URBAN EDGE

C Programming Noks by CodeWithHarry

What is Programming?

Computer Programming is a medium for us to communicate with computers. Just like we use 'Hindi' or 'English' to communicate with each other, programming is a way for us to deliver our instructions to the computer.

What is C?

C is a programming language.
C is one of the oldest and finest programming languages.
C was developed by Dennis Ritchie at AT&T's Bell Labs, USA in 1972.

Uses of C

C Language is used to program a wide variety of systems. Some of the uses of C are as follows:

1. Major parts of Windows, Linux and other operating systems are written in C.
2. C is used to write driver programs for devices like Tablets, printers etc.
3. C language is used to program embedded systems where programs need to run faster in limited memory (Microwave, Cameras etc.)
4. C is used to develop games, an area where latency is very important. ie. Computer has to react quickly on user input.

URBAN EDGE

Chapter 1 - Practice Set

Q1 Write a C program to calculate area of a rectangle:

(a) Using hard coded inputs
(b) Using inputs supplied by the User

Q2 Calculate the area of a circle and modify the same program to calculate the volume of a cylinder given its radius and height.

Q3 Write a program to convert Celcius (Centigrade degrees temperature to Farenheit)

Q4 Write a program to calculate simple interest for a set of values representing principal, no of years and rate of interest.

URBAN EDGE

Chapter 2: Instructions and Operators

A C program is a set of Instructions. Just like a recepie - which contains Instructions to prepare a particular Dish.

Types of Instructions

1. Type declaration Instruction
2. Arithmetic Instruction
3. Control Instruction

Type declaration Instruction

```
int a;
float b;
```

Other Variations:

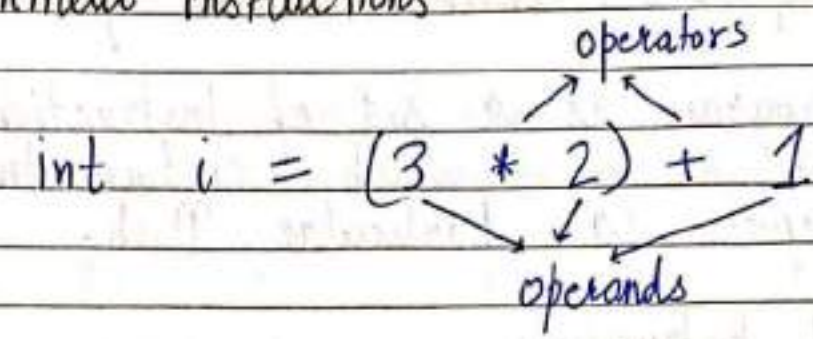
```
int i=10; int j=i; int a=2
int j= a + j - i;
```

```
float b = a+3; float a=1.1
```
=> ERROR! as we are trying to use a before defining it.

```
int a, b, c, d;
a=b=c=d=30;
```
=> Value of a, b, c & d will be 30 each.

URBAN EDGE

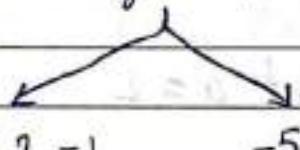
Arithmetic Instructions



Operands can be int / float etc.
+ - * / are arithmetic operators

```
int b=2, c=3;
int z; z = b*c;   ✓ legal
int z;  b*c = z;  ✗ Illegal (Not allowed)
```

% → Modular division operator
% → Returns the remainder
% → Cannot be applied on float
% → Sign is same as of numerator (-5%2 = -1)



Note :-

1. No operator is assumed to be present

```
int i = ab   → Invalid
int i = a*b  → valid
```

2. There is no operator to perform exponentiation in C. However we can use pow(x, y) from <math.h> (More later)

URBAN EDGE

Type Conversion

An Arithmetic operation between

Int and Int → Int
Int and float → Float
Float and float → Float

$5/2 \rightarrow 2$ $\quad$ $5.0/2 \rightarrow 2.5$ } Important!!

$2/5 \rightarrow 0$ $\quad$ $2.0/5 \rightarrow 0.4$ }

Note :-

`int a = 3.5;` In this case 3.5 (float) will be demoted to 3 (int) because a is not able to store floats.

`float a = 8;` a will store 8.0
8 → 8.0 (promotion to float)

Quick Quiz :

Q `int k = 3.0/9` Value of k? and why?

S $3.0/9 = 0.333..$ but since k is an int, it cannot store floats & value 0.33 is demoted to 0.

URBAN EDGE

Operator precedence In C

$3 * x - 8y$ is $(3x) - (8y)$ or $3(x - 8y)$?

In C language Simple mathematical rules like BODMAS, no longer Applies.

The answer to the above question is provided by operator precedence & associativity.

Operator precedence :- The following table lists the operator priority in C

Priority	Operators
1st	* / %
2nd	+ -
3rd	=

Operators of higher priority are evaluated first in the absence of parenthesis.

Operator Associativity :- When operators of equal priority are present in an expression, the tie is taken care of by associativity.

$x * y / z \Rightarrow (x * y) / z$

$x / y * z \Rightarrow (x / y) * z$

*, / follows left to right associativity

URBAN EDGE

Control Instructions
Determines the flow of control in a program
Four types of control Instructions in C are:

1. Sequence Control Instruction
2. Decision Control Instruction
3. Loop Control Instruction
4. Case Control Instruction.

URBAN EDGE

Chapter 3 - Conditional Instructions

Sometimes we want to watch comedy videos on YouTube if the day is Sunday.
Sometimes we order junk food if it is our friend's birthday in the hostel.
You might want to buy an Umbrella if its raining and you have the money.
You order the meal if dal or your favorite bhindi is listed on the menu.

