C

Preset:

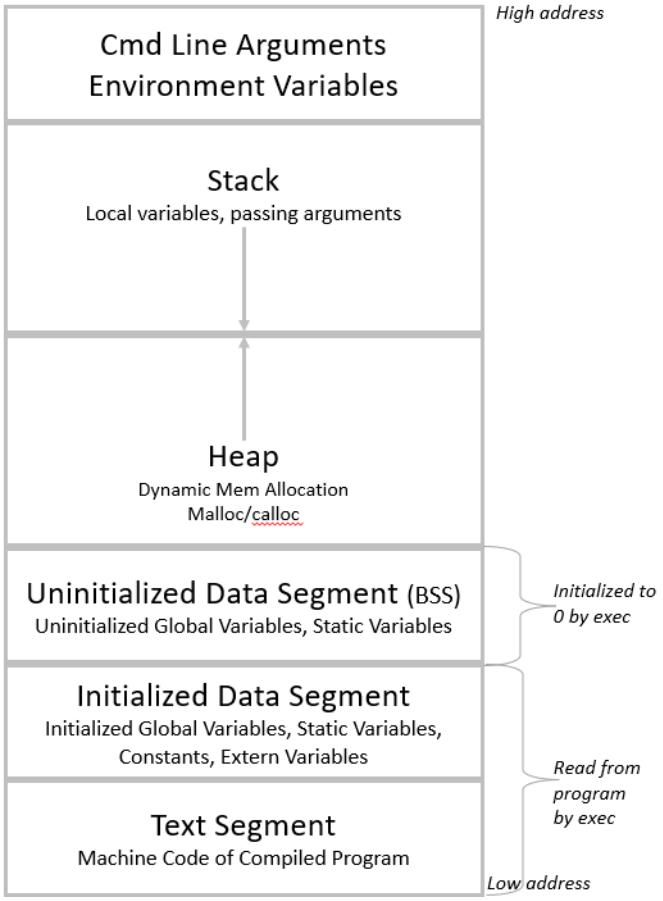
- Invented by Dennis Ritchie, 1972, Bell Labs to write Unix OS
- General Purpose, modular and portable
- Low-level memory access
- C-Standards: C89/90 (1989/90), C99 (1999), C11 (2011), C18(2018)

Structure of C Program:

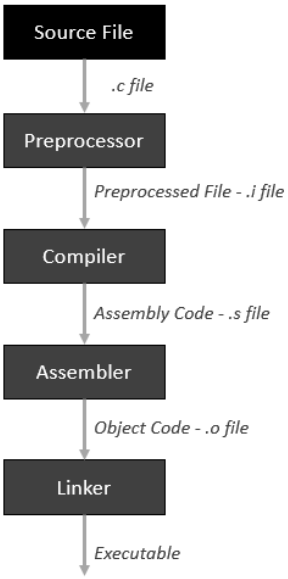
Header	#include <stdio.h>
main()	int main() {
Variable Declaration	int a = 10;
Body	printf("%d", a");
Return	return 0; }

- Header Files inclusion: Contains C function declarations and macro definitions to be shared between source files.
- Main Method: Execution starts from this function.
- Variable Declaration: Variables that are to be used in the function. No variables can be used without being declared.
- Body: Refers to operations that are performed in the functions.
- Return: Depending on return type of the function, the return refers to the returning values from a function

Memory Map:



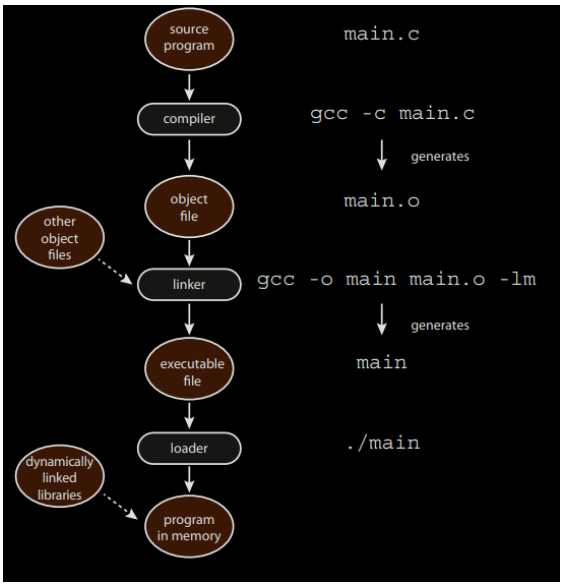
Compilation: Process of converting the source code of C lang into machine code.



