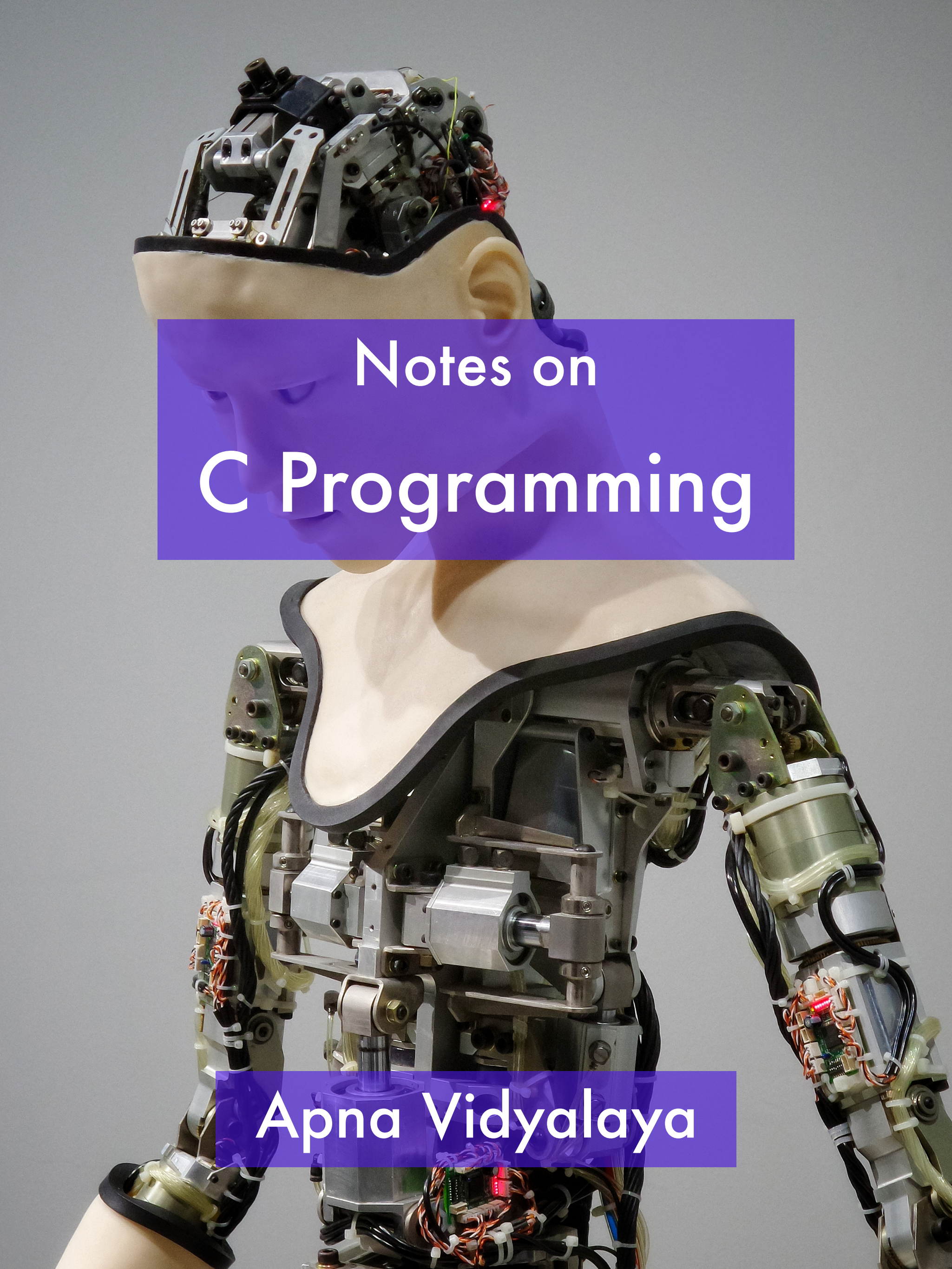

Notes on
C Programming
Apna Vidyalaya

Originally written for students of Apna Vidyalaya as a support text for our classroom program. But, for general interest, we have decided to release a free PDF version of this text through our website, even for students who have not enrolled for Apna Vidyalaya courses.

Office address:

206, 2nd floor, Om Tower

Alpha-1, commercial belt, Greater Noida, India

Pre-release edition, 2019.