All these are decisions which depends on a condition being met.
In C language too, we must be able to execute instructions on a condition(s) being met.

Decision Making Instructions in C
→ If - else Statement
→ Switch Statement

If - else Statement

The Syntax of an If - else Statement in C looks like:

```
if (condition to be checked) {
    Statements - if - condition - true;
}
else {
    Statements - if - condition - false;
}
```

URBAN EDGE

Code example:

```
int a = 23;

if (a > 18) {
    printf("You can drive \n");
}
```

Note that else block is not necessary but optional.

Relational Operators in C

Relational operators are used to evaluate conditions (true or false) inside the if statements. Some examples of relational operators are :

==, >=, >, <, <=, !=

↓ equals ↓ greater than or equal to ↓ not equal to

Important note : '=' is used for assignment where as '==' is used for equality check.

The condition can be any valid expression. In C a non-zero value is considered to be true.

Logical Operators

&&, || and ! are three logical operators in C. These are read as "AND", "OR" and "NOT". They are used to provide logic to our C programs.

URBAN EDGE

Usage of Logical Operators:

(i) && → AND → is true when both the conditions are true

"1 and 0" is evaluated as false.
"0 and 0" is evaluated as false.
"1 and 1" is evaluated as true.

(ii) || → OR → is true when at least one of the conditions is true. (1 or 0 → 1)(1 or 1 → 1)

(iii) ! → returns true if given false and false if given true

!(3==3) → evaluates to false
!(3>30) → evaluates to true.

As the number of conditions increases, the level of indentation increases. This reduces readability. Logical operators come to rescue in such cases.

else if clause
Instead of using multiple if statements, we can also use else if along with if thus forming an if-else if-else ladder.

```
if {
// Statements;
}
else if {
...
}
else {...
}
```

URBAN EDGE

Using if - else if - else reduces indents
The "last "else" is optional
Also there can be any number of "else if"

Last else is executed only if all conditions fail.

Operator precedence

Priority	Operator
1st	!
2nd	*, /, %
3rd	+, -
4th	<, >, <=, >=
5th	==, !=
6th	&&
7th	||
8th	=

Conditional Operators
A short hand "if - else" can be written using the conditional or ternary operators

Condition ? expression-if-true : expression-if-false

Ternary operators

URBAN EDGE

Switch Case Control Instruction

Switch-Case is used when we have to make a choice between number of alternatives for a given variable.

```
Switch (integer-expression)
{
  Case C1:
      Code;

  Case C2:
      Code;

  Case C3:
      Code;

  default:
      Code;
}
```

C_1, C_2 & C_3 → Constants

Code → Any valid C code.

The value of integer-expression is matched against C_1, C_2, C_3 .... If it matches any of these Cases, that case along with all subsequent "Case" and "default" Statements are executed.

* Quick Quiz: Writ a program to find grade of a student given his marks based on below:
  - → 90 - 100 → A
  - → 80 - 90 → B
  - → 70 - 80 → C
  - → 60 - 70 → D
  - → < 70 → F.

URBAN EDGE

Important Notes

1. We can use switch-case statements even by writing cases in any order of our choice (not necessarily ascending)

2. char values are allowed as they can be easily evaluated to an integer

3. A switch can occur within another but in practice this is rarely done.

URBAN EDGE

Chapter 4 - Loop Control Instruction

Why Loops

Sometimes we want our programs to execute few set of instructions over and over again. for ex: printing 1 to 100, first 100 even numbers etc.

Hence Loops make it easy for a programmer to tell computer that a given set of instructions must be executed repeatedly.

Types of Loops

Primarily, there are three types of loops in C language:

1. While loop
2. do - while loop
3. for loop

We will look into these one by one

While loop

```
While (condition is true) {
    // Code
    // Code
}
```

⇒ The block keeps executing as long as the condition is true.

URBAN EDGE

An example:

```
int i = 0

While (i <= 10) {
    printf (" The value of i is %d", i); i++;
}
```

Note: If the condition never becomes false, the while loop keeps getting executed. Such a loop is known as an infinite loop.

Quick Quiz: Write a program to print natural numbers from 10 to 20 when initial loop counter i is initialized to 0.

The loop counter need not be int, it can be float as well.

Increment and decrement operators

i++ → i is increased by 1
i-- → i is decreased by 1

```
printf ("--i = %d", --i);
```

This first decrements i and then prints it

```
printf (" i-- = %d", i--);
```

This first prints i and then decrements it

URBAN EDGE

* +++ operator does not exist ⇒ Important
* += is compound assignment operator just like -=, *=, /= & %= ⇒ Also Important

do-While Loop.
The syntax of do-while loop looks like this:

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

do-while loop works very similar to while loop.
While → checks the condition & then executes the code
do-while → Executes the code & then checks the condition

do-while loop = While loop which executes at least once.

↳ Quick Quiz: Wrik a program to print first n natural numbers using do-while loop.