1. Source File: Create a c program using an editor and save the file as filename.c
2. Pre-processor: This phase includes the following and produces filename.i

a. Removal of comments
b. Expansion of Macros
c. Expansion of Included files
d. Conditional Compilation

3. Compiling: Compiles filename.i and produces filename.s which includes assembly-level instructions.
4. Assembler: Takes filename.s as output and produces filename.o – object code. Existing code is converted into machine language.
5. Linking: All function calls linked, adds extra code to mark delimiters of each block and forms a executable file.
6. Loading: Loading the process to main memory



$gcc -Wall -save-temps filename.c –o filename // To save all intermediate files during compilation.

Comments and Line Splicing:
Single Line: //
Multi Line: /* */

```c
#include <stdio.h>
int main()
{
    // Line Splicing\
    printf("Hello GFG\n");
    printf("welcome\n");
    /* Below lines will be printed*/ \
    printf("Hello\t");
    printf("World");
    return (0);
}
/* Output
welcome
Hello World
*/
```

Prefer using Multi Line Comments every where

Tokens:

- Keywords
- Identifiers
- Constants/Literals: Variables with fixed values. *const a=19;*
- Strings: Array of chars with a null char at end. *char string[] = "ravikumar";*
- Special Symbols: [](){},:;*=#.~
- Operators:

Assignment	=, +=, -=, *=, /=, %=, &=, |=, ^=, >>=, <<=
Arithmetic	+ , - , * , / , % , ++ , --
Relational	==, !=, >, <, >=, <=
Logical	&&, ||, !
Bitwise	&, |, ^
Ternary	Condition? True : False;
Pointer	&, *
Misc	sizeof(), . (obj.val), &(ptr), *(access)

Keywords: Predefined and reserved words that special meanings to compiler

auto	*break*	*case*	*char*
double	*else*	*enum*	*extern*
int	*long*	*register*	*return*
struct	*switch*	*typedef*	*Union*
const	*continue*	*default*	*do*
float	*for*	*goto*	*if*
short	*signed*	*sizeof*	*static*
unsigned	*void*	*volatile*	*while*

Identifiers - Variables and Constants:

- Named location which has some memory allocated to it.
- Always start with letter/_, only alphanumeric, no whitespace, no keywords.

Types of variables:

- Local Variables: Declared inside a function or block of code
- Global Variables: When we have same name for local and global variable, local variable will be given preference over the global variable by the compiler. For accessing global variable in this case:

```c
#include <stdio.h>

// Global variable x
int x = 50;

int main()
{
    // Local variable x
    int x = 10;
    {
        extern int x;
        printf("Global x: %d\n", x);
    }
    printf("Local x: %d\n", x);
    return 0;
}
/* Output:
    Global x: 50
    Local x: 10 */
```

- Automatic Variables: All local variables are local by default. Scope is local and their life time is till the end of the block
- Extern Variables: To share variables between multiple C files.

```c
//----------myfile.h-----------
  extern int x=10;  //external variable (also global)


// ----------pgm1.c----------
  #include "myfile.h"
  #include <stdio.h>
  void printValue(){
  printf("X: %d", x);
  }
```

- Static Variables: Retains its value between multiple function calls.

```c
#include <stdio.h>
void function()
{
    int x = 20;        // local variable
    static int y = 30; // static variable
    x = x + 10;
    y = y + 10;
    printf("\n%d,%d", x, y);
}
int main()
{
    function();
    function();
    function();
    return 0;
}
/* Output:
30,40
30,50
30,60 */
```

