C programming language

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

↳ to include library of basic i/o functions
needs to always be declared

```
main ( ) {
    code;
}
```

→ main function whose code is run on compilation

```
printf ("sentence");
```
→ prints a sentence

```
printf ("%d", i);
```
prints a variable

```
scanf ("%d", &n);
```
→ address operator to access address
→ input a value

- **data types:**

1) int	(integer)	3) double	(floating num with more precision)
2) float	(floating num)	4) char	(characters)

- %d → format specifiers tell which data type is present

- format specifiers:

1)	%d	integer
2)	%hu	unsigned short int
3)	%u	unsigned int
4)	%hd	short int
5)	%ld	long int
6)	%lu	unsigned long int
7)	%lld	long long int
8)	%llu	unsigned long long int
9)	%c	char
10)	%f	float
11)	%lf	double
12)	%LF	long double
13)	%s	string
14)	%p	pointer

→ range of values they can display varies

camel case sensitive

↑ → variable declaration

- data_type var_name = value;

  eg int n = 5;

- escape sequences:

1) \a → generates a bell sound in C
2) \b → moves cursor back (backspace)
3) \f → moves cursor to start of next page
4) \n → takes cursor to start of next line
5) \r → moves cursor to start of current line
6) \t → adds space to left of cursor and moves it
7) \v → add vertical space
8) \\ → adds backslash character
9) \' → adds '
10) \" → adds ".
11) \? → adds ?
12) \ooo → represents octal number
13) \xhh → represents hex number
14) \0 → represents NULL character

→

→ sets precision of answer

. %.14f

- Arithmetic operators:

1) + addition
2) - subtraction
3) * multiplication
4) / division
5) % Modulus
6) ++ increment
7) -- decrement

- Comparison operators:

1) == equals to
2) > greater than
3) < less than
4) != not equal to
5) >= greater than equal to
6) <= less than equal to

Logical operators

1) && AND (both true)
2) || OR (either true)
3) ! NOT (true → false, false → true)

* **Bitwise operator:**

1) & Bitwise AND
2) | Bitwise OR
3) ^ Bitwise exclusive OR
4) ~n Bitwise complement -(n+1)
5) n<<n Shift left n x 2^n
6) n>>n shift right n / 2^n

shortcut

. Bodmass rules of preference followed.

⊙ #include<math.h>
↳ library for basic math functions

→

- Conditional statements:

```
if (condition) {
    execute this;
} else {
    execute this;
}
```

→ condition for 2 pathways

```
if (condition) {
    execute this;
} else if (condition) {
    execute this;
}
    .
    .
    .
} else {
    execute this;
}
```

→ to execute multiple condition checks

→

- if (condition) {
    if (condition) {
        execute this;
    }
}

→ nested if condition

- (condition) ? execute this : execute this;

↗ if true ↗ if false

↘ ternary operator (compressed conditional)

- switch (var_to_check) {
    case var_value1:
        execute this;
        break;
    case var_value2:
        execute this;
        break;
    :
    default:
        execute this;
}

→ to check variable values of 1 variable

→ exits the checking process

→ if none are executed this happens

void → means will return no value

- **loops:**

- while (condition) { → execute till condition is true, keep executing this
  execute this;
  }

- do (condition) { → executes at least once then checks condition
  execute this;
  } while (condition);

- for (initialize counter variable; loop condition; update var) {
  execute this;
  }

  ↳ for a set num of repetitions

  eg
  ```
  for (int i = 0; i < 4; i++) {
      printf("hello");
  }
  ```

→

- continue;
  ↳ code skips to next iteration

- break;
  ↳ code exits the loop without completion

- array declaration (1d-array)

- data_type arr_name [arr_size] = {value};
  " " " [row size][col size] = {" "};
  ↳ array declaration for 2d array

e.g. int arr[4] = 0;
int matrix[2][2] = 0;

- multi-dimensional arrays can be allocated too.