Written by: Ravi Shankar

Note: We have put in full effort to make every information in this book correct, but, some errors might have creeped in. In those cases, please write back to us at apnavidyalaya@gmail.com

Cover image story: C language is the first choice when writing programs that are performant and close to the hardware. So, we have chosen an image that highlights a tight integration between software and hardware.

1. INTRODUCTION

WHY C

C is one of the most used language of the world. Significant portion of Linux and windows OS are written in C. It lays the foundation for learning more complex languages such as Java, C++ and C#. Most popular languages of current generation such as JavaScript, Java, C#, C++, D have all been derived from C and they are called C-family of languages. It is still one of the best language when it comes to interact with hardware directly

WHAT IS C

It was developed by Dennis Ritchie at Bell laboratories of AT & T company in 1972. Shortly after coming into being, it started to replace popular languages of the time and developers started to prefer it over other languages.

CONSTANTS IN C

Constants have a very important role to play in any programming language. Constants reflect on the nature of data types available in any programming languages. Constants in C can be divided into two parts

- Primary constants
- Secondary Constants

PRIMARY CONSTANTS

These are the constants which corresponds to the data types available by default in C language. They are:

- Integer constant
- Real constant
- Character constant

SECONDARY CONSTANTS

These constants correspond to the complex data types or user defined data types in C language. These are:

- Array
- Pointer
- Structure
- Union, Enum etc

RULES FOR INTEGER CONSTANT

- Must have at least one digit
- Should not have decimal
- Could be either positive or negative
- If no sign exist, then it is assumed to be positive
- No comma or blanks allowed
- Allowable range is usually fixed from machine to machine. In some cases it may be -32768 to 32767

RULES FOR REAL CONSTANTS (FRACTIONAL FORM)

- Must have at least one digit
- Must have a decimal point
- It should be either positive or negative
- Default sign is positive
- No comma or blanks are allowed
- Example:
  - -234.8
  - +78.4

- 89.5

RULES FOR REAL CONSTANTS (EXPONENTIAL FORM)

- The mantissa part end exponential part must be separated by letter e
- The mantissa part may have a positive or negative sign
- Default sign of mantissa is positive
- The exponent should have at least one digit
- The exponent should be either positive or negative. Default sign is positive
- The range is from -3.4e38 to 3.4e38

Example:

```c
#include<stdio.h>

int main(){
  float fractionalConstant = 2.3;
  printf("Value of fractionalConstant is: %f\n", fractionalConstant);
  //2.300000

  double exponentialFloat = 3e5;
  printf("Value of exponentialFloat is: %f\n", exponentialFloat);
  //300000.00000

  return 0;
}
```

RULES FOR CONSTRUCTING CHARACTER CONSTANT

- It is a single alphabet, a single digit or a single special symbol enclosed within single quotes, (')
- The maximum length of character constant can be one character.
- Example:
  - 'A'
  - 'i'

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

```c
int main(){
  char c = 'A';
  printf("Character as integer: %d\n", c);
  //65

  printf("character as character: %c\n", c);
  //c
  return 0;
}
```

RULES FOR CONSTRUCTING VARIABLE NAMES

- A variable name must be of length between 1 and 31(upper limit depends on compiler)
- It may contain, alphabets, digits or underscore only
- The first character can't be a digit, it has to be alphabet or underscore only.
- No commas or blanks are allowed
- No special character(except underscore) is allowed
- Example:
  - _init_function
  - gross_sal_1

KEYWORDS

There are a list of keywords that the language has reserved for its own use, and users/ programmers are not allowed to use them for naming variables and functions etc.

- Example:
  - goto
  - char

WRITING FIRST C PROGRAM

Every instruction in C is to be written as a separate statement.

So, C program is a series of statements. All statements are written in small case letters. The statements in a program appears in the same order in which they are supposed to be executed. Every statement must end with a ";". C has no rule for position to write statements, so, it is often called free-form language

HELLO WORLD C PROGRAM

Let's look at a Hello World C program, and then we will understand how this program is written.

```
/*
 Apna Vidyalaya
*/
#include<stdio.h>

int main() {
    printf("Hello world!\n");
    return 0;
}
```

First part , which is enclosed within "/* */" is comment. If you want to detail something in the program, you write comments. There are two types of comments in C.

Multiline comments.

```
/*
  This is a multi-line comment
*/
```

Single line comment

```
// This is a single-line comment
```

Second part is importing library (#include<stdio.h>).