- Register Variables: Stored in CPU register instead of conventional storage (RAM).

```c
register int var = 20;
```

Type Qualifiers:

const	Variables of type const cannot be changed by program. They can only be given an initial value.
volatile	Variable's value may be changed in ways not explicitly specified by the program. Global variable which is passed to the OS's clock routine is a good example. Most compilers do not expect such changes and look for assignment operators for changes.
restrict	Applied to pointer declarations. Object

Storage Class Specifiers:

- **extern:** Used to specify that an object is declared with external linkage elsewhere in the program.
- **static:** These are permanent variables within their own function or file. They maintain their values between function calls. Types: static Local variables and static Global variables
- **register:** Variables declared with register are stored in CPU registers instead of RAM for faster access. If arrays or any data structures are being declared with register keyword will receive a preferential treatment by the compiler as they can't be stored in registers due to their large size

Constants/Literals:

They refer to fixed values that the program may not alter. Types include integers, hexadecimal, octal, string, backslash character constants.

Scope Rules:

Scope	Meaning
File Scope	Starts at beginning of the file and ends with end of the file
Block Scope	Starts at the opening of { and ends at }
Function prototype scope	Identifiers declared in a func prototype; visible within the prototype
Function Scope	Starts at the opening of { of a function and ends at } of a function

Data Types:

Type	Bits	Range
char	8	-127 to 127
unsigned char	8	0 to 255
signed char	8	-127 to 127
int	16/32	-32767 to 32767
unsigned int	16/32	0 to 65535
signed int	16/32	-32767 to 32767
short int	16	-32767 to 32767
unsigned short int	16	0 to 65535
signed short int	16	-32767 to 32767
long int	32	-2^{31}-1 to 2^{31}
long long int	64	-2^{63}-1 to 2^{63}
signed long int	32	-2^{31}-1 to 2^{31}
unsigned long int	32	0 to 2^{31}
unsigned long long int	64	2^{64} - 1
float	32	0 to 2^{31}
double	64	2^{64} - 1
long double	80	2^{80} – 1
bool		stdbool.h
float_complex		complex.h
float_imaginary		complex.h
double_complex		complex.h
double_imaginary		complex.h
long double_complex		complex.h
long double_imaginary		complex.h

INT_MIN and INT_MAX in limits.h give the limits of an integer

Input/Output:

```c
#include <stdio.h>

int main()
{
    int num;
    char str[100];
    char c;

    printf("%s\n", "Ravi");
    scanf("%d", &num);
    c = getchar();
    putchar(c);
    gets(str);
    puts(str);
    fgets(str, 100, stdin);

    return 0;
}
```

- scanf() stops reading characters when it encounters a space
- gets() reads a line of text from stdin(standard input) into the buffer pointed to by str pointer, until either a terminating newline or EOF (end of file) occurs.
- puts() writes the string str with a newline character ('\n') at the end to stdout.
- fgets reads characters and stores them as a C string into str until 100 characters have been read or either a newline or the end-of-file is reached, whichever happens first.
- gets() will keep reading until it encounters a newline character. Unless the buffer is large enough, or the length of the line being read is known ahead of time, gets() can potentially overflow the input buffer and start overwriting memory it is not supposed to, wreaking havoc or opening security vulnerabilities.
- getch(): Reads a char without echo; does not wait for carriage return
- getche(): Reads a char with echo; does not wait for carriage return

Format Specifiers:

int	%d, %i
char	%c
float	%f
double	%lf
short int	%hd
unsigned int	%u
long int	%li
long long int	%lli
unsigned long int	%lu
unsigned long long int	%llu
signed char	%c
unsigned char	%c
long double	%Lf

Control Statements:

