The C programming language

Version: 2.0. August 2020

Ramasamy Kandasamy

1. Compile and Debug

1.1. GCC

Usage:

```
gcc option hello.c -o hello
gcc hello.c -o hello
gcc -O3 hello.c -o hello
```

Options:

`-o`	Output file name.
`-E`	Output preprocessed but uncompiled C code.
`-c`	Compile and assembly, but do not link.
`-S`	Compile but do not assemble.
`-O<level>`	Optimization level, starts from 0.
`-g`	Creates breakpoints for debugging using GDB.
`-lm`	Missing link. Used to link certain libraries like `math.h` I could not find the reference in `man gcc` or elsewhere.
`-I`	Include additional directories in the search path. Eg: `gcc -I $ HOME /my_dir/my_libraries foo.c`

1.2. GDB

To run: `gdb hello`

`break`	Set break point. Eg: `break function` or `break line number`.
`run`	Run the program. The program stops at every break point.
`next`	Run until next breakpoint.
`print`	Eg: `print i` or `print &i`
`sizeof`	Eg: `print sizeof(i)`
`&i`	Address of i.
`*j`	Content of memory location j.
`ptype`	get type of a variable. Eg: `ptype(i)`
`set var`	Reassign variable. Eg: `set var i = 1`
`disassemble`	Use after break and run. Gives assembly code.

Accessing memory location using x

Usage: `x/nfs`. `nfs` describes the format.

n - Number of units to display.

f - Number format.

s - Size of each unit.

`x`	Hex.
`o`	Octal.
`t`	Binary.
`d`	Decimal.
`u`	Unsigned decimal.
`i`	Instruction.
`c`	Character.
`s`	String.
`b`	byte.
`h`	Halfword.
`w`	Word.
`g`	Giant.

1.3. General

`sizeof`	Compile time unary operator to get object size. Eg: `sizeof object` or `sizeof (type name)`.

2. Pre-processor directives

Pre-processor directive always start with a '`#`'.

- `#include filename`
  Replace with contents of *filename*
  - files with double quotes: `# include "file.h"`
    First pre-processor looks for `file.h` in the same directory as the source file and then in pre-configured list of standard system directories.
  - files with angle bracket: `# include <stdio.h>`
    The pre-processor looks only in the pre-configured list of standard system directories.
  - Additional directories can be include
- `#define NAME replacement text`
  `NAME` is replaced with *replacement text*
- `#define token(arg1,arg2) statement`
  Defining a macro. Eg:
  `#define max(A,B) ((A) > (B) ? (A) : (B))`
- `#undef NAME`
  Nullifies existing definition of `NAME`. Used to ensure a routine is a function.
- `#if`, `#elif`,`#else`, `#endif`
  Eg-1:

```
#if !define(HDR)
#define HDR
/*contents of hrd.h*/
#endif
```

  NOTE: `define()` after `#if` returns `1` if its argument is already defined.
  Here `#define HDR` is first line of `hrd.h` and this file is included only if it was not already included. Eg-2:

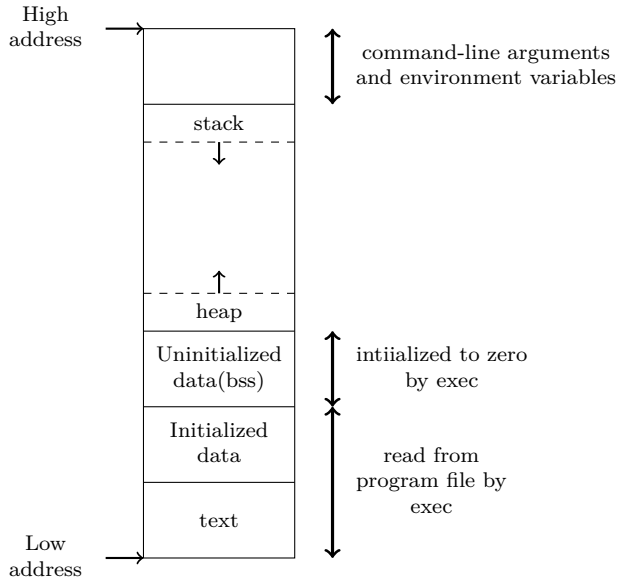
```
#if SYSTEM == SYSV
    #define HDR "sysv.h"
#elif SYSTEM == BSD
    #define HDR "bsd.h"
#elif SYSTEM == MSDOS
    #define HDR "msdos.h"
#else
    #define HDR "default.h"
#endif
```