When C program creates executables, it doesn't include code for all functionality in that executable, It only include code for those functionality that are absolutely required. #include is a way to include libraries (some functionalities such as input/output is available only after you have included library implementing them) in C. In our case, we included standard library, denoted by <stdio.h>, which gives us functionality for input and output i.e. printf and scanf.

Third part is main(). Almost all console C program will have a main section. Main is the entry point of a C application, this is where the program execution will start. When main

runs, it returns a value. This value tells whether the program did run successfully or not. If it runs successfully, it returns 0 otherwise it returns some other value.(1 means error). Before main(), you might have noticed int, this represents the type of value that the main will return.

Fourth part is printf. When you want to print something to the console (Terminal is Linux/ Mac and command prompt in Windows), you do it using printf.

Fifth part is return. Every time when main runs, it returns a value. Whether you explicitly return it or not. This is the standard way of returning value from main or any other function(will be taught later) in C.

C AS A FREE FLOWING LANGUAGE

```c
/*
 Apna Vidyalaya
*/
#  include    <stdio.h>

int
main() {
    printf(
        "Hello world!\n"
        );
    return 0
    ;
}
```

C is often called a free-flowing language. It places no restriction on where and how to put its statements. But there are a certain restrictions.

- #include should end at the same line
- Library should not contain space between angle brackets
- Apart from it, you have to liberty to insert spaces and newline wherever you would like to.

Although, C is a free-flowing language, you are not supposed to write code haphazardly. Programmers are expected to follow a certain coding style, which also defines indentation and alignment rules.

2. INSTRUCTIONS AND OPERATORS

A program is nothing but a set of Instructions. There are three types of instructions are C.

• Type Declaration Instruction

• Arithmetic Instruction

• Control Instruction

TYPE DECLARATION INSTRUCTION

These Instructions are used to declare variables in C. There are multiple ways we can do this.

```
int myAge;
float myAverageMarks;
```

We are other ways to do this also.

```
int myInt = 24;
float bankBalance = 100.24;
char nameInitial = 'D';

int myFirstInt = 20;
int mySecondInt = myFirstInt;


char myNameInitial, friendsNameInitial, randomInitial;
myNameInitial = friendsNameInitial = randomInitial = 'D';
```

But, the following will not work,

```
int randomAge = averageAge = 20;
```

Because, here the variable averageAge is used before being declared.

ARITHMETIC INSTRUCTION

An arithmetic instruction in C generally consists of variable name on the left side and variable names and constants on the right. These variable names and constants on the right side are connected by arithmetic operators. such as +, -, *, /.

Example:

```
myAge = TodaysYear - myDOB;
```

The variables and operators on the right are together called operands. The operands are evaluated first and then the resulting value is assigned to the variable on the left.

INTEGER AND FLOAT CONVERSION

When an integer and a real value is involved in an arithmetic operation, the integer is first promoted to real and the operation is performed. The following table should help you understand the concept.

Operation	Result	Operation	Result
5/4	1	4/5	0
5.0/4	1.25	4.0/5	0.8
5/4.0	1.25	4/5.0	0.8
5.0/4.0	1.25	4.0/5.0	0.8

TYPE CONVERSION IN ASSIGNMENT

When a variable of constant of incompatible type is assigned to a variable, the value on right in promoted or demoted to the variable type on left, before being assigned.

Example:

```c
#include<stdio.h>

int main(){
  int myAge;
  char c = 'A';
  myAge = c;
  // myAge will become 65
  printf("%d ", myAge);

  float avgMarks = 67.9;
  int marks = avgMarks;
  //marks will become 67
  printf("%d ", marks);

  int yourMarks = 99;
  float yourAvgMarks = yourMarks;
  //yourAvgMarks becomes 99.00000
  printf("%f ", yourAvgMarks);
```

```
  return 0;
}
```

Here, you can see that when you assign a real value to an integer variable, the decimal part is lost. And when you assign a character value to a numeral type variable, its ASCII value gets assigned to the given variable.

HIERARCHY OF OPERATORS

When operands involve multiple operators, then operators with highest priority is operated upon first. Let's check the following table that lists some of the operators according to its priority.

