Study Material for C

About C

C is a programming language developed at AT and T's Bell Laboratories of USA in 1972. It was designed by the man named Dennis Ritchie.

Getting Started - Constant, Variables and Keywords

Constants: A constant is an entity whose values are fixed and may not change during the execution of the program. Constants can be of any of the basic data types like an integer constant, a floating constant, a character constant, or a string literal.

Variables: A variable is an entity whose values **may** change. They are simply names used to refer to some location in memory.

Keywords: Keywords are predefined, reserved words used in programming that have special meanings to the compiler. They are a part of syntax and hence cannot be used as an identifier.

Hierarchy of Operations

Priority	Operators	Description
1st	*** / %**	Multiplication, Division, Modular Division
2nd	**+ -**	Addition, Subtraction
3rd	**=**	Assignment

Example 1: x = 2 * 3 / 4 + 4 / 4 + 8 – 2 + 5 / 8 evaluates to 8.

Reason: As per the priorities the compiler would treat the above problem as,

= ((2 * 3) / 4) + (4 / 4) + 8 – 2 + (5 / 8)

= 1 + 1 + 8 – 2 + 0

= 8

Example 2: Reduce the following algebraic equation as per the compiler's point of view.

$$(m + n)(a + b)$$

Solution: (m + n) * (a + b), **Note:** that here m + n * a + b would be incorrect as due to the priority list, n * a might be evaluated first. Hence to clear this to the compile we use closed brackets [( )].

Example 3: Reduce the following algebraic equation as per the compiler's point of view.

$$\frac{2BY}{d+1} - \frac{x}{3(z+y)}$$

*Solution: 2 * b * y / (d + 1) – x / 3 * (z + y)* ***or*** *(2 * b * y / (d + 1)) – (x / 3 * (z + y))*

Decision Making using if-else

3 types:
1. *if* Statement ***2.*** *if – else* Statement ***3.*** Nested *if – else*

(a)
```
If ( condition )
        do this;
```

(b)
```
if ( condition )
{
        do this;
        and this;
}
```

(c)
```
if ( condition )
        do this;
else
        do this;
```

(d)
```
if ( condition )
{
        do this;
        and this;
}
else
{
        do this;
        and this;
}
```

(e)
```
if ( condition )
{
        if ( condition )
                do this;
        else
                do this;
}
```

(f)
```
if ( condition )
        do this;
else
{
        if ( condition )
                do this;
        else
        {
                do this;
                and this;
        }
}
```

Switch

The control statement that allows us to make a decision from the number of choices is called a switch, or more correctly a switch-case-default.

Syntax:

```
switch (n)
{
    case 1: // code to be executed if n = 1;
        break;
    case 2: // code to be executed if n = 2;
        break;
    default: // code to be executed if n doesn't match any cases
}
```

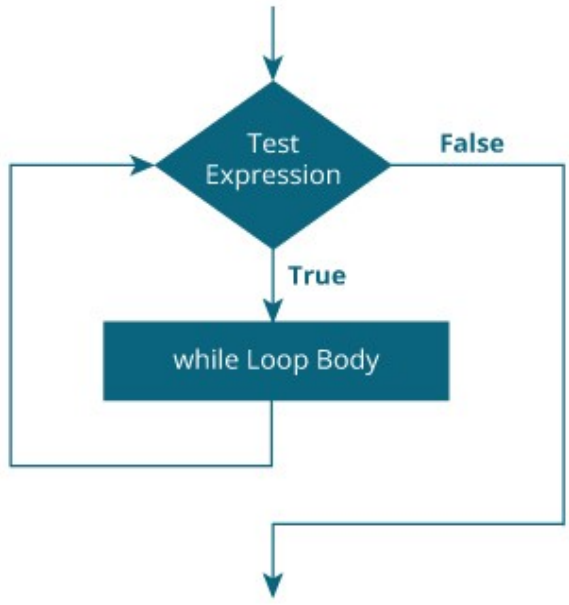
Note: (break;) helps the program to come out of the witch case. For cases when break is not encountered, the program flow may turn to the next case until the break statement is found.

While loop

If the test expression is true (nonzero), codes inside the body of while loop are executed. The condition is evaluated again. The process goes on until the condition is false.