- `#ifdef` and `#ifndef`
  Test whether a name is already defined.
  Eg-1 in above point can be replaced by: `#ifndef HDR`

```
#define HDR
/*contents of hdr.h*/
#endif
```

3. Memory layout of C program

Adapted from
https://www.geeksforgeeks.org/memory-layout-of-c-program/



In linux the command `size` can be used to see memory allocation of a compiled C program.
Eg:

```
$ size a.out
text data bss dec hex filename
1525 600 8 2133 855
```

Where is heap and stack located physically ?
See:
https://stackoverflow.com/questions/79923/what-and-where-are-the-stack-and-heap

4. Variables and Constants

4.1. Variable types

Type	Location	memory location	Scope
`Global`	Outside functions	data/bss	Global
`Static`	Outside functions	data/bss	Source file
`Static`	Inside a function	data/bss	Local
`Local`	Inside a function	stack	Local
`Register`#	Inside a function*	register	Local

* causes error if declared in global space.
# This is not strict. The compiler might choose not to put the variable in register.

4.2. Variable declaration and initialization

- Eg: `int num;`
  Local or global depending on context.
- Eg: `static int num;`
  Static variable.
- Eg: `const int num;`
  Causes error if `num` is modified.

4.3. Data types

Major variable types

`char`	1 byte.
`int`	4 bytes, but depends on the system.
`float`	4 bytes, but depends on the system.
`double`	8 bytes, but depends on the system.

Modifiers for variable types

`short`	Modifies int as `short int`
`long`	Modifies int as `long int` or simple `long` and as `long long int` or simply `long long` Modifies double as `long double`
`unsigned`	With int and char.
`signed`	With int and char.

4.4. Constants

`1234`	`int`
`1234567890L`	`long int`. When should I use L?
`010`	Octal ≡ 8.
`0x10`	Hexadecimal ≡ 16.
`0b10`	Binary ≡ 2. Binary format is not part of standard C, but is supported by gcc.
'x'	Character constant, has an integer value.
'\ooo'	Specify ASCII code of the character in octal.
'\xhh'	Specify ASCII code of the character in hexadecimal.

Escape sequences:
`\a, \b, \f, \n, \r, \t, \v, \\, \?, \', \"`.
Enumeration constants: List of constant integers.

- Eg: `enum ans {NO, YES};`
  By defaults integers are assigned from 0.
- Eg: `enum days {MON=1,TUE,WED,THU,FRI,SAT,SUN};`
  Here, `MON` is assigned 1, and by default, `TUE` is 2.
- Eg: `enum months {JAN =1, FEB, APR=4,MAY};`
  Here, `MAY` is 5.

5. Arithemetic and logical operation

	Operators	Associativity
1	`() [] -> .`	left to right
2	`! ~ ++ -- + - * & (type) sizeof`	right to left
3	`* / %`(reminder)	left to right
4	`+ -`	left to right
5	`<< >>`	left to right
6	`< <= > >=`	left to right
7	`== !=`	left to right
8	`&`	left to right
9	`^`	left to right
10	`|`	left to right
11	`&&`	left to right
12	`||`	left to right
13	`?:`	left to right
14	`= += -= *= /= %= &= ^= |= <<= >>=`	right to left
15	`,`	left to right

5.1. Explanation for selected operators

- `->` and `.`
  See **Structures**.
- **Casting**
  Changes the type of a variable.
  Eg: `float a = (int) 3/2;// a` is 1.0.
- **sizeof**
  `sizeof num;` // Give size of the variable num.
  `sizeof (int)` // Gives size of int, i.e. 4.
- **Bitwise operators**
  - `>>`
    Eg: `a = b >> 1` //Shift `b` bitwise to the right by 1 bit.
    Fill left most bits with the original left most bit.
  - `<<`
    Eg: `a = b << n` //Shift `b` bitwise to the left by n bits.
    Fill right most bits with zero.

  Others:

`&`	Bitwise AND
`|`	Bitwise inclusive OR
`^`	Bitwise EXOR
`~`	Bitwise NOT

- **Conditional expression `?:`**
  Eg: `char is_pos = (n > 0) ? 'y' : 'n';`
  If `n > 0`, `is_pos` equals `'y'`.
- **Comma operator**
  - `expr1,expr2`
    Use sparingly, Eg: `i++,j--`
  - `x = (Conditional expresion)? expr1: expr2`
    `x =expr1` if `Conditional expression` is true, else `x = expr2`.

6. Control Flow

If-else

```
if (expression)
    statement
else if (expression)
    statement
⋮
else
    statement
```

Switch

```
switch (expression)
{
    case const-expr: statements; break;
    case const-expr: statements; break;
    default: statements; break;
}
```

Without break statement the execution will *fall through* all the cases.

