C Language Tutorial for CS230 students

Created by Chang Hyun Park
Updated by Soojin Hwang

C language

- Developed by Dennis Ritchie while at Bell Labs (First appeared 1972)
  - Created to provide ease of porting assembly coded Unix from machine to machine
  - With C, mostly provide compilation support for new ar chitectures (with minimal architectural specific code)
- Used to reimplement the Unix OS
- Now also used to implement the Linux Kernel
- Still the de facto language for system programming

C is a Compiled Language

- You program in C, and your compiler compiles your C program into machine readable assembly
- Unlike Interpreted languages (Python, Javascript, etc.) you always need to (re)compile your code before execution

gcc

- gcc
  - a C compiler.

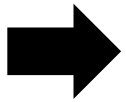
What is a compiler?

→ A language translator.

→ Compiler translates
a high-level language program (*human readable*)
to a low-level language program (*machine readable*)

vim hello.cpp

```
1 #include <iostream>
2
3 //CC510 source code
4 int main()
5 {
6     std::cout << "hello world!" << s
7 }
~
~
~
                                1,1
```



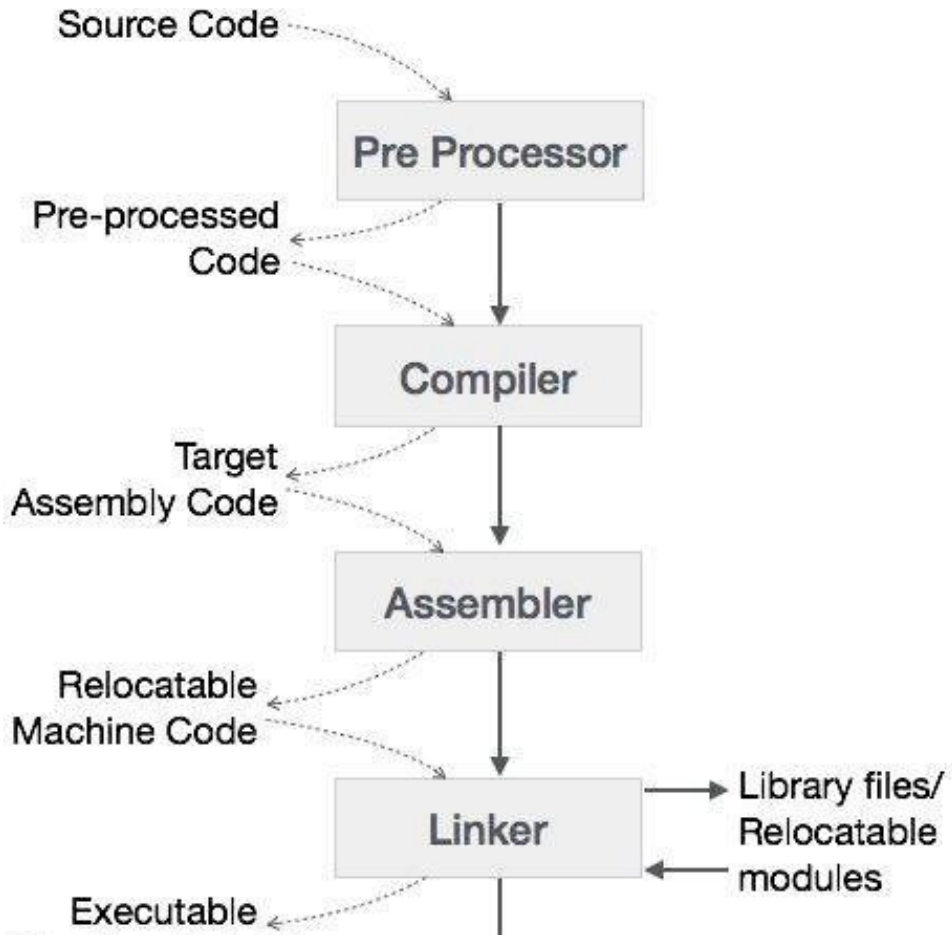
gumdaeng@gumdaeng: ~/CC510

```
0020440 000164 060555 067151 057400 073
 051137 063545 071551
0020460 062564 041562 060554 071563 071
 057400 052137 041515
0020500 042537 042116 057537 057400 052
 057515 062562 064547
0020520 072163 071145 046524 066103 067
 052145 061141 062554
0020540 057400 067151 072151 000000
0020547
gumdaeng@gumdaeng:~/CC510$
```