Input : 4

Output : 1
2
3
4

URBAN EDGE

for Loop

The syntax of for loop looks like this:

```
for( initialize ; test ; increment
                         or decrement)
  {
   // Code;
   // Code;
   // Code;
  }
```

Initialize → Setting a loop Counter to an initial value
Test → Checking a condition
Increment → Updating the loop Counter

An example:

```
for( i=0 ; i<3 ; i++) {
    printf( "%d", i);
    printf( "\n");
  }
```

Output:

```
0
1
2
```

Quick Quiz: Write a program to print first n natural numbers using for loop

URBAN EDGE

A Case of Decrementing for loop

```
for (i=5; i; i--)
    printf ("%d\n", i);
```

This for loop will keep on running until i becomes 0.
The loop runs in following steps:

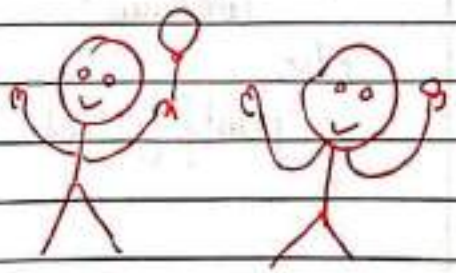
1. i is initialized to 5
2. The condition "i" (0 or none) is tested
3. The code is executed
4. i is decremented
5. Condition i is checked & code is executed if its not 0.
6. & so on until i is non 0

Quick Quiz: Write a program to print n natural numbers in reverse order.

The break Statement in C

The break statement is used to exit the loop irrespective of whether the condition is true or false.
Whenever a "break" is encountered inside the loop, the control is sent outside the loop

Let us see this with the help of an Example



```
for (i=0; i<1000; i++) {
  printf ("%d\n", i);
  if (i==5) {
    break;
  }
}
```

output ⇒ 0
1
2
3
4
5

and not 0 to 100 ☺

The Continue statement in C

The Continue statement is used to immideately move to the next iteration of the loop.

The control is taken to the next iteration thus skipping everything below "Continue" inside the loop for that itelation

Let us look at an example

```
int skip = 5;
int i=0;

While (i<10) {
  if (i!=skip)
    Continue;
  else
    printf ("%d", i);
}
```

output ⇒ 5

and not 0 ... 9

URBAN EDGE

Notes:

1. Sometimes, the name of the variable might not indicate the behaviour of the program.
2. break statement completely exits the loop.
3. Continue statement skips the particular iteration of the loop.

URBAN EDGE

Chapter 5 - Functions and Recursion

Sometimes our program gets bigger in size and its not possible for a programmer to track which piece of code is doing what.
Function is a way to break our code into chunks so that it is possible for a programmer to reuse them.

What is a Function?

A function is a block of code which performs a particular task.
A function can be reused by the programmer in a given program any number of times.

Example and Syntax of a function

```
# include <stdio.h>

Void display();   => Function prototype

int main() {
  int a;
  display();      => Function call
  return 0;
  }

Void display() {          => Function definition
  printf("Hi I am display");
  }
```

Function prototype
Function prototype is a way to tell the compiler about the function we are going to define in the program.
Here void indicates that the function returns nothing.

Function call
Function call is a way to tell the compiler to execute the function body at the time the call is made.
Note that the program execution starts from the main function in the sequence the instructions are written

Function definition
This part contains the exact set of instructions which are executed during the function call. When a function is called from main(), the main function falls asleep and gets temporarily suspended. During this time the control goes to the function being called. When the function body is done executing main() resumes.

Quick Quiz → Write a program with three functions
1. Good morning function which prints "Good Morning"
2. Good afternoon function which prints "Good Afternoon"
3. Good night function which prints "Good night"

main() should call all of these in order 1 → 2 → 3

URBAN EDGE

Important Points

→ Execution of a C program starts from main()

→ A C program can have more than one function

→ Every function gets called directly or indirectly from main()

→ There are two types of functions in C. Lets talk about them

Types of functions

1. Library functions → Commonly required functions grouped together in a library file on disk
2. User defined functions → These are the functions declared and defined by the user.

Why use functions?

1. To avoid rewriting the same logic again and again.
2. To keep track of what we are doing in a program
3. To test and check logic independently.

URBAN EDGE

Passing values to functions

We can pass values to a function and can get a value in return from a function.

```
int sum(int a, int b)
```

The above prototype means that sum is a function which takes values a (of type int) and b (of type int) and returns a value of type int

function definition of sum can be:

```
int sum(int a, int b) {
    int c;
    c = a+b;
    return c;
}
```

⇒ a and b are parameters

Now we can call sum(2, 3); from main to get 5 in return.

↳ Here 2 & 3 are arguments

```
int d = sum(2, 3);
```

⇒ d becomes 5

Note:

1. Parameters are the values or variable placeholders in the function definition. Ex a & b.

2. Arguments are the actual values passed to the function to make a call. Ex 2 & 3.

URBAN EDGE

3. A function can return only one value at a time

4. If the passed variable is changed inside the function, the function call doesnt change the value in the calling function.

```
int change (int a) {
    a = 77;
    return 0;
}
```