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

Example:

```
i = 2 * 3 / 4 + 4 / 4 + 8 - 2 + 5 / 8
i = 6 / 4 + 4 / 4 + 8 - 2 + 5 / 8
i = 1 + 4 / 4 + 8 - 2 + 5 / 8
i = 1 + 1 + 8 - 2 + 5 / 8
i = 1 + 1 + 8 - 2 + 0
i = 2 + 8 - 2 + 0
i = 10 - 2 + 0
i = 8 + 0
i = 8
```

ASSOCIATIVITY OF OPERATORS

When a constant/variable in involved with more than two operators of equal priority, then the operation is performed using the following rule.

- If the operator has left to right associativity, and the left side is unambiguous, then this operator will be evaluated first, otherwise this operator will wait for the other operators to get evaluated first.

- If the operator has right to left associativity, and the right side is unambiguous, then this operator will be evaluated first, otherwise this operator will wait for the other operators to get evaluated first.

In C, all the operators has associativity rules, and the operations happen according to the associativity rule. C doesn't follow BODMAS rule, it follows its priority and associativity rule. Below table mentions few operators with their associativity.

Operators	Associativity	Operators	Associativity
*	Left to right	-	Left to right
/	Left to right	!	Right to left
%	Left to right	++	Right to left
+	Left to right	"="	Right to left

CONTROL INSTRUCTIONS

In C, the program is executed sequentially, i.e, the order in which they are written. But, we can make use of control instructions to make program skip its normal and execute according the the condition provided. We will see them in the following chapters.

3.CONDITIONALS

To make a computer skip its normal flow and execute according to some condition, we make use of conditional statements. We will learn conditional statements available in C, in this chapter.

IF STATEMENT

In case your brother or sister gets more than 90% marks in an exam, you congratulate them. Let's see how we do it programatically.

```c
#include<stdio.h>

int main() {
  int marks = 99;
  if (marks > 90) {
    printf("Congratulations! \n");
  }

  if (marks <= 90) {
    printf("Work hard! \n");
  }
}
```

IF-ELSE STATEMENT

If you have only two options, and one of them have to be selected, then you can use if-else statement. The above example re-written as:

```c
#include<stdio.h>

int main() {
  int marks = 99;
  if (marks > 90) {
    printf("Congratulations! \n");
  } else {
    printf("Work hard! \n");
  }
}
```

NESTED IF-ELSE STATEMENT

C allows us to put if-else statements inside other if-else statements. And there is no limit as to how many levels you can go deep inside nesting. But, from an experience point of view, I

would suggest that you should never go deeper than level 2. Let's see an example that uses nested if-else statements.

```c
#include<stdio.h>

int main() {
  int marks = 68;
  if (marks > 90) {
    printf("Congratulations! \n");
  } else {
    if (marks > 70) {
      printf("Work hard! \n");
    } else {
      printf("Work very hard! \n");
    }
  }
}
```

ELSE-IF STATEMENTS

I didn't tell you earlier that if you have only one statement inside if or else block, then you don't need to put curly braces. I did so intentionally, because I and many others in the Industry consider it a bad practise to skip curly braces. The following example is correct.

```c
#include<stdio.h>

int main() {
  int marks = 68;
  if (marks > 90)
    printf("Congratulations! \n");
  else
    printf("Work hard! \n");

  return 0;
}
```

Well, entire if-else block is treated as one statement by c compilers. So, even the following example is correct.

```c
#include<stdio.h>

int main() {
  int marks = 68;
  if (marks > 90)
    printf("Congratulations! \n");
  else
    if (marks > 70) {
      printf("Work hard! \n");
```

```
    } else {
      printf("Work very hard! \n");
    }

  return 0;
}
```

Now, let's re-arrange the above code by moving the if inside the else statement at the line on which else is written (C is a free-flowing statement, so, obviously you can do that). You will get the following code.

```
#include<stdio.h>

int main() {
  int marks = 68;
  if (marks > 90)
    printf("Congratulations! \n");
  else if (marks > 70) {
      printf("Work hard! \n");
  } else {
    printf("Work very hard! \n");
  }

  return 0;
}
```

Now, you can see that else-if statements are just a re-arrangement of nested if-else statements. As soon as any condition becomes true, the program flow will not check any other case. It will fall out of conditionals.

SWITCH STATEMENTS

When the variables whose values you are checking in conditionals can take only a few discrete values, then switch condition can be used. Switch statements are more performant than if-else statements.