This can be used as:

```
switch (expression)
{

    case 1: case 2: case 3:
        group = 0;
        break;
    case 4: case 5:
        group = 1;
        break;
    default:
        group = -1;
        break;
}
```

While

```
while (expression)
    statement
```

For

```
for (expr1; expr2; expr3)
    statement
```

This for loop is equivalent to:

```
expr1;
while (expr2)
{
    statements;
    expr3;
{
```

Do While

```
do
    statement
while (expression);
```

Break

`break` : Exit the the loop or control.
`Works for while, for, do while and switch`

Continue

`continue`: Continue next iteration.
Works for `while, for, do while`
If `switch` is within a loop and if `continue` is used inside `switch` and if it is executed, then the loop skips to next loop.

Goto and label

Example:

```
if (disaster)
{
    goto error;
}
...
error:
{
    statement
}
```

7. Functions

7.1. int main function: command line arguments

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

argc	Number of arguments including the program name.
argv[0]	Pointer to program name. NOTE: argv[0] represents name of the compiled program and not the source file and how it is called eg: a.out vs ./a.out
argv[i]	Pointer to i^{th} argument.
argv[argc - 1]	Pointer to last argument.

7.2. Function definition

```
type function_name(argument list)
{
    statements
    ...
    return expression
}
```

eg:

```
int foo(int num, int cars[],char *pn)
{
    statements
    ...
    return expression
}
```

7.3. Function declaration

Implicit declaration:
The function is assumed to return `int`. Nothing is assumed about the arguments.
Explicit declaration:
Examples:

```
int foo(int, int [], int *);
int foo(int x, int y[], int *p);
```

Explicit declaration is not necessary if the function is defined before `main()`.

8. Pointers and arrays

8.1. Pointers

Declaration	*type *ptr*; Eg: `int *p;`
Initialization	*type *ptr = val*; Eg: `int *p = 0;` Eg: `int *p = NULL;`// Null pointer. Eg: `int *p = (int *) 100;` NOTE: Casting is required for integer other than zero. Also, the above memory location may not be available.
&	Gives address. Eg: `ptr = &x;`
*	Dereferencing operator. Eg, `*ptr` refers to x.

8.2. Arrays

- **Character arrays**

```
char s[100];
s = "abc"; //ERROR.
*s = "abc"; // OK.
char s[] = {'a','b','c'};
char s[] = "abc";
```

- **Integer, float arrays**

```
int num[100]
num = {1,2,3} // ERROR
*num = {1,2,3} // ERROR
int num[] = {1,2,3}
float num[] = {1.0,2.0,3.0}
```

8.3. Character pointers

Examples:

```
char *pmessage;
pmessage = "Hello, World!\n";
printf("%s",pmessage); # Prints the string
```

`*pmessage` refer to 'H'

8.4. Pointer to pointers

Examples:

`int **p`	`p` is a pointer to a pointer to `int` `*p` is poiner to a pointer to `int`
`char *line[MAXLEN]`	`line` is a pointer to a character array.

8.5. Multi-dimensional arrays

I think, multi-dimensional arrays can be thought of as pointers to pointers etc.
Example:
Given: `int a[2][2] = {{1,2},{3,4}}`
The following are equivalent:

`a[0][0]`	`**a`
`a[1][1]`	`*(*(a + 1) + 1)`

In the above example, `*a` refers to `{1,2}` and `*(a + 1)` refers to `{3,4}`

8.6. Arrays and pointers

The following two are equivalent:

`pa = &a[0];`	`pa = a;`
`a[i]`	`*(a + i)`
`p[i]`	`*(p + i)`
`f(int arr[])`	`f(int *arr)`

Legal	Illegal
`pa++`	`a++`
`pa[-1]`	`a[-1]`

Multi-dimensional arrays vs pointers:

```
int a[2][2] = {{1,2},{3,4}};
```

I think here `a` *is a pointer to an array of two integers.*

```
int* ptr = a;//WARNING: [-Wincompatible-pointer-types]
int** ptr = a;//WARNING: [-Wincompatible-pointer-types]
int (*ptr)[2] = a;//OK.
int* ptr = &a[0][0];//OK.
int* ptr = (int*) a; //OK.
```

The following are equivalent:

a[0][0]	*ptr
a[1][1]	*(ptr + 3)

NOTE: When an array is passed to a function, the function gets only the pointer to the first element as input. Information about size of array is lost.

