1. **Data Types**

// C requires that specify the data type of every variable you create, the first time use that variable:

|  |  |
| --- | --- |
| long (8 byte) / int (4 byte) | store integer numbers |
| double (8 byte) / float (4 byte) | store real numbers (doubles has higher precision) |
| char (1 byte) | store single characters (in ASCII) |
| bool (1 byte) | store logical valuestrue (nonezero) or false (zero) |
| void | represent the absence of value |

// Data type modifier is used to modify the sign or length/range of primitive data types:

Syntax: <modifier> <type>;

|  |  |
| --- | --- |
| short | works with integer that decreases the size of the int to 2 bytes |
| long | works with numbers that increase the original size of an numbers by two times. |
| unsigned | shifts the data type range to the positive part of the whole numbers. |

// Type casting is convert a data type to another data type:

Syntax: (new type) <variable>;

1. **Variables**

// Variable is container for storing data, but actually, it’s representational name that serves as pointer to an object.

* Declaration and Definition:

Syntax: <type> <variable>;

* Initialization:

Syntax: <type> <variable> = <value>;

* **Note**: Variables are declared only, not inititlize value, they are storing values called Garbage Values.

// Constants are read-only variables that values can’t be modified once it is defined:

Syntax: const <variable initialization>;

* **Note**: Always initialize the constant variable at the definition otherwise it will contain garbage value.

// Static variable is variable that declared and memory is allocated only once and exists for the entire program:

Syntax: static <variable initialization>;

// Scope determines in which block of code that variable can be accessed:

* Global variables - are declared outside all functions and are accessible from anywhere in the program.
* Local variables - are declared inside functions and are accessible only inside that block.
* **Note:** When local & global have same name, the compiler priority to local called Variable Shadowing.

1. **Operators**

// Operators are symbols used to perform operations on variables and values:

|  |  |  |
| --- | --- | --- |
| **Arithmetic Operators** | | +, -, \*, /, % |
| **Boolean Expressions** | **Logical Operators** | >, <, >=, <=, ==, != |
| **Relational Operators** | && (and), **||** (or),! (not) |
| **Assgin Operators** | | +=, -=, \*=, /=, %= |
| **Increment/Decrement Operators** | | ++, -- |

1. **Inputs and Outputs**

// printf - print “formatted string” (include text and placeholder for variables: **%s**, **%i** , **%f**, **%c**) to screen:

Syntax: printf(fomat string, arguments);

* **Note**: To control the number of decimal places, use . after the % followed by a number of decimal places.

// scanf - scans input from input buffer with the specified types in fomatted string then stores into **variables:**

Syntax: scanf(fomat string, addresses);

* **Note**: Scanf will only read up to the first space, tab (\t), or newline (\n).

// fgets - read a line from input buffer (include whitespace, newline) until encounter newline and store into string:

Syntax: fgets(<buffer>, <read-maximum-size>, stdin);

* **Note**: Sometimes newline characters are still left in the input buffer and fgets can read them.
* **Solution**: getchar() is used to reads a character from input buffer

1. **Control Flows**

**a. Conditionals**

// If-else statement:

Syntax: if (condition) {…}

else if (condition) {…}

else {…}

// Ternary operator - shorthand syntax if-else statement:

Syntax: <condition> ? <value if true> : <value if false>;

// Switch-case statement - allows us to match a expression’s value against a set of patterns:

Syntax: switch (expression) {

case pattern:

*…*

…

default:

*…*

}

**b. Loops**

// Loops allow your programs to execute lines of code repeatedly:

* while - repeat an unknown number, and possibly not at all:

Syntax: while (condition) {…}

* do-while - repeat an unknown number of times, but at least once:

Syntax: do {…} while (condition);

* for - repeat an known number of times:

Syntax: for(start; condition; step) {…}

* **Note:** In a while loop if the condition is always true then it is called an infinite loop.