Conditional	if-else-if
	switch
Loops	for
	while
	do-while
Jump	return
	goto
	break
	exit()
	continue
Expression	general statement with ;

```c
#include <stdio.h>

int main() {
    // Conditional statements (if-else)
    int num = 10;
    if (num > 0) {
        printf("Condition is true (num is positive)\n");
    } else {
        printf("Condition is false (num is non-positive)\n");
    }

    // Looping statements (for loop)
    for (int i = 1; i <= 5; i++) {
        printf("%d ", i);
    }

    // Looping statements (while loop)
    printf("\nWhile loop:\n");
    int counter = 3;
    while (counter > 0) {
        printf("Counter: %d\n", counter);
        counter--;
    }

    // Jumping statements (break and continue)
    printf("\nBreak and Continue:\n");
    for (int j = 1; j <= 5; j++) {
        if (j == 3) {
            printf("Skipping 3 using 'continue'\n");
            continue;  // Skip the rest of the loop body and continue with the next iteration
        }
        if (j == 5) {
            printf("Breaking at 5 using 'break'\n");
            break;  // Exit the loop when j is 5
        }
        printf("%d ", j);
    }

    // Jumping statements (goto)
    printf("\nGoto statement:\n");
    int k = 0;
    repeat:
        if (k < 3) {
            printf("Repeating (k = %d)\n", k);
            k++;
            goto repeat;
        }
    return 0;
}
/*
Condition is true (num is positive)

For loop:
1 2 3 4 5

While loop:
Counter: 3
Counter: 2
Counter: 1

Break and Continue:
1 2 Skipping 3 using 'continue'
Breaking at 5 using 'break'

Goto statement:
Repeating (k = 0)
Repeating (k = 1)
Repeating (k = 2)
*/
```

Escape Sequences:

\a	Alarm or Beep
\b	Backspace
\f	Form Feed
\n	New Line
\r	Carriage Return
\t	Tab (Horizontal)
\v	Tab (Vertical)
\\	Backslash
\’	Single Quote
\”	Double Quote
\?	Question Mark

\ooo	Octal Number
\xhh	Hexadecimal number
\0	Null

Macros and Preprocessors:
All lines that start with **#** are processed by preprocessor; the functionality is processed prior to other statements in the program.

- When we use *include* directive, the contents of included header file are copied to the current file. Angulars **<** and **>** indicate preprocessor to look in the standard folder where all header files are held. Double Quotes **“** and **“** indicate it to look into the current folder.
- When we use *define* for a constant, the preprocessor produces a C program where the defined constant is searched and matching tokens are replaced with the given expression.
- Macros can take full function like args and args are not checked for data type.

```
#define INC(x) x++;
```

- Macro args are not evaluated before macro expansion.

```
#include <stdio.h>
#define MULTIPLY(a, b) a* b
int main()
{
    // The macro is expanded as 2 + 3 * 3 + 5, not as 5*8
    printf("%d", MULTIPLY(2 + 3, 3 + 5));
    return 0;
}
// Output: 16
```

- Tokens passed to macros can be concatenated using **##**

```
#define concat(a,b) a##b
```

- A token passed to macro can be converted to a string literal by using # before it

```
#define conv_str(a) #a
```

- The macros can be written in multiple lines:

```
#define PRINT(i, limit)       \
    while (i < limit) {       \
        printf("Ravi ");      \
        i++;                  \
    }
```

- It’s better to avoid macros with arguments and inline functions should be preferred as they have type checking.
- Preprocessors also support if-else directives: #if, #else, #endif, #ifdef, #ifndef, #undef
- Standard Macros:

```
#include <stdio.h>

int main() {
    printf("File: %s\n", __FILE__);
    printf("Function: %s\n", __FUNCTION__);
    printf("Line: %d\n", __LINE__);
    printf("Date: %s\n", __DATE__);
    printf("Time: %s\n", __TIME__);
    return 0;
}
/* Output:
File: test.c
Function: main
Line: 6
Date: Feb  6 2024
Time: 11:53:41
*/
```

void main() or main(): In C, void main() has no defined(legit) usage, and it can sometimes throw garbage results or an error. However, main() is used to denote the main function which takes no arguments and returns an integer data type. To summarize the above, it is never a good idea to use void main() or simply, main() as it doesn’t confirm standards. It may be allowed by some compilers though.

int main() and int main(void): In C, if a function signature doesn’t specify any argument, it means that the function can be called with any number of parameters or without any parameters.