8.7. Pointer to functions

Eg:

```
int sum_f(int size, int arr[], int (*foo)(int )) // Definition.
{
    int sum = 0;
    for (int i = 0; i < size; i++)
        sum += (*foo)(arr[i]);
}
```

Here `foo` is a pointer to a function and `(*foo)` is the function.

```
sum_f(size, arr, square) // Usage.Sum of squares.
sum_f(size, arr, cube) // Usage.Sum of cubes.
```

NOTE: Function names act as pointers to the function.

9. Structures, unions and typedefs

9.1. Structure

- **Definition:**
```
struct point
{
    int x;
    int y;
};
```
NOTE-1: Structure tag is optional??
NOTE-2: In C a function cannot be a member of a structure.
- **Declaration examples:**
  - `struct {···} a,b,c;`
  - `struct point {···} a,b,c;`
  - `struct point a,b,c`
  - `struct point *p;`
- **Accessing members**
```
a.x = 5;
printf("%d\n",a.x);
```
- **Accessing members with pointer**
`struct point *p = &a;`
The following are equivalent:
  - `p -> x;`
  - `(*p).x;`
  - `a.x;`
- **Arrays of structure:**
```
struct points pts[100];
struct {···} pts[] = {{···},{···},···,{···}};
```
NOTE: In above assignment, in the RHS, elements within the braces could be of different types matching the members of the structure.
In case of simple members, each members need not be enclosed within braces.

9.2. typedef

Used to create new data type names:
Advantages:

- Aesthetics
- Portablility: Eg, `typedef int Length`. In a different machine `Length` could be `char` and only the `typedef needs to be changed`
- Readability, self documentation.

Examples:

- ```
  typedef int Length;
  Length len, maxlen;
  Length *length[];
  typedef struct tnode *Treeptr;
  typedef struct tree Treenode;
  ```
- The above examples could also be implemented by `#define`
The following can only be implemented by `typedef`
```
typedef int (*PFI) (char*, char*);
```

9.3. Union

A union is a variable that may hold objects of different types and sizes.
The syntax is based on structures:
```
union u_tag {
    int ival;
    float fval;
    char *sval;
} u;
```
For example, integer value of u can be accessed as:
`u.ival`

9.4. Bit fields

Eg:
```
struct {
    unsigned int is_keyword : 1;
    unsigned int is_extern : 1;
    unsigned int is_static: 1;
} flags;
```
This defines `flags` that contains three 1-bit fields.
The number following the colon represents the field width.

10. Input output

10.1. File Access

Format:
```
FILE *fp;
FILE *fopen(char *name, char *mode);
int fclose(FILE *fp);
fp = fopen(name, mode);
```
Allowable modes include:

`"r"`	read.
`"w"`	write.
`"a"`	append.
`"b"`	binary files.

Standard I/O

- `stdin`
- `stdout`
- `stderr`

10.2. Charcter I/O

`getchar`	Takes input from standard input. `int getchar()` Equivalent to `getc(stdin)`
`putchar`	Output to standard output. `int putchar(int)` Equivalent to `putc(c), stdout`
`getc`	`int getc(FILE *fp)`, Eg: `getc(stdin)`
`putc`	`int putc(int c, FILE *fp)`
`ungetc`	

10.3. String I/O

`gets`	Reads until `EOF`. `gets` deletes terminal `\n`
`puts`	Writes line to stdout. `puts` adds terminal `\n`
`fgets`	`char *fgets(char *line, int maxline, FILE *fp)`
`fputs`	`int *fputs(char *line, FILE *fp)`