- char string[10];
  ↳ to store a string since C doesn't support strong data type.

→ datatype of value it will return

→ parameter declaration

- data_type function_name (data_type para_name, ...) {
  execute this;
  return value;
  ↳ returns value → breaks out of function
  }
  ↳ function declaration
  → to call a function
- function_name (para_value);

- #include <stdlib.h>
  ↳ used for datatype conversion, memory allocation, algorithms etc.

- If function is defined below main function, it won't work so we do prototyping.

data_type function_name (data_type para_name, ...);

main () {

}

function definition later

header

- so by defining string statement beforehand and later defining the function, it still works.

- (! &condition) → negates condition

- structs are data-structure to hold multiple variables of different data-types.
  → kind of composite data-type

- Struct sname {
    data_type var1_name = value;
    // var2_name = value;
    .
    .
    .
    // varn_name = value;
  }

  → can be used to access values from struct.

sname.var1_name = 0

int ptr + 1

↳ this would add 2 instead of 1 to the address in ptr as int is each 2 bytes long. Same for all other data types

include <string.h>

↳ to access some string manipulation functions

- strcpy (string on which to copy, string which will be copied);
  ↳ function to copy one string to another place.
  ↳ same thing
- i = i + 1 ⟶ i++

- printf ("%p", &var_name);
  ↳ to print physical memory address of a variable.

- C needs memory addresses to access values of variables.
  ↳ data type

- a pointer is a memory address that refers to a location in memory

a pointer variable with no value is called a null pointer. int *ptr = NULL;

- &var_name
  - representation of pointer.
- data_type * pvar_name = &var_name;
  - to declare a pointer variable and storing address of a variable in it.
  - it should always be same as the variable whose address or storing
- to de-reference a pointer means to get value from the location stored in pointer.
  - → eg dereferences a pointer
- printf ("%d", *pvar_name);
  - can be anything → can declare infinite times *&*&
- printf ("%d", *&var_name);
  - to directly dereference a variable

→ declares a file
→ creates a pointer to the file
→ opens a file

- FILE *fpointer = fopen ("filename.txt", file

→ declaring a file
/creates

File modes:

r	→	opens file in read mode
w	→	" " " write mode
a	→	" " " append mode
r+	→	" " " both read & write
a+	→	" " " both append & read
w+	→	" " " both write & read

write: overwrites existing file
append: adds to end of file

fopen()	→ opens new/existing file
fclose()	→ closes file
getc()	→ reads a char from file
putc()	→ writes a char in file
fscanf()	→ reads set of data in file
fprintf()	→ writes set of data in file

fgets() → reads a line from a file and stores in string var. stops reading when \n is reached or end of file (EOF) reached.

getw() → reads integer from file
putw() → writes integer to file
fseek() → sets position to desired point → starting
ftell() → gives current position in file
rewind() → sets position to beginning point

- you can specify directory for file, and if you don't, it'll just be stored in the same directory as C source files.

e.g.

FILE *fpointer = fopen("C:/files/password/filename.txt", "w");
→ directory

- fprintf(filepointer, "info to be added");
→ to write to a file
→ to get line from file

- fgets(var to store line, size, filepointer);
→ size of value to read

- 32 bit & 64 bit represents ~~percentage~~ size used to store 1 address

- unary operator only needs 1 operand

- recursion is a programming concept when the function calls itself inside it's definition. This is used as a iterative construct but is slower in execution time as compared to loops.