Syntax:

```
while ( condition )
{
    //statements
}
```

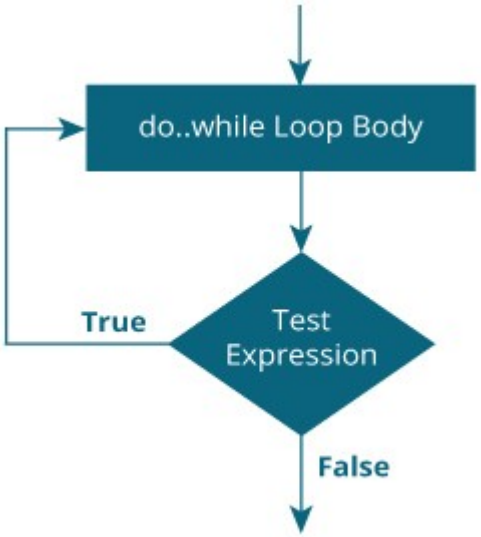


do - while Loop

The do-while loop is similar to the while loop with one important difference. The body of do-while loop is executed once, before checking the test expression. Hence, the do-while loop is executed at least once.

Syntax:

```
do
{
  // codes
}
while ( condition );
```

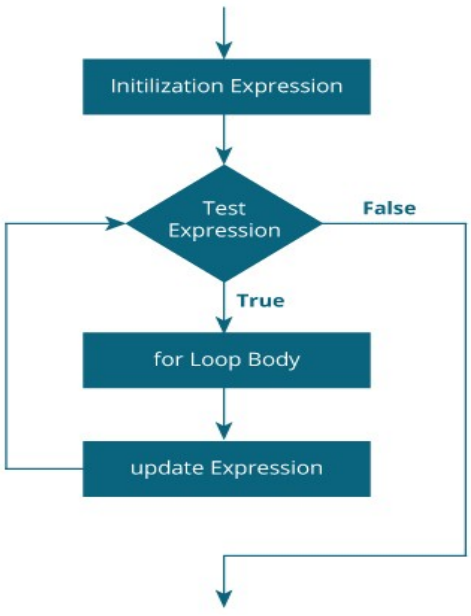


For loop

The initialization statement is executed only once. Then, the test expression is evaluated. If the test expression is false (0), for loop is terminated. But if the test expression is true (nonzero), codes inside the body of for loop is executed and the update expression is updated. This process repeats until the test expression is false.

Syntax:

```
for (initialization-Statement; condition; increment/decrement)
{
        // codes
}
```



Break vs. Continue

Break: When a break statement is encountered inside a loop, the loop is immediately terminated and the program control resumes to the next statement following the loop.

Syntax: break;

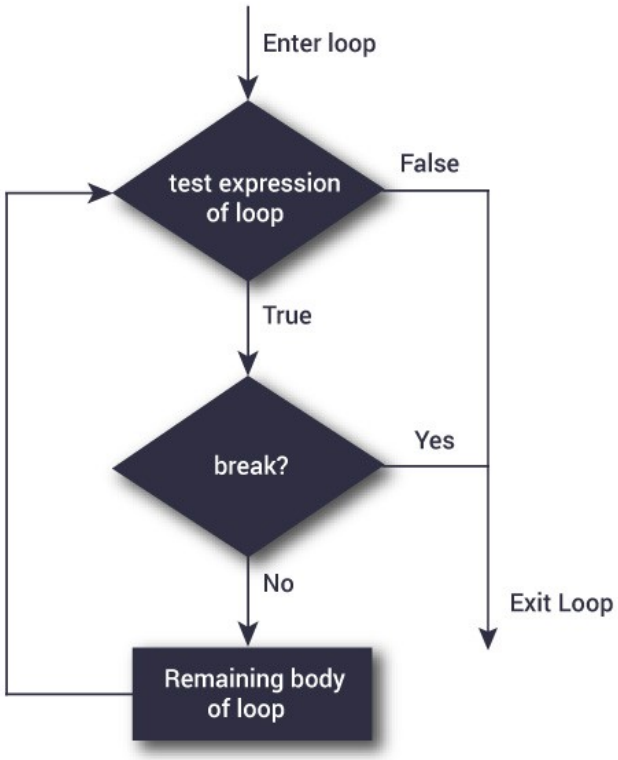
Working:

```
while (testExpression) {
    // codes
    if (condition to break) {
        break;
    }
    // codes
}
```

```
do {
    // codes
    if (condition to break) {
        break;
    }
    // codes
}
while (testExpression);
```

```
for (init; testExpression; update) {
    // codes
    if (condition to break) {
        break;
    }
    // codes
}
```

Flow Chart:



Continue: When a continue statement is encountered inside a loop, it gets the control to the beginning of the loop for the next iteration, skipping the execution of statements inside the body of loop for the current execution.