⇒ Misnomer

change is a function which changes a to 77. No if we call it from main like this

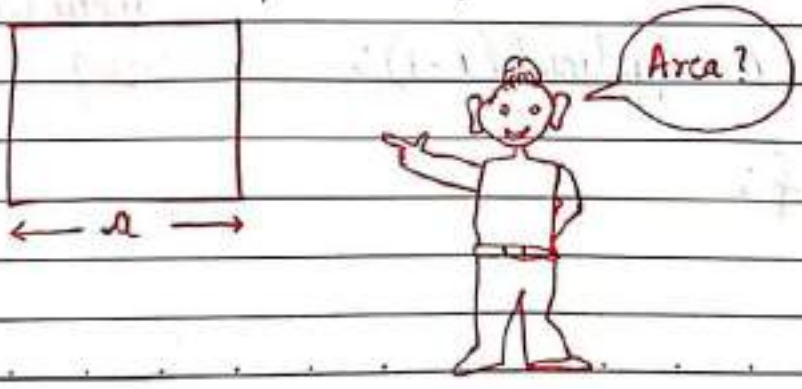
```
int b = 22
change (b);
printf(" b is %d", b);
```

⇒ The value of b remains 22

⇒ prints "b is 22"

This happens because a copy of b is passed to the change function

Quick Quiz → Use the library functions to calculate the area of a square with side a.



URBAN EDGE

Recursion

A function defined in C can call itself. This is called recursion.
A function calling itself is also called 'recursive' function.

Example of Recursion

A very good example of recursion is factorial

$$\text{factorial}(n) = 1 \times 2 \times 3 \cdots \times n$$

$$\text{factorial}(n) = \underbrace{1 \times 2 \times 3 \cdots n-1}_{\text{factorial}(n-1)} \times n$$

$$\text{factorial}(n) = \text{factorial}(n-1) \times n$$

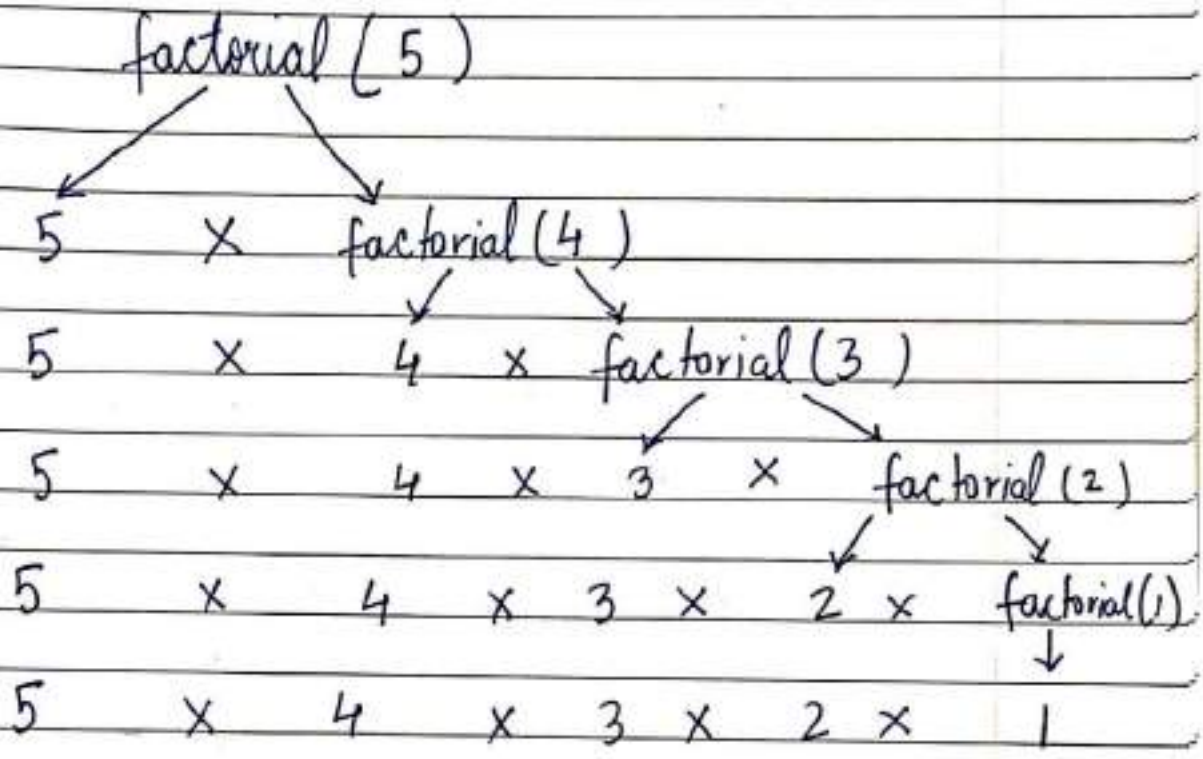
Since we can write factorial of a number in terms of itself, we can program it using recursion.

```
int factorial (int x) {
  int f;
  if (x == 0 || x == 1)
      return 1;
  else
    f = x * factorial(x-1);

  return f;

  }
```

⇒ A program to calculate factorial using recursion

URBAN EDGE

How does it work?



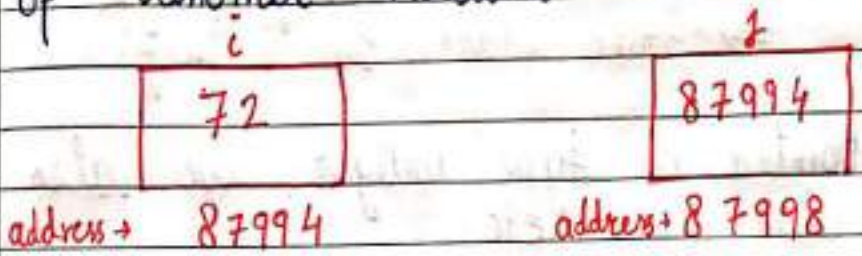
Important Notes:

1. Recursion is sometimes the most direct way to code an algorithm
2. The condition which doesn't call the function any further in a recursive function is called as the base condition.
3. Sometimes, due to a mistake made by the programmer, a recursive function can keep running without returning resulting in a memory error.