- A base case stops the recursion
- A recursive case has the header of the function

- a pointer can be used to traverse a array as well using pointer arithmetic
- The name of an array is a pointer itself

/constants

- All variables passed as arguments to a function in C are passed by Value. It means the value of the variable will be stored in another variable in the function and that variable will be used by the code, while the original variable will remain unchanged

- All arrays in a function are passed by refrence which means that the memory address of the array passed as a argument is given which means any changes made to it will change the original value of the variable too.

```
datatype  func_name (int a) {
        if (condition) {
            do this;        → base case
        } else {
            func name (a);  → recursive case
        }
}
```

☆ String.h ; (imp functions)

- strlen(str_var); → gives length of string
- strcat (str_var1, str_var2);
  ↳ concatenates 2 strings
- strcpy(str_var1, str_var2);
  - str_var1 → where to copy string
  - str_var2 → which string to copy
  - returns a pointer to destination string
- strcmp(str1, str2);
  ↳ compares 2 strings lexicographically.
  returns 0 if it's equal and
  <0 or >0 value NOT equal
  0 if it's equal

string in which to search

character to search for

strchr(str, c);

↳ for first occurence of

↳ searches a character in a string

if found, returns pointer to it

or else returns NULL.

strrchr(str, c);

↳ finds last occurence of a character in a string

strstr(str1, sbstr); → searches for a substring in a string

↓ string in which it will search

↓ substring it is searching for

↳ if found, returns a pointer to first character found of the substring else returns a NULL pointer

data type storage limit:

Integer → 4 bytes
float → 4 bytes
double → 8 bytes
char → 1 byte

- in pointer arithmetic, increment or decrement doesn't neccessarily mean adding or subtracting 1 but rather the num of bytes ~~per location~~ allocated to that data type.

Eg:
```
int a = 10;
int *ptr = &a;
ptr = ptr + 1;
```
↳ it will actually add 4 to the address in the ptr as 4 bytes are given for an integer

- Once a pointer variable points to one variable, it won't point to another unless you explicitly define it.

- sizeof (data_type);
  ↳ gives the size of objects.

- *(p+1)
  ↳ will return value at next location
  if no value stored at the location,
  it will return a garbage value.

- The data_type of a pointer variable must be the same as the data-type of the variable it's pointing too.

  ↳ This is because in de referencing this helps. as it looks for the num of bytes according to the defined data-type.

- ∅

- the address the pointer returns is always of the starting byte.

(data_type*) variable_name;

↳ this is typecasting, which changes the data_type of a variable implicitly. E.g

a = (int*) b;

- A pointer can be assigned the value of another pointer but you have to use typecasting. E.g

```
int a = 1025;
int *ptr = &a;
char *p0;
p0 = (char*)p;
```

if pointer is declared as void it will not need type casting, but a void pointer cannot be derefrenced, e.g

```
void *ptr = &a;
printf("%d", *ptr);
```

↳ will give a compilation error

- A pointer to pointer is a pointer, that points to another pointer, that points to a variable with a value. e.g

```
int a = 5;
int *p = &a;
int **p1 = &p;
```

↳ syntax for pointer to pointer

↑pointer to pointer to pointer

- int ***p

- printf ("%d", *(*p));

↳ dereferencing pointer to pointer

- to pass a variable by value to a function you have to pass a pointer to the variable as the argument to the function.

- The name of the array is a pointer itself to the first location/value in the array.
↳ base address

- address of value of array you can get by:
&A[i] or &(A+i)

- printf ("%d", A);

↳ will return address of first element of array

A++; ✗ wrong

*p = A; ✓ write
p++;

- arrays can be passed as arguments of a function. but as reference parameters

- to find size of an array:

sizeof(A); → returns size in bytes

- to find num of elements in array;
→ size of each integer

sizeof(A) / sizeof(A[0]);

→

• strings in C have to be terminated by null character

• char C[20] = "John";

} has to be in 1 line

} this automatically adds null character at end

• null character isn't counted in string length

• char C[5] = {'J', 'O', 'H', 'N', '\0'};

} another method to store strings

• if a pointer variable is pointing to an array, that pointer can be used for array manipulation.

char *C = "Hello";

↳ this will store the string as a constant and it can't be modified.