// Jump Statements controls the flow of execution of the loop:

|  |  |
| --- | --- |
| break | break out of the loop when encountered |
| continue | skip the current iteration of a loop and continue with the next iteration |
| return | terminate the function and returns a value |

1. **Functions**

// Function is a black box with a set of 0+ inputs and 1 output that helps organize, simplify, and reuse:

* Declaration (Prototype):

Syntax: <return type> <function name>(<parameters>);

* Definition:

Syntax: <return type> <function name>(<parameters>) {…};

* Call:

Syntax: <function name>(<parameters>);

* **Note**: C, C++ both require every program must have a main() function as the entry point:

Syntax: int main() {…}

// The main function receive arguments from the command line that called **Command-Line Arguments:**

Syntax:int main(int argc, char\* argv[]) {…}

* argc - number of arguments passed from the command line, including the program name.
* **argv** - an array containing strings of command line arguments.\
* **Note 1**: Parameters is a variable (placeholder) to receive the argument when you call the function.
* **Note 2**: Arguments is actual value of parameter to pass to the function when calling a function.

1. **Debugging**

// When you're debugging C code, you often pause execution at a breakpoint and then step through the code line by line to observe what it’s doing:

* Step Into - executes the current line of code (at breakpoint) and enters into any function call on that line.
* Step Over - executes the current line of code but does not enter functions; it just treats them as a statement.
* Step Out - runs the rest of the current function and returns to the calling function.

1. **Arrays**

// Array is a data structure that stores values of the same type at contiguous memory locations:

* One-Dimension Array:

|  |  |  |  |
| --- | --- | --- | --- |
| **Array** | **element 1** | **element 2** | **element 3** |
| **Index** | **0** | **1** | **2** |

Access a element of array:

Syntax: **array[index];**

Syntax: <elements type> <array name>[<size>] = {<elements>};

* Two-Dimesion Array:

|  |  |  |  |
| --- | --- | --- | --- |
| **Arrays/Elements** | **0** | **1** | **2** |
| **0** | **arr[0][0]** | **arr[0][2]** | **arr[0][3]** |
| **1** | **arr[1][1]** | **arr[1][2]** | **arr[1][3]** |

Syntax: <elements type> <array name>[<array>][<element>] = { {<elements>}, …};

* Accessing element:

Syntax: array[index];

* **Note:** The compiler automatically determines the size of the array if you leave it blank.

1. **Hexadicimals**

// As a computer scientist, being able to represent data in the same way that computers do is very useful, but the problem is that trying to analyze a large string of 0s and 1s can be really hard.

**➔ Hexadecimal makes this mapping easy because a hexadecimal digit consists of four binary digits that can have 16 different combinations:**

**0 1 2 3 4 5 6 7 8 9 A B C D E F**

1. **Memories**

// Hexadecimal in memory is used to represent the address of a block of memory that starts with 0x prefix.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  | **Integer 4 byte**  **0x123** | | | |
|  |  |  |  | **0x123**  **Base Address** | **0x124** | **0x125** | **0x126** |

// We have 2 operators relate to memory:

* The reference operator **&** get the address of a variable stored in memory (use **%p** in format string)
* The deference operator **\*** access the value of the variable the pointer points to

1. **Pointers**

// Pointers is a speacial variable that is capable of storing base address of variable which it points to.

* Declaration:

Syntax: <type>\* <pointer>;

* Initalization:

Syntax: <type>\* <pointer> = &<variable>;

* **Note:** The asterisk in the declaration used to tells the compiler that the variable is a pointer.

// Common pointers:

* Double Pointer - is a pointer point to another pointer:

Syntax: <type>\*\* <pointer> = &<pointer>;

* Constant Pointer - is a pointer whose address cannot be changed:

Syntax:<type>\*const <pointer>= &<variable>;

* NULL Pointer - doesn't point to any memory location that often represents an invalid memory location.
* **Note**: Segmentation fault is an error that occurs when accessing invalid memory area.

// The relationship between an array and a pointer is that an array’s name acts as a pointer to its first element.

* **Note**: The sizeof(variable) operator returns the size in bytes of data type of a variable.

1. **Swaps**

// Call by value is a method of passing arguments to function where the values of the arguments are copied and passed into the parameters of the function.

// Call by reference is a method of passing arguments to function where the addresses of arguments are passed into the parameters of the function

1. **Strings**

// String in C is stored as an array characters and end with character ( ' \0 ' ) that enclosed within double quotes.