Human-readable ***Machine-readable***

gcc

Compilation Process



http://www.tutorialspoint.com/compiler_design/compiler_design_overview.htm

gcc

- gcc

```
gumdaeng@gumdaeng:~
 $ cat hello.c
#include <stdio.h>

int main()
{
    printf("hello, world!\n");
}
gumdaeng@gumdaeng:~
 $ gcc hello.c -o hello
gumdaeng@gumdaeng:~
 $ ./hello
hello, world!
```

Simple C Program

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

Include header for I/O (printf)

```
int add(int x, int y){
        return x+y;
}
```

Function

```
int main(void){
        int a, b, sum;

        a=1; b=2;
        // single line comment
        /* multiple line comment */
        sum = add(a, b);
        printf("%d+%d=%d\n", a, b, sum);

        return 0;
}
```

Types of variables

Function call

```
gumdaeng@gumdaeng:~
 $ cat sum.c
#include <stdio.h>

int add(int x, int y){
        return x+y;
}

int main(void){
        int a, b, sum;

        a=1; b=2;
        // single line comment
        /* multiple line comment */
        sum = add(a, b);
        printf("%d+%d=%d\n", a, b, sum);

        return 0;
}
gumdaeng@gumdaeng:~
 $ gcc sum.c -o sum
gumdaeng@gumdaeng:~
 $ ./sum
1+2=3
```

Agenda

- Types
- Flow control
- Functions
- Pointers
- Array
- Structures
- Allocation
- Array (advanced)

C: Types

- Refer to the Wiki page for more details

```
Signed    : Size                  Min                  Max
-------------------------------------------------------------
char      :    1                 -128                  127
short     :    2               -32768                32767
int       :    4          -2147483648           2147483647
long      :    8  -9223372036854775808  9223372036854775807
long long :    8  -9223372036854775808  9223372036854775807

Unsigned  : Size                  Min                  Max
-------------------------------------------------------------
char      :    1                    0                  255
short     :    2                    0                65535
int       :    4                    0           4294967295
long      :    8                    0  18446744073709551615
long long :    8                    0  18446744073709551615

Miscellaneous sizes:
-------------------------------------------------------------
Single precision float:                                    4
Double precision float:                                    8
size_t:                                                    8
ssize_t:                                                   8
```

Type sizes for the x86 64bit architecture. Reference link
Note that there are differences among architectures

C: Types

- Unlike Python where you can insert any value type into a variable, C requires you to declare the type of the variable and only assign the same type.

```
…
int main(void){
        int a, b, sum;
        a=1; b=2;
        // single line comment
        /* multiple line comment */
        sum = add(a, b);
        printf("%d+%d=%d\n", a, b, sum);

        return 0;
}
```

Declare that a, b, sum are integer types

Assign integers 1 and 2 to a and b

C: Typecasting

- Modifying value of one type to a value of another type
- This is similar to typecasting in python
  - Python: int_variable = int(4.0)
  - C: int int_variable = (int)4.0;

```c
#include <stdio.h>

int main() {
    int int_div = 5;
    float float_div = 5.0;

    printf("(int)5/2 = %d\n", int_div/2);
    printf("(float)5/2 = %f\n", float_div/2);
    printf("(typecast float)5/2 = %f\n", (float)int_div/2);
    return 0;
}
```

Cast int_div to float then divide by 2

```
➜ /tmp ./test
(int)5/2 = 2
(float)5/2 = 2.500000
(typecast float)5/2 = 2.500000
```

C: Flow Control: If else statements

- Similar to Python's

```
if [condition]:
    statement
elif [condition]:
    statement
else:
    statement
```

