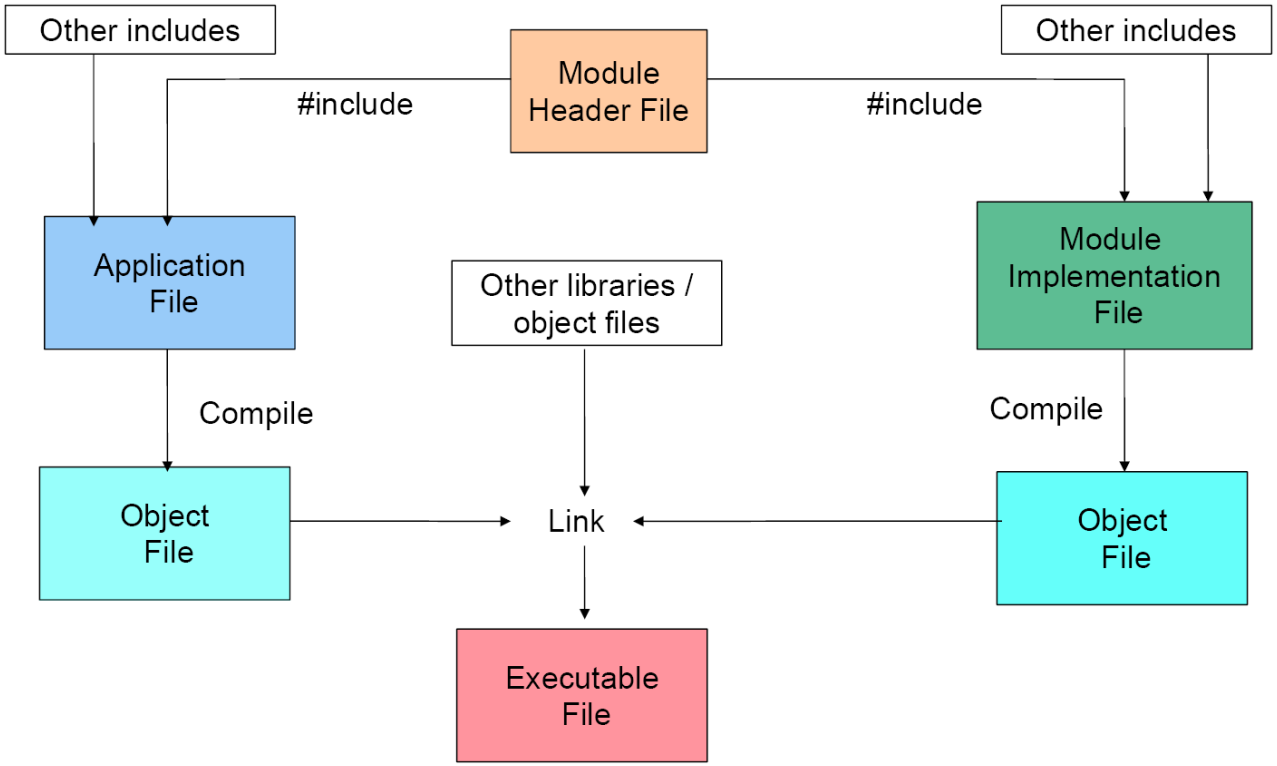
Contents:

- 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 6

1. Let us define the `value` of a string as the sum of ASCII values of its characters. For example, `value` of the string "In2You" is $550 = (73 + 110 + 50 + 89 + 111 + 117)$. Write a program that will take a set of strings as inputs and show them in the ascending order of `value` as the output.
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.

Problems – Day 6

3. Write macros to define the following operations:
   - Evaluates the value of "x implies y"
   - Swaps the values of x and y
   - Finds the minimum of x and y
   - Finds the maximum of x and y
   - Runs an infinite loop over a print statement
   - Finds the least significant bit of x
4. Write a program that will count the number of preprocessor lines used in its source code. Note that, the source code might contain multi-line macros.
5. 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.