Syntax: continue;

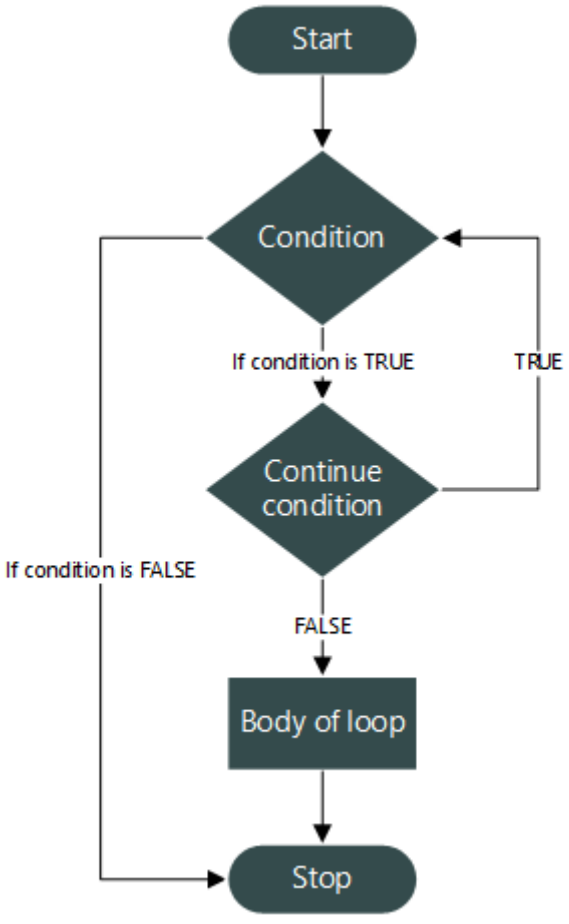
Working:

```
while (testExpression) {
    // codes
    if (testExpression) {
        continue;
    }
    // codes
}
```

```
do {
    // codes
    if (testExpression) {
        continue;
    }
    // codes
} while (testExpression);
```

```
for (init; testExpression; update) {
    // codes
    if (testExpression) {
        continue;
    }
    // codes
}
```

Flow Chart:



C goto Statement

Avoid **goto** keyword! It makes a C programmer's life miserable. It's use is one of the reasons that programs become unreliable, unreadable and hard to debug. The big problem with **goto**s is that when we do use them we can never be sure how we got to a certain point in our code. They obscure the flow of control. So as afar as possible, skip them. You can always get the job done without them. Trust me, with good programming skills **goto** can always be avoided. However for the sake of completion of the Study Material, the following program shows how to use **goto** :

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

int main()
{
        int number;
        printf("Enter 1 or 2: ");
        scanf("%d", &number);

        if(number == 1)
        {
        goto first;
        }

        if(number == 2)
        {
        goto second;
        }

        first:
        printf("This is First, Number 1 was chosen.");

        second:
        printf("This is Second, Number 2 was chosen.");