- Strict indenting required

- C:

```
if (condition) {
    statement;
} else if (condition) {
    statement;
else {
    statement;
}
```

- Indenting is not strict
- Use of brace {,} is important

If there are oneliner statements for if/else if/else then braces can be ommitted

```
if (val < max)
   val = max;
//End of if statement
```

C: Flow Control: While statement

- Similar to Python's
  while [condition]:
      statements
      to_keep_on
      looping

- Strict indenting required

- C:
  while (condition) {
      statements;
  }
- Indenting is not strict
- Use of brace {,} is important

If there are oneliner statements for while then braces can be ommitted

```
while (val < max)
   val *= 2;
//End of while statement
```

C: Flow Control: Do-while

- C:
  ```
  do {
      statement;
  } while (condition);
  ```
- Similar to the while statement
- However, when condition is false, do-while will always execute the statement before evaluating condition (thus always execute once)
- While will not execute statement if condition is false

C: Flow Control: For loop

- C:

```
for (i = 0; i < 10; i++) {
    printf("This is the %dth iteration!\n", i);
}
```

- Loop while variable i is between [0, 10)
- for (*init, condition, increment*)
- **Init** is executed before entering for loop
- **Condition** is checked before executing loop
- After loop execution **increment** is executed

C: Flow Control: For loop

```
3 int main() {
4     for (;;) {
5         printf("Hello world\n");
6         sleep(1);
7     }
8     return 0;
9 }
```

Just a for loop
Init: empty (nothing to do)
Condition: empty (implies true)
Increment: empty (nothing to do)

```
3 int main() {
4     int i;
5     int count = 0;
6     for (i = 1; i <= 10; i++) {
7         count += i;
8     }
9     printf("Sum of 1 to 10 is %d\n", count);
10    return 0;
11 }
12
```

Init: assign 1 to i
Condition: while i is between 1 and 10
Increment: increment i by 1

C: Flow Control: Loop helpers

- break: break out of the loop
  - Useful to break out of loops when certain sub-condition is met
  - Sub-condition: a condition that is not the condition to the looping condition
- continue: omit the rest of the statements in the current loop and start next loop
  - Similar to a large if/else statement within the loop

```
for(;;) {
    statements;
    if (condition)
        continue;
    ommitted_statements;
}
```

```
for(;;) {
    statements;
    if (!condition) {
        ommitted_statements;
    }
}
```

C: Functions

- Break down your program code into different group s.
- *return_type* function_name (arguments,…)
- You need to return the value at the end of each non-void function
- Ex) Our ‘main’ function is also a function

```
int main() {
    ...
    return 0;
}
```

Function return type: int
Function name: main
Function arguments: none

```
int main(int argc, char *argv[]) {
    ...
    return 0;
}
```

Function return type: int
Function name: main
Function arguments: argc, argv

C: Functions

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("add(5+7)=%d\n", add(5, 7));
    return 0;
}

int add(int a, int b) {
    return a + b;
}

```

```
➜ /tmp gcc -Wall -o test test.c
test.c: In function ‘main’:
test.c:4:5: warning: implicit declaration of function ‘add’ [-Wimplicit-function-declaration]
     printf("add(5+7)=%d\n", add(5, 7));
     ^
➜ /tmp
```

You must define/declare your function before you call the function!

C: Functions

1. Define before calling