NOTE: Never use `gets`
`gets` does not check for buffer overrun, and keeps reading until it encouters new line or `EOF`. It has been used to break computer security.
```

10.4. Conversion formats for `printf` and `scanf`

Character	Argument type; Printed as
`d,i`	int; decimal number
`o`	int; unsigned octal number (without leading zero)
`x,X`	int: unsigned hexadecimal number (without a leading 0x or 0X), using `abcdef` or `ABCDEF`.
`u`	int; unsigned decimal number.
`c`	int; single character.
`s`	char *; Print string, scan word. NOTE: `scanf` reads only a word and not the entire string.
`f`	double; [-]*m.dddddd*.
`e,E`	double; [-]*m.dddddd*±*xx*, or [-]*m.dddddd*±*xx*.
`g,G`	double; %e or %E if exponent is < -4 or >= precision, else use %f.
`p`	void *; pointer.
`%`	no argument; Print %.

Between % and conversion character there may be in order:

- Minus sign: Left justification.
- Number: Minimum field width.
- Period: Separates field width from precision.
- Number: Precision.
  For float: number of digits after decimal.
  For int: minimum number of digits.
  For string: maximum number of characters to be printed.
- h: for short integer.
  l: for long integer.

Formatting rules

Examples

	Wildcard
%*d	Wild card. Here precision can be specified dynamically. Eg: `printf("%*d", 6, foo);`, this is equivalent to `printf("%6d",foo);`
	Integers
%d	print as decimal integer.
%6d	print as decimal integer, at least 6 characters wide.
	Floating point numbers
%6f	print as floating point.
%.2f	print as floating point, 2 characters after decimal point.
%6.2f	print as floating point, at least 6 characters wide and 2 characters after decimal point.
	String: example `hello, world` (12 chars).
`:%s:`	`:hello,␣world:`
`:%10s:`	`:hello,␣world:`
`:%.10s:`	`:hello,␣wor:`
`:%-10s:`	`:hello,␣world:`
`:%.15s:`	`:hello,␣world:`
`:%-15s:`	`:hello, world␣␣␣:`
`:%15 .10s:`	`:␣␣␣␣␣hello, wor:`
`:%-15 .10s:`	`:hello,␣wor␣␣␣␣␣:`

10.5. Printf and variants

- `int printf(char *format, arg1, arg2, ···);`
- `int sprintf(char *string, char *format arg1, arg2, ···);`
- `int fprintf(FILE *fp, char *format arg1, arg2, ···);`

Format:
`printf("format", var1, ···, varn);`
Example:

```
printf("%s\n", string_var);
printf("%s", "hello, world");
printf("square of n is %d\n", n_square);
```

10.6. Scanf and variants

- `int scanf(char *format, arg1, arg2, ···);`
- `int sscanf(char *string, char *format arg1, arg2, ···);`
- `int fscanf(FILE *fp, char *format arg1, arg2, ···);`

NOTE1: unlike printf, in scanf the variable are pointers.
NOTE2: In `scanf` `%s` reads a word and not the entire string.
Also see: https://stackoverflow.com/a/1248017/5607735

11. Libraries

Usually in Linux, the library header files are stored in `/usr/include`

11.1. Character class tests: `<ctype.h>`

```
isalpha(c)
isupper(c)
islower(c)
isdigit(c)
isalnum(c)
isspace(c)    true for ASCII codes: 9-13 & 32.
toupper(c)
tolower(c)
```

11.2. String functions: `<string.h>`

```
strcat(s,t)       Concatenate t to end of s
strncat(s,t,n)
strcmp(s,t)
strncmp(s,t,n)
strcpy(s,t)       Copy t to s
strncpy(s,t,n)
strlen(s)
strchr(s,c)       Return pointer to first c
strchr(s,c)       Return pointer to last c
```

11.3. Mathematical function: `<math.h>`

```
sin(x)
asin(x)
hsin(x)
exp(x)
log(x)
log10(x)
sqrt(x)
ceil(x)
floor(x)
pow(x,y)    x^y
fabs(x)     Absolute value of x
%           Modulus operator. Eg: int a = 23 % 5;
```

11.4. Utility Functions: `<stdlib.h>`

`system(char *s)`
Run system commands.
Eg: `system("date")`

String to numbers

- `double atof(const char *s)`
- `int atoi(const char *s)`
- `long atol(const char*s)`
- `double strtod(const char *s, char **endp)`
  NOTE on usage of `**endp`, Eg:

```
char *endp;
int a = strtod(s, &endp);
```

- `long strtol(const char *s, char **endp, int base)`
  `base` is used as base.
  If `base` is `0` 8, 10 or 16 is used; Leading `0` indicates octal and `0x` indicates hexadecimal.
- `unsigned long strtoul(const char *s, char **endp, int base)`

Memory management

- `void *calloc(size_t nobj, size_t size)`
  Return pointer to array of `nobj` of size `size`.
  The space is initialized to 0.
- `void *malloc(size_t size)`
  Returns pointer to an object of size `size`.
  The space is uninitialized.
- `void *realloc(void *p, size_t size)`
  Change size of the object pointed to by p to `size`.
  Return pointer to new space.
- `free (void *p)`
  Deallocates space point to by `p`.

Sort and search

- `void *bsearch(const void *key, const void *base, size_t n, size_t size, int (*cmp) (const void *keyal, const void *datum))`
  Searches `base[0]` $\cdots$ `base[n-1]` for `key`.
  Comparison function must return negative if its first argument (search key) is less than its second argument (a table entry) and so on.
- `qsort(void *base, size_t n, size_t size, int (*cmp)(const *void, const *void))`

12. Topics to update

`auto` keyword in C.
Used to declare local variables / functions.
See: https://iq.opengenus.org/auto-in-c/