- for multidimensional arrays can be used with pointers.

```
int B[4][4];
int *p = B;        ⟶ wrong ✗
int *p[1-4] = B;   ⟶ correct ✓
```

as 2d pointers for 2-d arrays are pointers to an array as they are an array of arrays.

- *printf("%d", *(B+i));

↳ will return the sub array at this index.

→

printf ("%d", *(B+i)+j);

↳ will return specific

printf ("%d", *(B+i)+j);

↳ will return a specific value at row i and column j

specific row → number of columns in array

* printf ("%d", *((B + i * cols) + j)); → specific col

↳ to return a specific value

array dimensions

- data_type (*ptr_var_name)[ ]....[ ];

↳ declaration of pointers for an array

E.g. int B[4][3];

int (*p)[4][3] = B;

- To pass multidimensional arrays as argument of a function, do this;

```
int B[3][2][2]
int
void Func(int (*B)[2][2]);
```

Typical memory architecture

→ extra free memory for allocation by user. (used for DMA)

→ holds function calls and their data

→ holds global variables

→ holds instructions

each function space is called a stack frame

(data_type *)

↳ typecasting

- If we ~~call~~ have a bigger function call stack then the reserved memory for the stack, then stack overflow error occurs and the program crashes. (Eg infinite recursion)

- Heap doesn't have a fixed size and memory can be allocated and de-allocated according to the users needs-

→ typecasting is needed as void pointer is returned

- malloc(size);

↳ tells the compiler how much memory to allocate a variable on the heap. It will return a void pointer to the starting address of the memory allocated on the heap.

E.g

```
int a;
int *p;
```

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

↳ This ~~creates~~ reserves space on heap for a

- memory allocated on the heap by malloc() doesn't automatically delete itself and it needs to be cleared.
  ↗ it is done by this
- free (ptr_var);
  ↘ clears memory on heap

- int a[2n];
  int *p;
  p = (int*) malloc(n*sizeof(int));

to store an array on the heap.

- if no memory available on heap, it returns NULL.

→

↗ data_type for size of objects

void* malloc (size_t size) → general syntax

- typecasting is imp for derefrencing the void pointer returned by malloc().
- memory manipulation on DMA always happens through pointers.

size of one element
↗ of a data type

void* calloc (size_t num, size_t size);

↓
num of elements
of a data_type

- in malloc, if no data is given to allocated memory, it stores garbage values. If calloc() is used, if and no data is given, the allocated memory is initialized to zero.

void* realloc (void* ptr, size_t size);

ptr → pointer to starting address of existing block

size → size of new block.

if a section of heap is already allocated with malloc() or calloc(), and you want to change it's size so you use realloc() to allocate memory again.

- A function can return pointers as well. E.g.

```
int *add(int *a, int *b){
    int *c = (int*)malloc(sizeof(int));
    *c = (*a) + (*b);
    return c;
}
```

- In memory, function is a set of instructions stored in a contigous block of memory.

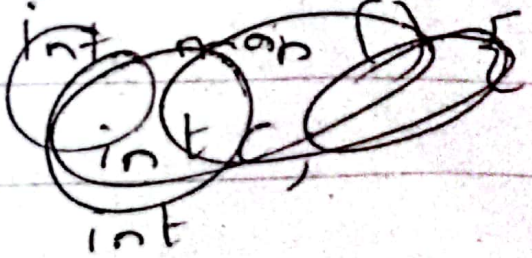
- Entry point of a function is the staring address of the memory block of the function.

- pointers to function can be defined as well. Eg.

```
data_type (*ptr_var_name) (arg1_type, arg2_type...);
ptr_var_name = &func_name;
```

Eg:

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



```
int main () {
    int c;
    int (*p)(int, int);
    p = &add;
    c = (*p)(2, 3); //calling function
    printf("%d", c);
}
```

add ✓

Command-line arguments:

```
main (int argc, int *argv[]);
```

- argc → num of arguments
- argv → array of strings that holds all arguments