Let's see an example of switch statements.

```
#include<stdio.h>

int main() {
  char nameInitial = 'R';

  switch (nameInitial)
  {
    case 'R':
    case 'r':
```

```
      printf("You might be Ravi!\n");
      break;

    default:
      printf("You might be anyone!\n");
      break;
  }

  return 0;
}
```

In switch statements, it is important to put break statements in every case, otherwise the program will execute all the cases starting from the case which matches first. When it encounters a break statement, the program control falls out of the switch construct.

TERNARY STATEMENTS

Ternary statements can be used to perform some simple conditional checks. Let's check its usage. If the statement before the '?' is true, it returns the value right after '?' otherwise it returns the value right after ':'.

```
#include<stdio.h>

int main() {
  int var = 9;
  var = var == 5 ? 10 : 1;

  printf("%d\n", var);

  return 0;
}
```

4. LOOPS

Loops exist in programming languages to repeat a set of instruction many times. For example, if you want to print integers from 0 to 100, you won't like to write print statements 100 times. For these scenarios, loops exist in programming languages.

There are three loops in C:

- while loop
- do-while loop
- for loop

WHILE LOOP

Let's first see a program and then understand how this works.

```
#include<stdio.h>

int main() {
  int value = 10;
  int index = 0;
  while(index < value) {
    printf("Index is: %d\n", index);
    index++;
  }
}
```

```
The program will have the following output:

Index is: 0
Index is: 1
Index is: 2
Index is: 3
Index is: 4
Index is: 5
Index is: 6
Index is: 7
Index is: 8
Index is: 9
```

This is the standard syntax of while loop. while is followed by a condition in a pair of small parenthesis. If the condition evaluates to true, the loop body will execute, otherwise the control flow will go to the line immediately after the while loop.

Note that, while loop depends on conditions in the parenthesis to stop. So, inside loop body, you have to modify the loop variable, so that it eventually leads the condition to become false, otherwise the loop will become an infinite loop.

DO-WHILE LOOP

do-while loop is almost same as while loop, with one major difference that do-while loop executes at least once even if the loop condition evaluates to be false the first time. Let's see the same code that we wrote for while loop being written using do-while loop.

```
#include<stdio.h>

int main() {
  int value = 10;
  int index = 0;
  do {
    printf("Index is: %d\n", index);
    index++;
  } while (index < value);
}
```

No, doubt, the output is same.

FOR LOOP

For loop is a bit different from the other two loops in a way that it lets you initialise loop variable, increment loop variables and check for conditional inside the loop syntax only.

```
#include<stdio.h>

int main() {
  int number, tableMax = 10;
  printf("Enter the number you want the table of: ");
  scanf("%d", &number);

  for(int i = 1; i <= tableMax; i++) {
    printf("%d * %d = %d \n", number, i, number * i );
  }

  return 0;
}
```

PATTERN PRINTING

These problems are more of an aptitude problem than programming problems. The key to solving pattern problems is to identify:

- Co-relation between variables representing rows and columns
- Direction of loop (0 -> n or n -> 0)
- Print blanks or other characters conditionally
- Finding suitable patterns etc

Let's look at some of the examples.

```
/*

* * *
* * *
* * *

*/
#include<stdio.h>

int main() {
  for(int i = 0; i <= 3; i++) {
    for(int j = 0; j <= 3; j++) {
      printf("*");
    }
    printf("\n");
  }
  return 0;
}
```

```
/*

*
* *
* * *
* * * *

*/
#include<stdio.h>

int main() {
  for(int i = 0; i <= 3; i++) {
    for(int j = 0; j <= 3; j++) {
      printf("*");
```

```
    }
    printf("\n");
  }
  return 0;
}



/*

      *
    * *
  * * *
* * * *

*/
#include<stdio.h>

int main() {
  for(int i = 4; i > 0; i--) {

    for(int j = 1; j <= 4; j++) {
      if (j < i) {
        printf("  ");
      } else {
        printf("* ");
      }
    }
    printf("\n");
  }
  return 0;
}



/*

* * * * *
* *   * *
*       *

*/
#include<stdio.h>

int main() {
  int k = -1;
  for(int i = 0; i < 3; i++, k += 2) {
    int numOfStars = ( 5 - i) / 2;
    int charCount = 0;
    //
```

```
    for (int p = 0; p < numOfStars; p++) {
      printf("* ");
      charCount = charCount + 1;
    }
    //
    for( int p = 0; p < k; p ++){
      printf("  ");
      charCount = charCount + 1;
    }
    //
    while(charCount < 5) {
      printf("* ");
      charCount = charCount + 1;
    }
    printf("\n");
  }
  return 0;
}
```

5. FUNCTIONS

In the beginning of civilisation, people used to do all work all by themselves. But, as the civilisations became advanced, people started to specialise. Today, we have highly skilled individuals such as doctors, engineers, managers, lawyers, accountants etc. A person who has practised software all his life will obviously write better software than a part doctor, part engineer, part teacher, part lawyer. Similarly, modern day software is divided into multiple parts depending upon the requirement.