}
```

Here are two sample runs of the program:
>Enter 1 or 2: 1
>This is First, Number 1 was chosen.

and

>Enter 1 or 2: 2
>This is Second, Number 2 was chosen.

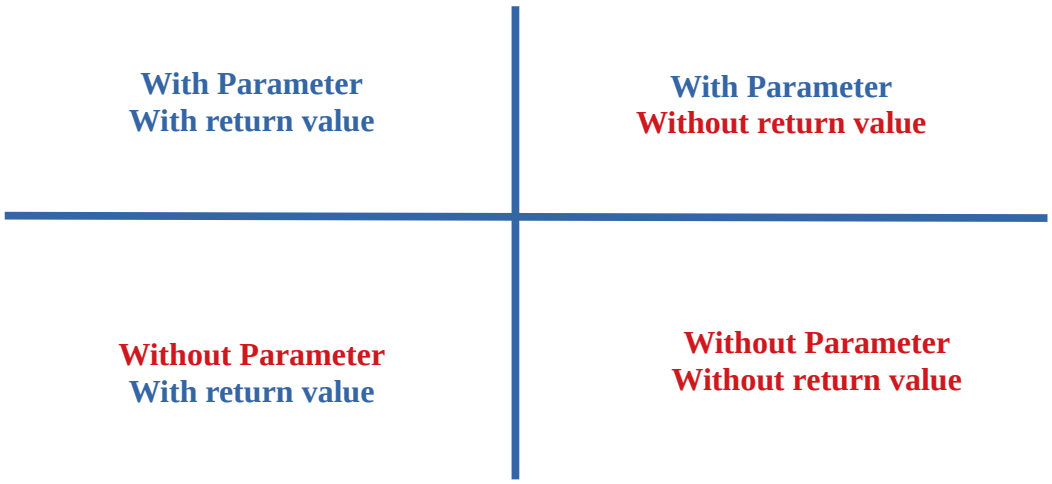
Here, first: and second: are called **labels**. A statement **label** is meaningful only to a **goto** statement.

Functions

A function is a block of code that executed a specific task. They are of 2 types: Standard Library Functions and User Defined Functions.

Standard Library Functions: The standard library functions are built-in functions in C programming to handle tasks such as mathematical computations, I/O processing, string handling etc. These functions are defined in the header file. When you include the header file, these functions are available for use.
For example: printf() and scanf() are the library functions under the library stdio.h.

User Defined Function: Defined by users, could be classified into 4 types:




```
#include<stdio.h>

int add (int, int); //prototype declaration of function

int main()
{
        int a,b,c; //add(a,b); is called function call.
        c = add(a,b);  //The parameters here given are said to be "Actual Parameters"
}

int add(int x, int y)  //The parameters here given are said to be "Formal Parameters"
{
        return x+y;   //Everything written inside the function is "Function Definition"
}
```

Recursion

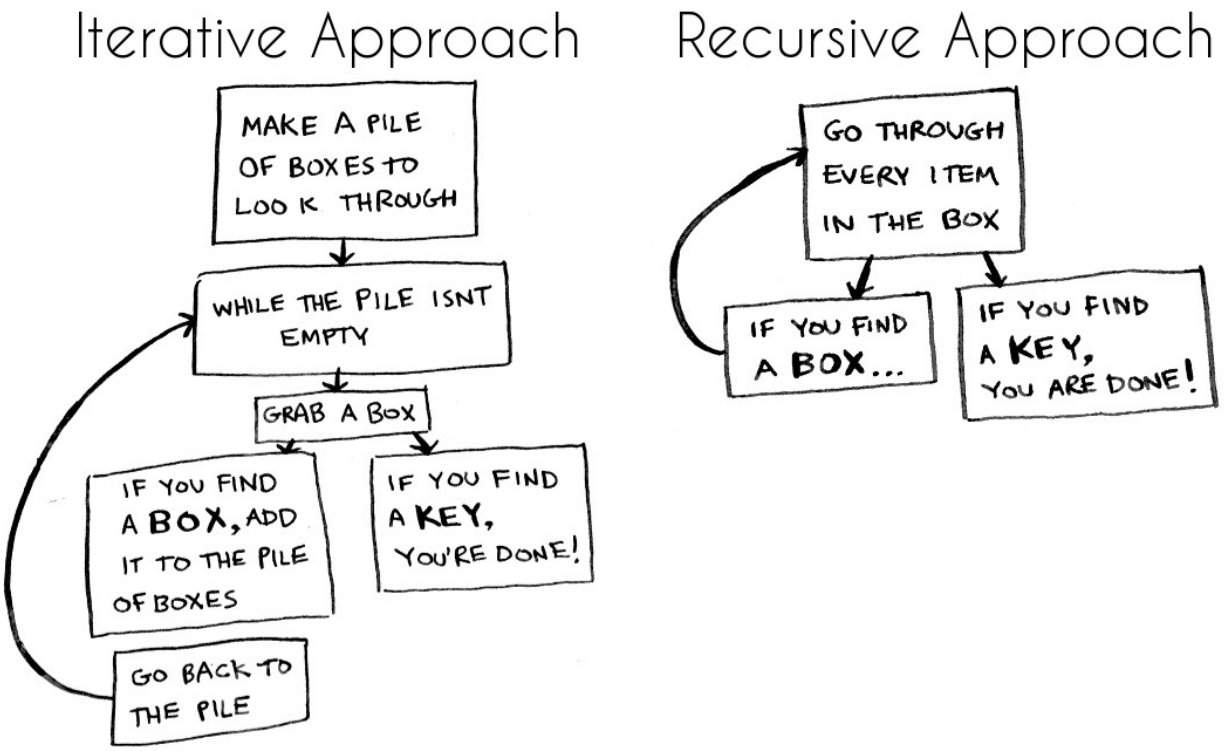
In C, it is possible for the functions to call themselves. A function is called **'recursive'** if a statement within the body of a function calls the same function. **Recursive** is thus the proces of defining something in terms of itself.

Example:

```
#include<stdio.h>

int main()
{
        int number, fact;
        printf("Enter number to get factorial: ")
        scanf("%d", &number);

        fact = factorial(number);
        printf("Factorial Value: %d\n", fact);
        return 0;
}

int factorial(int x)
{
        int f=1;
        if(x==1)
          return 1;
        else
          f = x * factorial(x-1);

        return f;
}
```

Explanation: In the program if the input is **1**, the number gets stored into **fact**. *On making a function call by value*, the value is then transferred to **x** and is then used in the **function factorial**. The **if** statement then checks if the value of **x** is **1** or not, In this case, it evaluate to **true** and hence the program **returns 1** as the factorial of 1 is 1 itself. For a second case if the number entered is **3,** again the function factorial is called by value and the value of the number stored in **fact** is transferred to **x**, which is then used in the function factorial. This time the **if statement** evaluates to **false** as the number is **3** and hence moves on to the next occurring set of statements which is else, here **f** is assigned the value of **x * factorial(x-1).** *This means that the current value is multiplied with value that would be returned if the same function is called with an actual argument of the number (or value of x), decreased by 1.* That means **f = 3 * factorial(2)** and **when factorial(2) will be evaluated f = 2 * factorial(1) .**

Following diagram depicts a very simple explanation of recursion works and why is it better than Iterative Approach:



Few points of summary:

1. A fresh set of variables get created every time a function gets called (normally or recursively).
2. Adding too many functions and calling them frequently may slow down the program execution.

Pointers

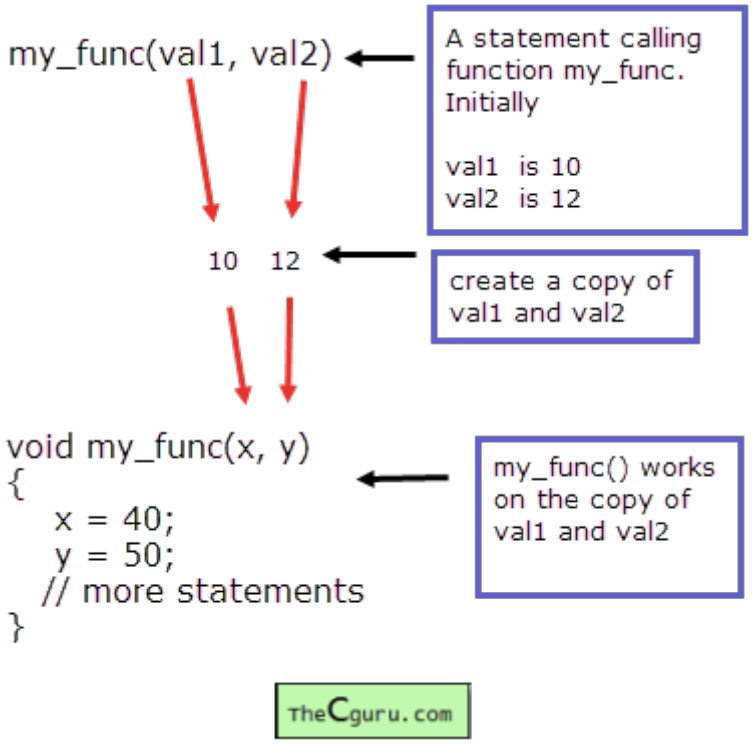
A pointer is a variable whose value is the address of another variable. Like any variable or constant, you must declare a pointer before using it to store any variable address.

Syntax: ***datatype *variableName;*** *// * can also be read as 'value at address'*
/ also & can be read as 'Address of' */*

Call by Value vs. Call by reference

Before diving in, let us make it clear that when values are passed to a function, the values move from the actual arguments or actual parameters to formal arguments or formal parameters.

Call by Value: In call by value a copy of actual arguments is passed to respective formal arguments. This could be better understood from the following diagram:



Call by reference: In Call by Reference the location (address) of actual arguments is passed to formal arguments, hence any change made to formal arguments will also reflect in actual arguments.

This could be seen in the following code block:

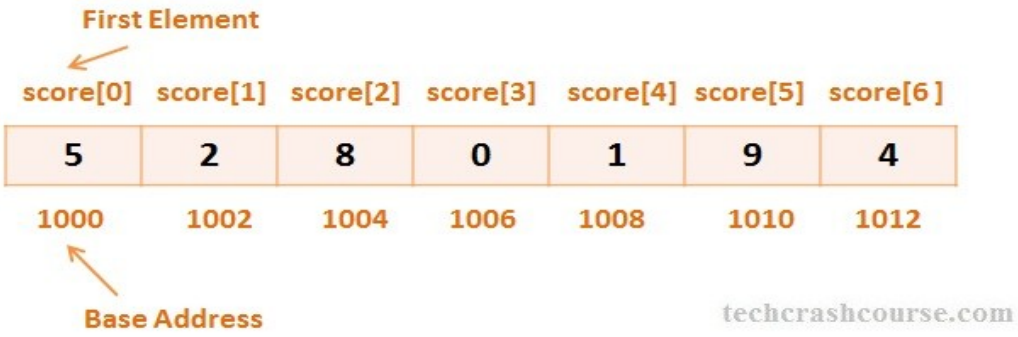
```
#include<stdio.h>