```
 3 int add(int a, int b) {
 4     return a + b;
 5 }
 6
 7 int main(int argc, char *argv[]) {
 8     printf("add(5+7)=%d\n", add(5, 7));
 9     return 0;
10 }
```

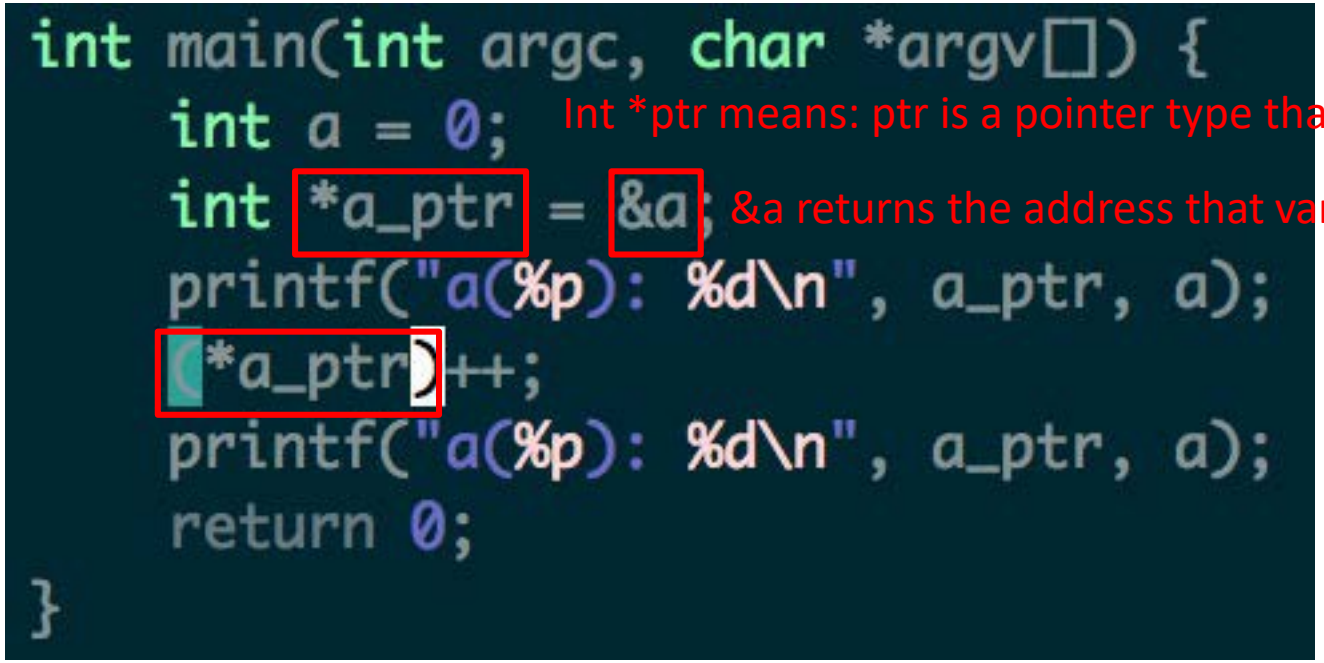
2. Declare before calling