WHAT IS A FUNCTION?

A function is a set of statements contained within a block and achieving a very specialised objective. Let's write a function that greet people.

```
#include<stdio.h>

void greet(){
  printf("Hello! \n");
}
int main() {
  greet();
}
```

WHY TO WRITE A FUNCTION?

We have already used printf function. This function is written in stdio library. Every time you need to output something to the console, you call this function and it does the work. Outputting something to the console is a complex task, it involves talking to the host OS and giving it instruction with the data to output, and then the host OS outputs the data on console. Printf even has to parse the data. Do you remember that we give %d and %f, but the output doesn't contain these symbols. This happens because, printf does the parsing work before getting it outputted on console.

Now the question is, is it worthwhile to do all this work ourself whenever we need to output something to the console? The answer wold be no. Now the second question is, will it be worthwhile to write the same tedious logic again and again? Again the answer would be no. If a tedious task has been done once, just re-use the code and move on.

Function does the same thing, it hides how functionalities are implemented and gives you an interface or a method name to use those functionalities. You call the function using the function name and you are able to use those functionalities. Also note that, a function is

written only once, but it can be called multiple times. This is called re-usability. In short. a function:

- hides implementation details
- enables re-use

HOW TO WRITE A FUNCTION?

Generally, the functions that we write has to be declared before it can be used. And then the function has to be defined somewhere. Let's see an example first.

```c
#include<stdio.h>

int addNum(int, int);

int main() {
  int sum, num1 = 5, num2 = 8;
  sum = addNum(num1, num2);
  printf("Sum of %d and %d = %d \n", num1, num2, sum);
  return 0;
}
```

```c
int addNum(int number1, int number2){
  return number1 + number2;
}
```

The line just after the line where we included stdio library, is the declaration of addNum function. While declaring a function, we first write the data type of value that this function will return followed by the function name. Right after the function name, in a pair of parenthesis, we write the data types of the values that this function will accept.

Just after main closes, you can see function definition. Here you write the actual implementation of the function. Notice that here, you have to write the parameter name, you cannot just do away by writing the data types the way we did while declaring it.

On the line,

```c
sum = addNum(num1, num2);
```

The function, addNum is called with values of num1 and num2, and addNum and the value which addNum returns will be assigned to sum variable.

If the function body is small and/or the function is called very frequently, then you can define the function at the place where you declare it. And you won't need to declare it. This is called inline-functions, and it offers some performance improvement.

To call or invoke a function, we write function name followed by a pair of parenthesis, example: greet(). If the function accepts some arguments, then we write arguments inside the parenthesis, example: addNum(5, 6).

Remember that any function can call any function any number of times.

PASSING VALUES TO A FUNCTION

When you call a function, you may or may not call it with some values. Let's call a function without passing any value.

```
#include<stdio.h>

void sayHello();

int main() {
  sayHello();
  return 0;
}

void sayHello(){
  printf("Hello! \n");
}
```

Here, you can see we are calling sayHello without any arguments. Now, let's see an example where we are calling a function with some arguments.

```
#include<stdio.h>

int getSquare(int);

int main() {
  int square, number = 5;
  square = getSquare(number);
  printf("Square of %d = %d \n", number, square);
  return 0;
}

int getSquare(int number){
  return number * number;
}
```

Similarly, you can pass multiple values to a function. Check addNum function in the previous examples if you need the code. If you call a function by giving it some value, and the called function change the value of that variable, then the original value will remain unaffected. Because, when a function is being called, it creates a local copy of the arguments which is a duplicate of the original arguments. See the following example to understand this.

```c
#include<stdio.h>

int modifyValue(int);

int main() {
  int modifiedNumber, number = 5;
  modifiedNumber = modifyValue(number);
  printf("The original number is: %d \n", number);
  printf("Modified value: %d \n", modifiedNumber);
  return 0;
}

int modifyValue(int number){
  number = number * number + 10;
  return number;
}
```