* Define by using Array - allows acess and modify (have all feature of array):

Syntax:char <array>[] = “string”;

+ Define by using Pointer - allows only access since pointers point to read-only memory:

Syntax: char\* <pointer> = “string”;

// String have a set of functions that manipulate strings in library **<string.h>:**

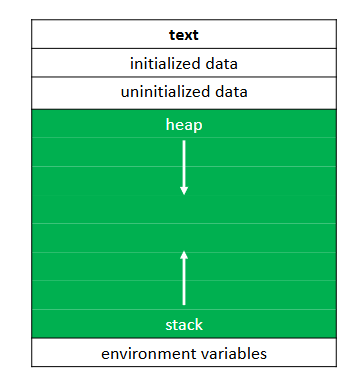
|  |  |
| --- | --- |
| strlen(string) | returns the total number of bytes in a string (not includethe null) |
| strcat(A, B) | concatenates B into A and return A |
| strcpy(A, B) | copies the B into A and return A |
| strcmp(A, B) | compares A with B and return a integer |

1. **Dynamic Memory Allocation**

// Valgrind is a tool that memory related bug where using malloc or you used free not yet after you use malloc:

Syntax: valgrind ./nameprogram

// Compile-time Memory Allocation is the process by which C automatically allocates memory statically (on **stack**) when the program is compiled.



* **Problem:** The program runs, we may need more or less memory than we initially anticipated, and if only use static memory (on **stack**), we cannot change the memory size while the program is running.
* **Solution:** Dynamic Memory Allocation is the process where the user uses pointers to allocate memory dynamically (on **heap**) when the program runs

// C have some function work with dynamic memories:

* **malloc** -allocates a contiguous block of memory with specified size (in byte) but not initialize value and returns pointer to that memory, or NULL:

Syntax: malloc(<size>);

* **calloc** - allocates a block of memory like malloc but initializes all memory areas to 0:

Syntax: calloc(<amount>, <element size>);

* **realloc** - resizes allocated memory and returns a pointe to that resized memory:

Syntax:recalloc(<allocated memory>, <new size>);

* **free** - frees up a block of memory that allocated. If not free, it will be leak memory:

Syntax: free(<allocated memory>);

1. **Call Stacks**

// Call Stack (Stack Memory) is memory area for local variables and function call context.

// When you call a function, the system sets aside space in stack memory for that function, called a stack frame

// These frames are arranged in a stack so they will work on the Last In, First Out principle:

+ When a new function is called, a new frame is **pushed** onto the top of the stack and becomes active frame.

+ When a function finishes, its frame is **popped** off of stack, and the frame below it becomes active.

* **Note:** When too many functions are called, the stack will be full → Crash is called Stack OverFlow

1. **File I/O**

// File Pointer is a tool that allows the program to read data from file and write data to file

// C implicitly defines the concept of file through the FILE data structure:

Syntax: FILE\* <file pointer>;

* fopen - opens a file and returns a file pointer to it:

Syntax: fopen(<file name>, <mode>);

* fclose - closes a file:

Syntax: fclose(<file pointer>);

* fgetc - reads and returns the next character:

Syntax: fgetc(<file pointer>);

* fputc - writes or appends the specified character:

Syntax: fputc(<character>, <file pointer>);

+ fread - reads <qty> units of size <size> from the file pointed to and stores them in memory in a buffer:

Syntax: fread(<buffer>,<size>,<qty>, <file pointer>);

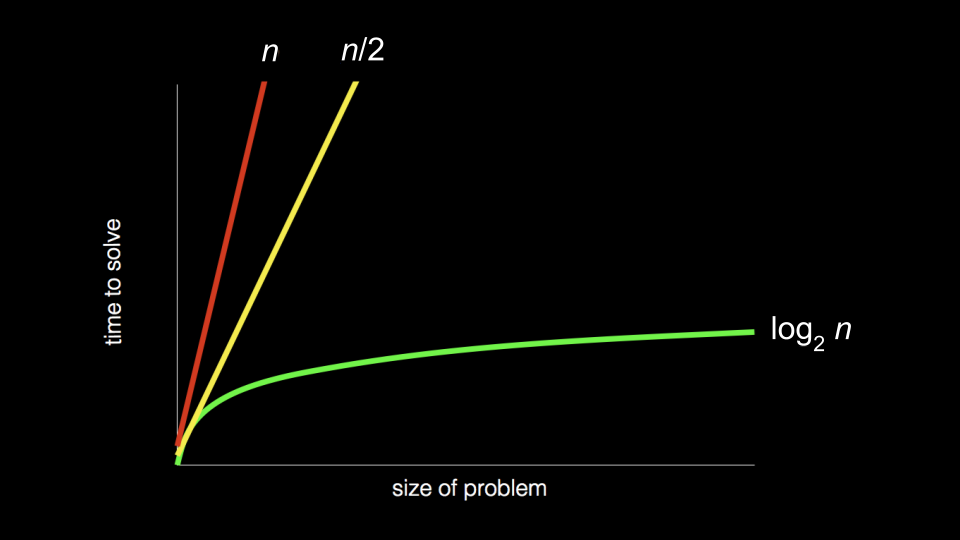
+ fwrite - writes <qty> units of size <size> to the file pointed to by reading them from a <buffer>:

Syntax: fwrite(<buffer>,<size>,<qty>, <file pointer>);

1. **Algorithmic complexity (Big O Notation)**

// Big O notation is a mathematical notation to describe the performance of an algorithm relative to input size

**➔ An efficient algorithm help find the solution we are looking for and turn a slow program into a faster.**



O (upper bound or worst case): O(2n) < O(n²) < O (n) < O (log(n)) < O (1): Fastest

1. **Searches**
2. **Linear Search**

// The idea of the algorithm is to find the smallest unsorted element and add it to the end of the sorted list.

// Pseudo-code:

Repeat, starting at the first element:

+ If the first element is target, stop.

+ Otherwise, move to the next element.

1. **Binary Search**

// The idea of the algorithm is to divide and conquer, reducing the search area by half each time, trying to find a specified element (sorted array).

// Pseudo-code:

Repeat until the (sub)array is of size 0:

+ Calculate the middle point of the current (sub)array

+ If the target is at the middle, stop.

+ Otherwise, if the target < what’s at middle, repeat, changing end point to be just to left of the middle.

+ Otherwise, if the target > what’s at middle, repeat, changing start point to be just to right of middle.

1. **Sorts**
2. **Selection Sort**

// The idea of the algorithm is to find the smallest unsorted element and add it to the end of the sorted list.

// Pseudo-code:

Repeat until no unsorted elements remain:

+ Search the unsorted part of the data to find the smallest value.

+ Swap the smallest value with the first of the unsorted part of the array.

1. **Bubble Sort**

// The idea of the algorithm is to move higher valued elements generally towards the right and lower value elements generally towards the left.

// Pseudo-code:

Set swap counter to a non-zero value

Repeat until the swap counter is 0:

+ Reset swap counter to 0

+ Look at each adjacent pair: If two adjacent elements are not in order, swap them and add one to the swap counter.

1. **Merge Sort**

// The idea of the algorithm is to sort smaller arrays and then combine thoses arrays together (merge) in sorted order

// Pseudo-code:

Recursively sort each half:

Sort the left half of the array (assuming n > 1)

Sort the right half of the array (assuming n > 1)

Merge the two halves together

1. **Recursions**

// Recursion is a process in which a function call itself.

Syntax: return function(parameters) {

// base case - which when triggered will terminate the recursive process.

// recursive case - which is where the recursion will actually occur.

}

1. **Structures**

// The typedef keyword is used to define an alias for the already existing datatype:

Syntax: typedef <old type> <new type>;

// Structures is a user-defined data type that allows multiple different data types to be grouped into a single type

|  |  |  |  |
| --- | --- | --- | --- |
| **Member 1** | **Member 2** | … | … |

Syntax: struct <struct name> {

<type> <member>;

… };

* Initialize a variable:

Syntax: struct <struct name> <variable> = {<values>};

* Access a member of variable:

Syntax: <variable>.<member>;

* **Note**: The arrow operator allows a pointer to a variable of a structure to directly access a member of it:

Syntax:<pointer> ➔ <member>

Actually: (\*pointer).member

|  |
| --- |
| **Member 1**  **Member 2**  **…** |

// Union is user-defined data type similar to structures, unions can contain many different data types. However, members share the same memory location and the size of the union is equal to the size of the largest data type. (That is, if you assign a value to a member of a union, the values of other members will be overwritten, because all members share the same memory area.)