int main()
{
        int a,b,c;
        c = add(&a,&b);  //addresses of a and b are the actual arguments.
}

int add(int *x, int *y)  //we used pointers x and y to store the addresses of a and b.
{
        return x+y;
}
```

Arrays

An array is a collection of variables of same datatype.

Declaration of Arrays: ***datatype variableName [***SizeOfArray***];***

Following is the memory map of Arrays



In the above Memory Map it can be clearly seen that there exist an Array named as **score** of type **int** with size of **7.** The point to be noted here is that the memory addresses are increased by 2 every time when allocated for the next space of array. This is because in a 32 bit OS, the size of int is 2 bytes.

Initialization of an Array: int mark[5] = {19, 10, 8, 17, 9};
Another method to initialize array during declaration: int mark[] = {19, 10, 8, 17, 9};

mark[0]	mark[1]	mark[2]	mark[3]	mark[4]
19	10	8	17	9

Based on how the
Array is declared, they can be of 2 types:
1. One Dimensional Array
2. Multi – Dimensional Array

One Dimensional Array: Conceptually you can think of a one-dimensional array as a row, where elements are stored one after another. These type of arrays are the ones we have discussed so far.

Multi – Dimensional Array: A **multi-dimensional array** is an array that has more than one dimension. It is an array of arrays; an array that has multiple levels. The simplest multi-dimensional array is the **2D array**, or two-dimensional array. It is technically an array of arrays, as you will see in the code. A 2D array is also called a **matrix**, or a table of rows and columns.

2 Dimensional Arrays
Declaration: ***datatype arrayName[rows][columns];***
Following diagram shows a 2 Dimensional Array of Size 4x4 or a Matrix of 4x4.

	Col 1	Col 2	Col 3	Col 4
Row 1	0, 0	0, 1	0, 2	0, 3
Row 2	1, 0	1, 1	1, 2	1, 3
Row 3	2, 0	2, 1	2, 2	2, 3
Row 4	3, 0	3, 1	3, 2	3, 3

Following Code snippet explains how to take inputs for a **1D Array** and **2D Array.** Again you can use the same approach to print output.

```
#include<stdio.h>

int main()
{
	int input;
	int a[5];
	int b[2][2];

	//taking input for 1D Array
	for(int i=0;i<5;i++)
	{
	scanf("%d", &a[i]);
	}

	//taking input of 2D Array
	for(int i=0;i<2;i++)
	{
	for(int j=0;i<2;j++)
		{
			scanf("%d", &b[i][j]);
		}
	}

	//Similarly you can use the same approach for printing output.
}
```

Strings

The way a group of integers can be stored in an Integer Array, similarly a group of characters can be stored in a Character Array. Often these **Character Arrays are termed as Strings.**

gets() and puts()

While taking input, scanf() is not capable of receiving multi-word strings. Therefore names such as John Ramsey would be unacceptable. The way to get around this limitation is by using the functions **gets().** The usage of functions **gets()** and its counterpart **puts()** is described in the code snippet below.