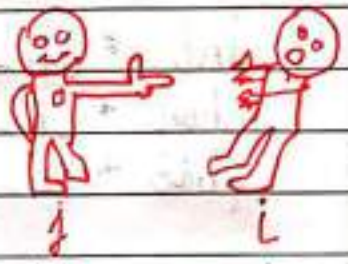
URBAN EDGE

Chapter 6 - Pointers

A pointer is a variable which stores the address of another variable



j is a pointer
j points to i



The "address of" (&) operator
The address of operator is used to obtain the address of a given variable

If you refer to the diagrams above

&i ⇒ 87994
&j ⇒ 87998

Format specifier for printing pointer address is '%u'

The 'value at address' operator (*)
The value at address or * operator is used to obtain the value present at a given memory address. It is denoted by *

*(&i) = 72
*(&j) = 87994

URBAN EDGE

How to declare a Pointer?

A pointer is declared using the following Syntax

```
int *j;      ⇒ declare a variable j of type int-pointer
j = &i       ⇒ Store address of i in j
```

Just like pointer of type integer, we also have pointers to char, float etc.

```
int *ch_ptr;   → Pointer to integer
char *ch_ptr;  → Pointer to character
float *ch_ptr; → Pointer to float
```

Although its a good practice to use meaningful variable names, we should be very careful while reading & working on porgrams from fellow programmers.

A Program to demonstrate pointers

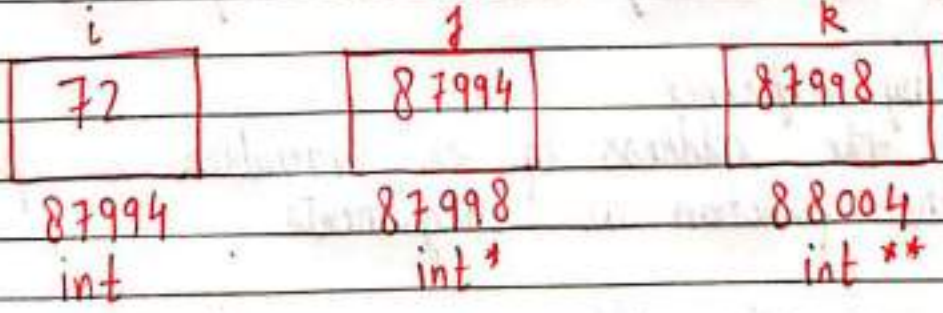
```
#include <stdio.h>
int main() {
  int i = 8;
  int *j;
  j = &i;
  printf("Add i = %u\n", &i);
  printf("Add i = %u\n", j);
  printf("Add j = %u\n", &j);
  printf("Value i = %d\n", i);
  printf("Value i = %d\n", *(&i));
  printf("Value i = %d\n", *j);
  return 0;
}
```

Output:
Add i = 87994
Add i = 87994
Add j = 87998
Value i = 8
Value i = 8
Value i = 8

This program sums it all. If you understand it, you have got the idea of pointers.

Pointer to a pointer
Just like j is pointing to i or storing the address of i, we can have another variable k which can further store the address of j. What will be the type of k

```
int **k;
k = &j;
```



We can even go further one level and create a variable l of type int *** to store the address of k. We mostly use int* and int** sometimes in real world programs.

URBAN EDGE

Types of function calls
Based on the way we pass arguments to the function, function calls are of two types.

1. Call by value → Sending the values of arguments
2. Call by reference → Sending the address of arguments

Call by value
Here the value of the arguments are passed to the function. Consider this example:

int c = Sum(3, 4); ⇒ assume x=3 and y=4
        x, y

if Sum is defined as Sum(int a, int b), the values 3 and 4 are copied to a and b. Now even if we change a and b, nothing happens to the variables x and y.
This is call by value.

In C we usually make a call by value.

Call by reference
Here the address of the variables is passed to the function as arguments.

Now since the addresses are passed to the function, the function can now modify the value of a variable in calling function using * and & operators. Example:

URBAN EDGE

```
Void Swap (int *x, int *y)
{
  int temp;
  temp = *x;
  *x = *y;
  *y = temp;
}
```

This function is capable of swapping the values passed to it. if a = 3 and b = 4 before a call to swap(a, b), a = 4 and b = 3 after calling swap.

```
int main() {
int a = 3
int b = 4        => a is 3 and b is 4
Swap(a,b)
return 0;        => Now a is 4 and b is 3
}
```

URBAN EDGE

Chapter 7 - Arrays

An array is a collection of similar (imp) elements.