Syntax:union <union name>{

<type> <member>;

… };

// Enumeration is a special type that represents a group of constants:

Syntax:enum <enum name>{

<type> <constant 1>;

… };

* **Note**: Setting the value of an member of enum, the next member’s value will be equal to the previous value increased by 1 .

1. **Data Structures**

// Data structure is the way to organize, store data in memory in a way so that it can be used efficiently.

// Abstract data structures are data structures built using primitive types (integers, …) and providing more complex and specialized operations (arrays, …)

1. **Singly-Linked Lists**

// Singly-linked list is a list made up of nodes, where each node contains a data field, a pointer to the next node:

HEAD

Node 3

NULL

Node 1

Node 2

NULL

|  |  |
| --- | --- |
| **Data** | **Next** |

// Create a node:

Syntax: struct <node name> {

<type> <data>;

…

struct <node name>\* <next>; }

// Operation with Singly-Linked Lists:

+ Traversal: start at head and move through each node until NULL.

+ Insertion: at the beginning, at the end, after a specific node.

+ Deletion: from the beginning, from the end, a specific node.

* **Note 1:** Avoid using the HEAD directly because if you change the HEAD , you will lose the original position of the list.
* **Note 2:** Always update the next part properly before adding/removing nodes to avoid list breakage (losing links between nodes).
* **Note 3:** Multiple operator arrows can be used to access multiple node**s:**

<pointer> ➔ <member> ➔ <member> ➔ …

1. **Doubly-Linked Lists**

// Doubly-linked list is a list made up of nodes, where each node contains a data field, a pointer to the next node and a pointer to the previous node:

HEAD

Node 3

NULL

Node 1

Node 2

NULL

|  |  |  |
| --- | --- | --- |
| **Previous** | **Data** | **Next** |

// Create a node:

Syntax: struct <node name> {

…

struct <node name>\* <next>;

struct <node name>\* <previous>; };

* **Benefits:** Linked lists support extremely efficient insertion and deletion of elements like a resizable array.
* **Trade off:** Unlike arrays, elements in linked lists are not contiguous in memory but can be stored in different locations **➔** Lost the ability to randomly-access list elements.

1. **Stacks**

HEAD

Node 1

// Stacks is implemented as an array or as a singly- inked list and follows the Last In, First Out (LIFO) principle.

// Operation with Stacks:

Node 2

+ **Push**: Add a new element to the top of the stack.

Node 3

+ **Pop**: Remove the most recently-added element from the top of the stack.

NULL

Pop

Node 1

HEAD

NULL

Push

NULL

Node 2

1. **Queues**

**//** Queues is implemented as an array or as a doubly-linked list and follows the First In, First Out (FIFO) principle.

// Operation with Stacks:

+ **Enqueue**: Add a new element to the end of the queue.

NULL

+ **Denqueue**: Remove the first element from the front of the queue.

HEAD

TAIL

Denqueue

Node 1

Enqueue

Node 2

1. **Hash Tables**

// Hash tables combine the random access capabilities of an array with the dynamism of a linked list.

// Structure of Hash Table:

+ Hash function - is a function that convert a data (key) to a nonnegative integer value (hash code)

+ Bucket - is an array used to store datas, in which each element can be a value or a data structure.

Bucket[hashcode=0]

Node

Node

data

…

Node 1

Bucket[hashcode=1]

hash code

hash function

…

// Collision occurs when two keys, when run through the hash function, yield the same hash code, attempting to store data in the same bucket.

+ Linear Probing is a technique that when there is a collision, we will find next empty cell to place the data.

**➔** Problem: Clustering is a phenomenon where elements collide and are inserted close together in the hash table. As these clusters grow larger, adjacent cells will be occupied, increasing the possibility of new collisions.

**➔** Recommened: Chanining is a technique of using each cell of a hash table as the head of a linked list.

* **Benefits:** Hash tables include the advantages of both like searching, adding and removing with O(1).
* **Trade off**: Hash tables are not efficient at sorting or organizing data.

1. **Tries**

// Tries combine structures and pointers together to store sets of character strings.

// Structure of Tries:

+ Each node contain an array pointers to other nodes (indexes represents characters in ASCII).

+ A path from the root to a leaf corresponds to a string.