```
#include<stdio.h>

int main()
{
        char name[25];
        printf("Enter your name: ");
        gets(name);   //takes the name as input
        puts(Hello, ); //prints Hello to the Console Window
        puts(name);   //prints the name
        return 0;
}
```

The program and the output are self-explanatory except for the fact that, ***puts()*** *can display only one string at a time.* The *plus point with* ***gets()*** *is that it can take multi-word strings as input.*

Standard C Library String Functions

Function	Use	Short Example
strlen()	Finds length of a string.	Int x = strlen(name);
strlwr()	Converts a string to lower case	strlwr(name);
strupr()	Converts a string to upper case	strupr(name);
strcpy()	Copies a string into another	strcpy(target, source);
strcmp()	Compares two Strings. *(if the strings match, it will return 0)*	Int x = strcmp(name1,name2)
strcat()	Appends First n characters of a string at the end of another.	strcat(target, source);

Key Notes:
1. Being an array, all the characters of a string are stored in contiguous memory locations.
2. Both **printf()** and **puts()** can handle multi-word strings.

Structures and Unions

Structures	Unions
Keyword **struct** defines a structure.	Keyword **union** defines a structure.
Example structure declaration: *struct s_tag* *{* *int rollNumber;* *char name[20] ;* *char c;* *}structureVariable;*	Example union declaration: *union u_tag* *{* *int rollNumber;* *char name[20] ;* *char c;* *}unionVariable;*
Within a structure all members gets memory allocated.	For a union compiler allocates the memory for the largest of all members. The container is big enough to hold the widest member.
The total size of a structure is the sum of the size of all the members or more because of appropriate alignment.	The total size of the Union is equal to the size of its largest member.
Within a structure all members gets memory allocated; therefore any member can be retrieved at any time.	While retrieving data from a union the type that is being retrieved must be the type most recently stored. It is the programmer's responsibility to keep track on that.
Any member of structure can be accessed at anytime.	Only one member of the Union can be accessed at a time.

File Handling

When dealing with files, there are 2 types of files one must know about:

1. Text Files

Text files are the normal .txt files that you can easily create using Notepad or any simple text editors. When you open those files, you'll see all the contents within the file as plain text. You can easily edit or delete the contents.

2. Binary Files

Binary files are mostly the .bin files in your computer. Instead of storing data in plain text, they store it in the binary form (0's and 1's). They can hold higher amount of data, are not readable easily and provides a better security than text files.

Following are the steps of working with files in C

1. Create a File Steam using the Structure File
2. Open a File
3. Perform Read/Write Operations to File
4. Close File

Working with Files

When working with files, you *need to declare a pointer* of type **FILE**. This declaration is needed for communication between the file and program.

Syntax: FILE *fptr;

Opening a File

Opening a file is performed using the *library function* in the **"stdio.h"** header file: **fopen()**.

Syntax: fptr = fopen("*C:/location/of/File.txt*","mode");
Example: fptr = fopen("names.txt", "r");

Here **Mode** means how would you like to interact with your File.

Following is the **List of few basic Modes** for C File Operations:

Mode	Task	Operations Possible
"r"	Searches File. If the file is opened successfully **fopen( )** loads it into memory and sets up a pointer which points to the first character in it. If the File cannot be opened **fopen( )** returns NULL.	**Reading** from the File.
"w"	Searches the File. If the File exists, it's contents are over-written. If the File doesn't exist a new File is created. Returns NULL if the unable to open the File.	**Writing** to the File
"a"	Searches File. If the File is opened successfully **fopen( )** loads it into memory and sets a pointer that points to the last character in it. If the File doesn't exist, a new file is created. Returns NULL if unable to open the file.	**Adding new contents** at the end of File.
"r+"	Searches File. If is opened successfully **fopen( )** loads it into memory and sets up a pointer to the first character in it. Returns NULL if unable to open the file.	**Reading** existing contents, **writing** new contents, **modifying** existing contents of the file.
"w+"	Searches File. If the file exists its contents are over-written. If the file doesn't exist, a new file is created. Returns NULL if unable to open the File.	**Writing** new contents, **reading** them back and **modifying** existing contents of the file.
"a+"	Searches File. If the File is opened successfully **fopen( )** leads it into memory and sets up a pointer to the first character in it. Returns NULL if unable to open the file.	**Read** existing contents of the file, **Appending new contents** to end of File. **Cannot modify** existing contents.

**We don't use single quotes for mode because there are strings and hence must be within double quotes.*

Closing a File

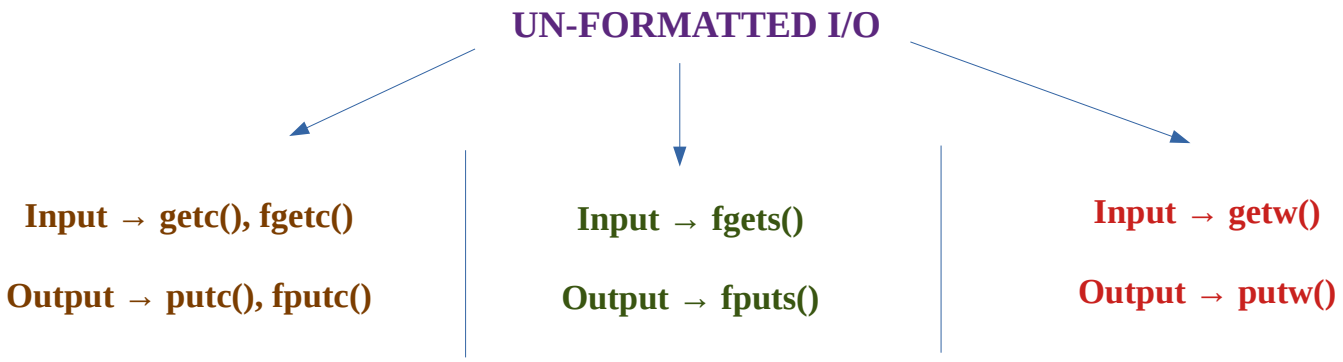
The file (both text and binary) should be closed after reading/writing. Closing a file is performed using library function **fclose().**

Syntax: **fclose(fptr);** //It is necessary to remember that **fptr** is the file pointer that was used to open the file.

Reading and Writing to a Text File.

Un-Formatted File: Un-Formatted file means that file contains only characters or integers or strings.

Formatted File: Formatted File is a combination of all, i.e., characters, integers and strings.



Following code snippet is a simple example of writing a character to a File:

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

int main()
{
  int num;
  FILE *fptr;
  fptr = fopen("program.txt","w");

  if(fptr == NULL)
  {
    printf("Error!, File not found!");
    exit(1);
  }
```

```
  printf("Enter num: ");
  scanf("%d",&num);

  putw(num,fptr);  //writes to the text file.
  fclose(fptr);

  return 0;
}
```