```
 3 int add(int a, int b);
 4
 5 int main(int argc, char *argv[]) {
 6     printf("add(5+7)=%d\n", add(5, 7));
 7     return 0;
 8 }
 9
10 int add(int a, int b) {
11     return a + b;
12 }
13
14
```

When you have multiple Source code files for a program, you will need to declare functions in header files.
Multiple File programs: here

C: Pointers

- Pointers are types of variable that store the memory address of another variable/structure/function



Int *ptr means: ptr is a pointer type that ***points*** to an integer

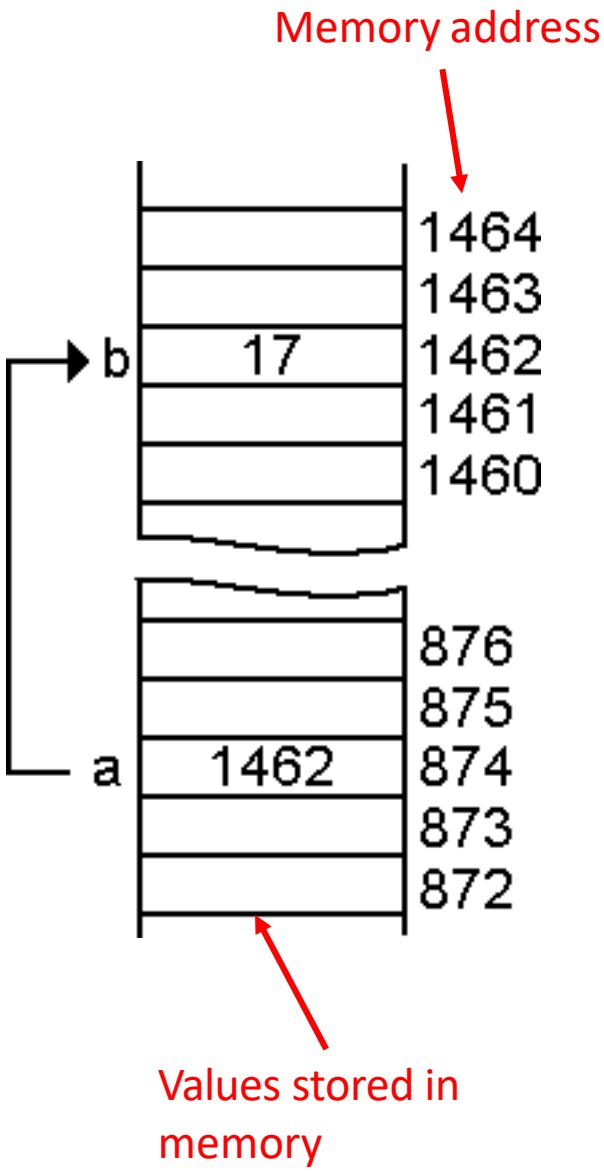
&a returns the address that variable a is stored at

*a_ptr 'dereferences' the pointer, meaning accesses the value stored at the location pointed by a_ptr

C: Pointers

- Example:
- int b = 17;
- int *a = &b; //1462

- Now A points to the address of B
- *a += 2;
- This will dereference a to get B
- Then add 2 to (dereferenced) B
- printf("B: %d\n", 19);



C: Arrays

- Arrays are a consecutive region of memory that holds multiple number of the same data type
- Example 1 : int arr[10];
  - Statically allocates an array of 10 integers
  - ‘arr’ is a pointer variable (int*)
- Example 2 : int arr[3] = {1, 2, 3};
  - Statically allocates an array of 3 integers with initialization
  - Alternative expression : int arr[] = {1, 2, 3};
  - ‘arr’ is a pointer variable (int*)

C: Arrays

- int arr[10];
  //arr == 0x1000
  for (int i = 0; i < 10; i++)
  arr[i] = i;

- Note that you can access the ith element in an array using the [] syntax.
- arr[i] -> accesses the ith value in the array

Address	Value	access
0x1000	0	arr[0]
0x1004	1	arr[1]
0x1008	2	arr[2]
0x100C	3	arr[3]
0x1010	4	arr[4]
0x1014	5	arr[5]
0x1018	6	arr[6]
0x101C	7	arr[7]
0x1020	8	arr[8]
0x1024	9	arr[9]
0x1028	Unallocated space	
0x102C		
0x1030		

C: Structures

- Complex data types that hold multiple variables in a single structure variable
- Definition of a structure

```
struct rectangle {
    float width;
    float height;
};
```

- Using structures (use the “.” operator to access members)

```
struct rectangle rect;
rect.width = 5.0;
rect.height = 2.0;
```

C: Structures

- Structures can be passed as arguments

```
float calculate_size(struct rectangle rect) {
    return rect.width * rect.height;
}
```

- Pointers can also point to structures

```
float calculate_size(struct rectangle *rect) {
    return rect->width * rect->height;
}
```

  - Notice the '->' operator instead of the '.' operator for structure pointers.

C: Structures

- Structure allocation pitfalls. The code below is very different

```
struct rectangle rect;
struct rectangle *rect2;
```

- rect allocates the entire structure in the stack memory.
- rect2 only allocates a pointer to the structure.
  If the structure is 30B, rect2 will only reserve 8B
  (all addresses are 8B in x86 64bit)
- You will likely run into memory corruptions
  if you don't dynamically allocate memory for rect2.

C: Structure & Pointer illustration

```
struct rectangle {
    double width;
    double height;
};
```

```
struct rectangle rect;
struct rectangle *rect2 = &rect;
```

Address	Value	Type
0x1000	Rectangle.width	struct rectangle
0x1002		
0x1004		
0x1006		
0x1008	Rectangle.height	
0x100A		
0x100C		
0x100E		
0x1010	0x1000	Struct rectangle *
0x1012		
0x1014		
0x1016		
0x1018		

C: Structure & Pointer illustration

- Never do this!