One Variable ⇒ Capable of storing multiple values

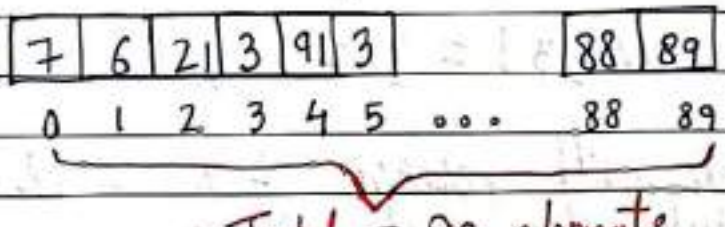
Syntax
The Syntax of declaring an Array looks like this:

```
int marks[90];    ⇒ Integer array
char name[20];    ⇒ Character array or String
float percentile[90]; ⇒ float array
```

The values can now be assigned to marks array like this:

```
marks[0] = 33;
marks[1] = 12;
```

Note: It is very important to note that the array index starts with 0.



URBAN EDGE

Accessing elements
Elements of an array can be accessed using:

Scanf ("%d", &marks[0]); ⇒ Input first value

printf ("%d", marks[0]); ⇒ output first value of the array

Quick Quiz → Write a program to accept marks of five students in an array and print them to the screen.

Initialization of an Array
There are many other ways in which an array can be initialized.

int cgpa[3] = {9, 8, 8} ⇒ Arrays can be
float marks[] = {33, 40} initialized while declaration

Arrays in memory
Consider this array:

int arr[3] = {1, 2, 3} ⇒ 1 integer = 4 bytes

This will reserve 4×3 = 12 bytes in memory
4 bytes for each integer.

1	2	3
62302	62306	62310

⇒ arr in memory

URBAN EDGE

Pointer Arithmetic

A pointer can be incremented to point to the next memory location of that type.

Consider this example

```
int i = 32;
int *a = &i;   ⇒ a = 87994      address → 87994
a++;           ⇒ Now a = 87998
```

i

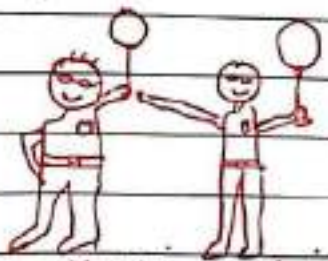
32

```
char a = 'A';
char *b = &a;  ⇒ b = 87994
b++;           ⇒ Now b = 87995

float i = 1.7;
float *a = &i; ⇒ Address of i or a = 87994
a++;           ⇒ Now a = 87998
```

Following operations can be performed on a pointers:

1. Addition of a number to a pointer
2. Subtraction of a number from a pointer
3. Subtraction of one pointer from another
4. Comparison of two pointer variables

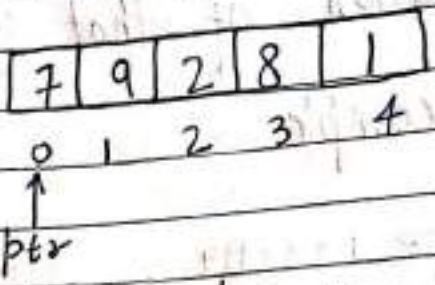
Quick Quiz → Try these operations on another variable by creating pointers in a separate program. Demonstrate all the four operations.



Yay! we understood pointer arithmetic

Accessing Arrays using pointers

Consider this array



If ptr points to index 0, ptr++ will point to index 1 & so on...

This way we can have an integer pointer pointing to first element of the array like this:

```
int *ptr = &arr[0]; → or simply arr
ptr++;
*ptr => will have 9 as its value
```

Passing arrays to functions

Arrays can be passed to the functions like this

```
printArray(arr,n);      => function call
Void printArray(int *i, int n); => function prototype
                or
Void printArray(int i[], int n);
```

URBAN EDGE

Multidimensional Arrays

An array can be of 2 dimension / 3 dimension / n dimensions

A 2 dimensional array can be defined as:

```
int arr[3][2] = { { 1, 4 }
                  { 7, 9 }
                  { 11, 22 } };
```

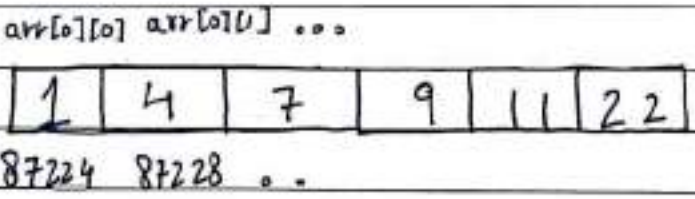
We can access the elements of this array as arr[0][0], arr[0][1] & so on...

arr[0][0] → Value = 1

arr[0][1] → Value = 4

2-D arrays in Memory

A 2d array like a 1-d array is stored in contiguous memory blocks like this:



Quick Quiz: Create a 2-d array by taking input from the user. Write a display function to print the content of this 2-d array on the screen.

Chapter 8 - Strings

A string is a 1-D character array terminated by a null ('\0')

↳ This is null character

null character is used to denote string termination characters are stored in contiguous memory locations

Initializing Strings

Since string is an array of characters, it can be initialized as follows:

```
char S[] = {'H', 'A', 'R', 'R', 'Y', '\0'};
```

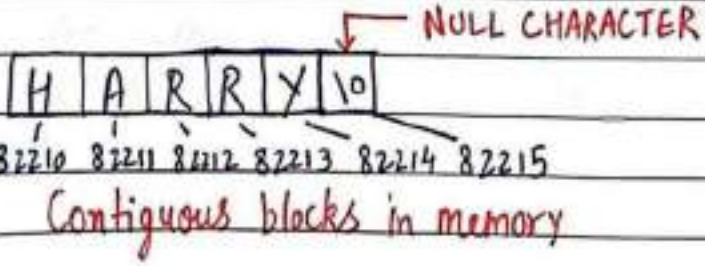
There is another shortcut for initializing strings in C language:

```
char S[] = "HARRY";
```

=> In this case C adds a null character automatically.