Tip: The best bet!! *For cases when you forget about which formatting functions was to be used with what, like for example I forgot which one was the library function to write integers to the file, then we have a library function which could be used in almost all cases.* ***fprintf()*** *and* ***fscanf().*** *Here f stands for File.*

NOTE: ***fscanf()*** *and* ***fprintf()*** *are the input output library functions for* ***FORMATTED FILES.***

Syntax: ***fprintf(File_Pointer, "type_of_value_%d_%c", value);***
fscanf(File_Pointer,"type_of_value_%d_%c", value_to_be_stored_here);

Following is a simple program to read from a text file

```
#include<stdio.h>

int main()
{
        FILE *fptr;
        fptr1 = fopen("names.txt", "w"); //to write contents to file
        fptr2 = fopen("names.txt", "r"); //to read contents from file
        char name[25];
        char storeName[25];

        printf("Enter your name: ");
        gets(name);
        fprintf(fptr1,"%s", name);   //this prints the name into the file.

        //now reading that name from file.
        fscanf(fptr2, "%s", storeName);
        //will store the contents read from the file into storeRead.
        puts(storeName);

        fclose(fptr1);  //closing file stream, fptr1
        fclose(fptr2);  //closing file stream, fptr2
        return 0;
}
```

Your reading was appreciating…

If you've made it this far going through all the text then you're probably one of the most determined and dedicated person I've met so far. However this course does not include C Preprocessor and operation on bits but would be added in future as per the needs. Congratulations! You've completed a crash course of C! You are permitted for celebration now ;D

Have a nice and wonderful day!!