Command Line Arguments: A command line argument is the info that follows the program's name on the command line of the OS. Two built-in arguments: - argc holds the number of arguments on the command line and is an integer. - argv is pointer to an array of character pointers. Each element in this array points to a command line argument.
int main(int argc, char *argv[])

Call by Value: This method of passing args to a subroutine copies the values of args into the formal parameters of the sub routine. Changes made to the parameter have no effect on the arg.
Call by Reference: This method of passing args to a subroutine copies the addresses of args into parameters. Inside the subroutine, the address is used to access the actual arg. Changes made to the parameter affect the arg.

Program Error Signals:

SIGFPE	Arithmetic Error	Division by zero, Floating point error etc.
SIGILL	Illegal Instruction: Instruction with no privilege to get executed, got executed.	Stack overflow, Object file corrupted
SIGSEGV	Segmentation fault: Process trying to access a mem location not allocated to it.	De-referencing a wild pointer, Programs gets far from its mem space
SIGBUS	Bus error. Invalid memory is accessed (not existing mem accessed)	De-referencing memory location out of mem space
SIGABRT	Internal Error	General used in assert function
SIGSYS	System Call error	Invalid arg passed to a sys call
SIGTRAP	Exception Occurred, debugger to be informed	Variable changes its value.

Pointers:

- A Pointer is a variable that holds a memory address of another object in memory.
- Pointer Arithmetic: Incrementing a pointer makes it point to a memory location which equals current memory location + sizeof(Pointer data type).
- Function Pointers: Pointers having memory address of where the function begins

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

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

int main() {
    // Declaration and Initialization of Pointers
    int x = 5;
    int *ptrX = &x;

    // Arithmetic operations on pointers
    printf("Value at ptrX: %d\n", *ptrX);
    printf("Address of x: %p\n", &x);
    printf("Address stored in ptrX: %p\n", ptrX);

    // Incrementing and Decrementing pointers
    printf("Incrementing ptrX: %p\n", ++ptrX);
    printf("Decrementing ptrX: %p\n", --ptrX);

    // Function pointers
    int (*addPtr)(int, int);  // Declare a function pointer
    addPtr = &add; // Assign the address of the add function

    // Using function pointer to call the add function
    int result = addPtr(3, 4);
    printf("Result of add function: %d\n", result);

     // Dangling pointer
    int *danglingPtr;
    danglingPtr = malloc(sizeof(int));
    free(danglingPtr);
    danglingPtr = NULL; // Error

    return 0;
}
/*
Output:
Value at ptrX: 5
Address of x: 0x7fffcb451738
Address stored in ptrX: 0x7fffcb451738
Incrementing ptrX: 0x7fffcb45173c
Decrementing ptrX: 0x7fffcb451738
Result of add function: 7
*/
```

Dynamic Allocation:

• malloc(): Allocates the required memory space during execution time. Returns address of the first byte of allocated space/NULL if the memory allocation fails

Syntax: ptr = (datatype *) malloc (size); //stdlib.h

• calloc(): Allocates required memory size during execution time and initializes memory with 0s. Returns address of the first byte of allocated space/NULL if the memory allocation fails.

Syntax: ptr = (datatype *) calloc (n, size); //stdlib.h; n= no. of blocks

• realloc(): To reallocate the size of memory allocated by malloc() or calloc()

Syntax: ptr = (datatype *) realloc (ptr, size); //stdlib.h

• free(): De-allocates the memory allocated by malloc() or calloc()

free(ptr);

Arrays:

Single Dimension Arrays: type var_name[size];

EX: double balance[100];

Two Dimensional Arrays:

Ex: int d[10][20]; // first - rows, second, columns

Multidimensional Arrays:

Syntax: type name[size 1][size 2]….[size n];

Ex: int m[4][3][4][5];