Strings In Memory

A string is stored just like an array in the memory as shown below



URBAN EDGE

Quick Quiz → Create a string using " " and print its content using a loop.

Printing Strings
A string can be printed character by character using printf and %c
But there is another convenient way to print strings in C.

```
Char St[] = "HARRY";
printf("%s", St);
```
⟹ prints the entire string.

Taking string input from the user
We can use %s with scanf to take string input from the user:

```
Char St[50];
Scanf("%s", &St);
```

Scanf automatically adds the null character when the the enter key is pressed.

Note:
1. The string should be short enough to fit into the array
2. Scanf cannot be used to input multi-word strings with spaces.

gets() and puts()

gets() is a function which can be used to receive a multi-word string.

```
Char st[30];
gets(st);
```
⇒ The entered string is stored in st

Multiple gets() calls will be needed for multiple strings

Likewise, puts can be used to output a string.

```
puts(st);
```
⇒ prints the string
places the cursor on the next line

Declaring a string using pointers

We can declare strings using pointers

```
Char *ptr = "Harry";
```

This tells the compiler to store the string in memory and assigned address is stored in a char pointer

Note:

1. Once a string is defined using char st[] = "Harry", it cannot be reinitialized to something else.
2. A string defined using pointers can be reinitialized

```
ptr = "Rohan";
```

Standard library functions for Strings

C provides a set of standard library functions for string manipulation.

Some of the most commonly used string functions are:

Strlen()

This function is used to count the number of characters in the string excluding the null ('\0') character.

```
int length = Strlen(st);
```

These functions are declared under <string.h> header file

Strcpy()

This function is used to copy the content of second string into first string passed to it.

```
Char source[] = "Harry";
char target[30];
Strcpy(target, source); ⇒ target now contains "Harry"
```

Target string should have enough capacity to store the source string.

URBAN EDGE

Strcat()
This function is used to concatenate two Strings

```
char S1[] = "Hello";
char S2[] = "Harry";

Strcat(S1, S2); ⇒ S1 now contains "HelloHarry"
                                  (No space in between)
```

Strcmp()
This function is used to compare two Strings. It returns: 0 if Strings are equal
Negative value if first String's mismatching character's ASCII value is not greater than second String's corresponding mismatching character. It returns positive values otherwise

```
Strcmp("Far", "Joke"); ⇒ Positive value
Strcmp("Joke", "Far"); ⇒ Negative value
```

URBAN EDGE

Chapter 9 - Structures

Arrays and strings ⇒ Similar data (int, float, char)

Structures can hold ⇒ dissimilar data

Syntax for creating Structures
A C structure can be created as follows:

```
struct employee {
    int code;
    float salary;
    char name[10];
};
```

⇒ This declares a new user defined data-type!

→ Semicolon is important

We can use this user defined data type as follows:

```
Struct employee e1;
Strcpy(e1.name, "Harry");
e1.code = 100;
e1.salary = 71.22;
```

⇒ Creating a structure variable

So a structure in C is a collection of variables of different types under a single name.

Quick Quiz: Write a program to store the details of 3 employees from user defined data. Use the structure declared above!

EDGE

Why use structures?
We can create the data types in the employee structure separately but when the number of properties in a structure increases, it becomes difficult for us to create data variables without structures. In a nut shell:

(a) Structures keep the data organized.
(b) Structures make data mangement easy for the programmer.

Array of Structures
Just like an array of integers, an array of floats and an array of characters, we can create an array of structures.

```
struct employee facebook[100]; ⇒ An array of structures
```

We can access the data using:

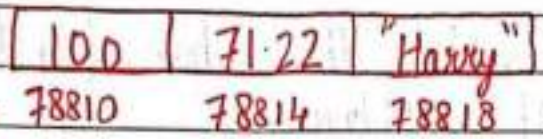
```
facebook[0].code = 100;
facebook[1].code = 101;
... & so on
```

Initializing Structures
Structures can also be initialized as follows:

```
struct employee harry = {100, 71.22, "Harry"};
struct employee shubh = {0}; ⇒ All elements set to 0
```

URBAN EDGE

Structures in memory
Structures are stored in contiguous memory locations
For the structure e1 of type struct employee, memory layout looks like this:



In an array of structures, these employee instances are stored adjacent to each other.

Pointer to structures
A pointer to structure can be created as follows:

```
struct employee *ptr;
ptr = &e1;
```

Now we can print structure elements using:

```
printf ("%d", *(ptr).code);
```

Arrow Operator
Instead of writing *(ptr).code, we can use arrow operator to access structure properties as follows

*(ptr).code or ptr -> code

Here -> is known as the arrow operator.

URBAN EDGE

Passing Structure to a function

A structure can be passed to a function just like any other data type.

```
Void show (struct employee e);
```
⇒ function prototype

Quick Quiz: Complete this show function to display the content of employee.

