Introduction to Programming

C – Parameter Passing, File Handling, Header Files, Multi-file Programs

Malay Bhattacharyya

Assistant Professor

Machine Intelligence Unit
Indian Statistical Institute, Kolkata

January, 2021

Different mechanisms of parameter passing in C

- Call by value
- Call by reference
- Call by name

Call by value

- Arguments are evaluated and then copied into local storage area of the `called` function.
- Changes made to parameters in the called function do not get reflected in the `calling` function.

Call by value

- Arguments are evaluated and then copied into local storage area of the called function.
- Changes made to parameters in the called function do not get reflected in the calling function.

```
void SWAP(int a, int b){
    int t; t = a; a = b; b = t;
}
int main(){
    int m = 10, n = 20;
    printf("(%d, %d)", m, n); // Prints (10, 20)
    SWAP(m, n);
    printf("(%d, %d)", m, n); // Prints (10, 20)
    return 0;
}
```

Call by value

- C uses call by value in general, but arrays are interpreted as pointers.

Call by value

- C uses call by value in general, but arrays are interpreted as pointers.

```
void SWAP_FirstPair(int *A, int i, int j){
    int t; t = A[i]; A[i] = A[j]; A[j] = t;
}
int main() {
    int m[] = {10, 20, 30, 40};
    printf("(%d, %d)", m[0], m[1]); // Prints (10, 20)
    SWAP_FirstPair(m, 0, 1);
    printf("(%d, %d)", m[0], m[1]); // Prints (20, 10)
    return 0;
}
```

Call by reference

- Changes made to parameters in the `called` function get reflected in the `calling` function.
- C simulates call by reference for efficiency.

Call by reference

- Changes made to parameters in the `called` function get reflected in the `calling` function.
- C simulates call by reference for efficiency.

```
void SWAP(int *a, int *b){
    int t; t = *a; *a = *b; *b = t;
}
int main(){
    int m = 10, n = 20;
    printf("(%d, %d)", m, n); // Prints (10, 20)
    SWAP(&m, &n);
    printf("(%d, %d)", m, n); // Prints (20, 10)
    return 0;
}
```

Note: `&m` and `a` point to the same location. Therefore, changing `*(&m)` (= `m`) and `*a` are equivalent.

Call by name

- Parameters are literally replaced in body of the `calling` function by arguments (like string replacement)

Call by name

- Parameters are literally replaced in body of the calling function by arguments (like string replacement)

```
#define SWAP(a, b) { int t; t = a; a = b; b = t;}
int main(){
    int m = 10, n = 20;
    printf("(%d, %d)", m, n); // Prints (10, 20)
    SWAP(m, n);
    printf("(%d, %d)", m, n); // Prints (20, 10)
    return 0;
}
```

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

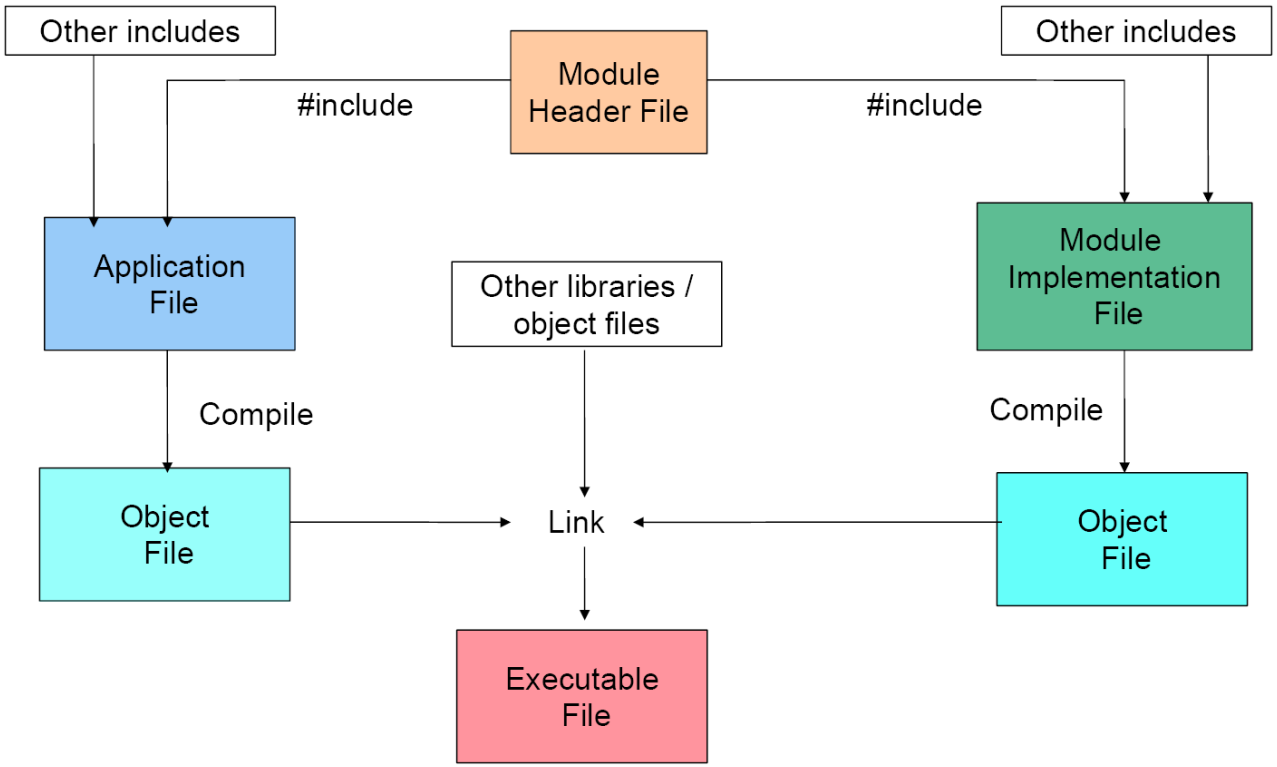
Contents:

- 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.