```
struct rectangle *rect;
rect->height = 4.0;
rect->weight = 2.0;
```

- Rect points to an arbitrary location
- You'll be writing to the arbitrary location!
- What you should do:

```
struct rectangle *rect;
rect = (struct rectangle*)malloc(sizeof(struct rectangle));
rect->height = 4.0;
rect->weight = 2.0;
```

C: Dynamic memory allocation

- All variable are allocated space by the compiler on the ‘stack’ space
- Stack space is limited in size, and lifetimes are restricted (details should be covered in class)
- Dynamic memory allocation allocates requested size of memory in the ‘heap’ space.
- The heap size is limited by your machine’s available memory (and the address space memory)

C: Dynamic memory allocation

- Malloc header & prototype (function declaration)

```
#include <stdlib.h>

void *malloc(size_t size);
void free(void *ptr);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
```

- If you want to use malloc/free/calloc/realloc you should #include <stdlib.h>
- Malloc receives a size value (how many bytes to allocate) and returns a pointer(of a void type)
- You can typecast the void * to the type that suits you.

C: Dynamic memory allocation

- Dynamic memory allocation examples

```
struct rectangle *rect;
rect = (struct rectangle*)malloc(sizeof(struct rectangle));

int *arr;
arr = (int *)malloc(sizeof(int) * 10);
```

- You can allocate a single variable
  - Note the use of *sizeof(type)* for specifying the size
- You can allocate multiple instances of variables
  - We allocated 10 integers (40B)
  - arr now points to a region of 10 consecutive integers!
  - Also callled arrays!

C: Free memory!

- After using your allocations, free them back

```
/** Use your dynamically allocated space! */

free(rect);
free(arr);
```

- Otherwise you'll get a memory leak
  - You won't be able to use that space during the remainder of the execution of your program
  - If you exit your program it'll be automatically returned to the OS

C: Pitfalls with Freeing memory

- Beware of following cases

```
free(rect);
free(rect); //Error: Double freeing will cause errors!
free(arr+1); //Error: Only free the area that was given by malloc!
```

1. Double free

```
arr: 0x67c030
rect: 0x67c010
*** Error in `./test': double free or corruption (fasttop): 0x000000000067c010 ***
```

2. Invalid memory address

```
➜ /tmp ./test
arr: 0x2388030
rect: 0x2388010
*** Error in `./test': free(): invalid pointer: 0x0000000002388034 ***
```

C: Arrays with Memory Allocation

- int *array2 = (int *)malloc(sizeof(int) * 10);
  - Dynamically allocates an array of 10 integers
- free(array2);
  - Deallocates memory for the array

C: Strings: array of characters

- The C language represents strings as array of characters.
- You can use either char *string; or char string[];
- Strings are packed by “”

vi test.c

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

int main()
{
    char string[] = "Hello world";
    printf("%s(%d)\n", string, strlen(string));
    printf("sizeof(string) = %d\n", sizeof(string));
}
```

```
➜  ~ gcc -o test test.c -w
➜  ~ ./test
Hello world(11)
sizeof(string) = 12
```

C: String representation

- Why did the compiler allocate 12 characters for “Hello world”? Its only 11 letters!

H	e	l	l	o		w	o	r	l	d	\0

- Strings in C are always *terminated by the ’\0’ (null-character)*
- Without the null character no way to know the length of the strings.

C: Strings operations

- strlen: Get the length of the string (without null-character)
- strcpy: copy a string from one location to another (make sure you allocate additional space for the null-character at the destination buffer)
- strcmp: Compare if strings are the same
- strcat: Concatenate two strings (make sure you to strlen for both strings and have enough space for both string length + 1-the null character)
- strstr: Locate a substring within a string