Typedef keyword

We can use the typedef keyword to create an alias name for data types in C. Typedef is more commonly used with structures.

```
struct Complex {
    float real;
    float img;
};
```
⇒ struct Complex C_1, C_2;
for defining Complex numbers

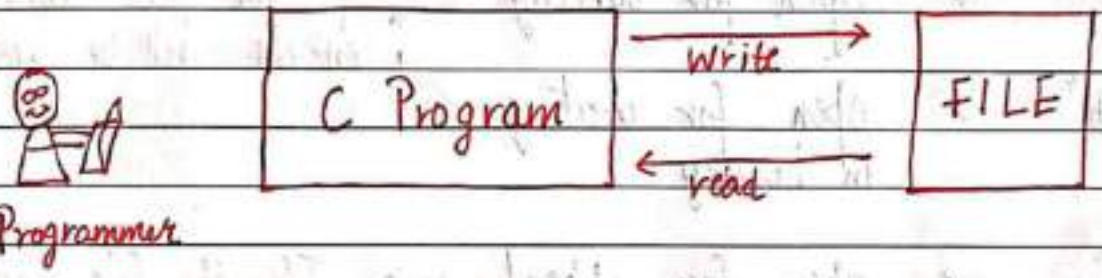
```
typedef struct Complex {
    float real;
    float img;
} ComplexNo;
```
⇒ ComplexNo C_1, C_2;
for defining Complex numbers

URBAN EDGE

Chapter 10 - File I/O

The Random Access Memory is volatile and its content is lost once the program terminates. In order to persist the data forever we use files.

A file is data stored in a storage device. A C program can talk to the file by reading content from it and writing content to it.



FILE pointer
The "FILE" is a structure which needs to be created for opening the file.
A file pointer is a pointer to this structure of the file.

FILE pointer is needed for communication between the file and the program.

A FILE pointer can be created as follows:

```
FILE *ptr;
ptr = fopen("filename.ext", "mode");
```

EDGE

File opening modes in C

C offers the programmers to select a mode for opening a file.
Following modes are primarily used in C file I/O

"r" → open for reading → If the file does not exist, fopen returns NULL

"rb" → open for reading in binary

"w" → open for writing → If the file exists, the contents will be overwritten

"wb" → open for writing in binary

"a" → open for append → If the file does not exist, it will be created

Types of files

There are two types of files:

1. Text files (.txt, .c)
2. Binary files (.jpg, .dat)

Reading a file

A file can be opened for reading as follows:

```
FILE *ptr;
ptr = fopen("Harry.txt", "r");
int num;
```

Let us assume that "Harry.txt" contains an integer We can read that integer using:

```
fscanf (ptr, "%d", &num);
```

⇒ fscanf is file counterpart of scanf

This will read an integer from file in num variable.

Quick Quiz: Modify the program above to check whether the file exists or not before opening the file.

CLOSING the file
It is very important to close the file after read or write. This is acheived using fclose as follows:

```
fclose (ptr);
```

This will tell the compiler that we are done working with this file and the associated resources could be freed.

Writing to a file
We can write to a file in a very similar manner like we read the file

```
FILE *fptr;
fptr = fopen ("Harry.txt", "w");
```

```
int num = 432;
fprintf (fptr, "%d", num);

fclose (fptr);
```

fgetc() and fputc()
fgetc and fputc are used to read and write a character from/to a file

```
fgetc (ptr)              ⟹ used to read a character from file
fputc ('c', ptr);        ⟹ used to write character 'c' to the file
```

EOF : End of file
fgetc returns EOF when all the characters from a file have been read. So we can write a check like below to detect end of file

```
While (1) {
    ch = fgetc (ptr);        ⟹ When all the content of a file has been read, break the loop!
    if (ch == EOF) {
            break;
            }
    // code
    }
```

URBAN EDGE

Chapter 11 - Dynamic Memory Allocation

C is a language with some fixed rules of programming. For example: changing the size of an array is not allowed.

Dynamic Memory Allocation

Dynamic memory allocation is a way to allocate memory to a data structure during the runtime. We can use DMA functions available in C to allocate and free memory during runtime.

Functions for DMA in C

Following functions are available in C to perform Dynamic memory Allocation:

1. malloc()
2. calloc()
3. free()
4. realloc()

malloc() function

malloc stands for memory allocation. It takes number of bytes to be allocated as an input and returns a pointer of type void

Syntax:

```
ptr = (int*) malloc(30 * sizeof(int))
```

- `(int*)` → casting void pointer to int
- `30 *` → space for 30 ints
- `sizeof(int)` → returns size of 1 int

The expression returns a null pointer if the memory cannot be allocated.

Quick Quiz: Writ a program to create a dynamic array of 5 floats using malloc().

Calloc() function
Calloc stands for continuous allocation.
It initializes each memory block with a default value of 0.

Syntax:

```
ptr = (float *) calloc (30, sizeof(float));
```

↓ Allocates contiguous space in memory for 30 blocks (float)

If the space is not sufficient, memory allocation fails and a NULL pointer is returned.

Quick Quiz: Writ a program to create an array of size n using calloc where n is an integer entered by the user.

free() function
We can use free() function to de allocate the memory.
The memory allocated using calloc/malloc is not deallocated automatically.

Syntax:

```
free(ptr);
```

⟹ Memory of ptr is released.

Quick Quiz: Write a program to demonstrate the usage of free() with malloc().

realloc() function

Sometimes the dynamically allocated memory is insufficient or more than required.

realloc is used to allocate memory of new size using the previous pointer and size.

Syntax:

```
ptr = realloc(ptr, newSize);
ptr = realloc(ptr, 3*sizeof(int));
```

⇓

ptr now points to this new block of memory capable of storing 3 integers.