FUNCTION CALLING FUNCTION

A function can call another function, which in turn call another function, and this can go on. In this chain, even the first function can be called again! Let's see an example:

```c
#include<stdio.h>

void pakistan() {
  printf("I am in Pakistan\n");
}
void india(){
  printf("I am i India\n");
}
void asia() {
  india();
  pakistan();
}

int main() {
  asia();
  return 0;
}
```

```
The output would be:
I am i India
I am in Pakistan
```

RECURSION

Just the way a function can call another function, it can also call itself. This functionality is called recursion. Let's write a program that gives us the factorial of a number.

```c
#include<stdio.h>

int factorial(int number) {
  if (number <= 1) {
    return number;
  }
  return number * factorial(number - 1);
}

int main() {
  int num = 5;
  int factorialValue = factorial(num);
  printf("Factorial of %d is: %d \n", num, factorialValue);
  return 0;
}
```

Let's look at another very famous program of fibonacci series.

```c
#include<stdio.h>

int fibonacci(int n) {
  if (n <= 2) {
    return 1;
  }
  return fibonacci(n - 1) + fibonacci(n - 2);
}

int main() {
  printf("Which term of fibonacci series you want to print? ");
  int N;
  scanf("%d", &N);
  int nth_term = fibonacci(N);
  printf("%dth term of fibonacci series is: %d\n", N, nth_term);
  return 0;
}
```

6. POINTERS

If you are at home, and I know your address, I can come at your address and meet you. Similarly, if you know a variable's address in C, you can go to its address and access its value. That's the idea behind pointers. In this chapter, we will learn to write code for pointers.

INTRODUCTION

What happens when you declare in integer?

```
int number = 10;
```

The computer choose some location in memory, let's say 4096, the memory will be given the name: number, and a value 10 will be put into that memory. It will also mark that this memory is of type number.

ACCESS VALUES IN A VARIABLE USING POINTER

How can you access the value of number? Till now, we have been using the name of the variable, i.e. number to access the values. But, we can also access the value if we knew the address where number is stored. To store the address of a variable, you need another variable. The variable which stores the address of another variables is called pointers.

Let's see an example:

```
#include<stdio.h>

int main() {
  int num = 8;
  int * pt = &num;
  printf("%d\n", *pt);
  return 0;
}
```

Here, pt is a variable of type (int *). Which means, pt is a pointer which can hold address of a variable of type int. If I were to break the above program, I would do the following

```
#include<stdio.h>

int main() {
  int num = 8;
  int * pt;
  pt = &num;
```

```
  printf("%d\n", *pt);
  return 0;
}
```

The second program adds more clarity. pt, which is a variable if type 'int *', was assigned the address of num(using &num). And the value at address pt was being outputted using *pt (value at the address pt).

Just to be clear, * in both the following lines means the different things.

```
int * pt;
```

and

```
printf("%d\n", *pt);
```

In the first line, * is associated to int, this is (int *), which is a data type. And the overall meaning of this statement is that pt is a variable of type int *.

In the second line, * is associated to pt, this is (*pt), which means that the value at the address pt.

For clarity, you can read the following constructs as:

```
  int * pt // integer pointer pt
  &num // address of num
  *pt // value at the address pt
```

PRINTING THE ADDRESS OF A VARIABLE

If you want to print the value of a pointer, i.e. the address that it contains, then you can do so. The format specifier is %p. Many books such as "Let's C by Yashwant Kanetkar", writes that you can do so using %u, but that is incorrect. Let's see an example:

```
#include<stdio.h>

int main() {
  int num = 8;
  int * pt;
  pt = &num;
  printf("The address where pt points to is: %p \n", pt);
  return 0;
}
```

Well, the more correct version would be

```
#include<stdio.h>

int main() {
```

```
  int num = 8;
  int * pt;
  pt = &num;
  printf("The address where pt points to is: %p \n", (void *)pt);
  return 0;
}
```

The difference in in the way pt is being passed to printf function. In the second case, pt is explicitly type casted to a pointer of type void, whereas in the first case that is implied.

When you print the memory address, you will almost certainly get a hex number. The standard doesn't define whether it should be hex or decimal, but almost all the implementations gives out the address in hex format.

CHANGING VALUES OF A VARIABLE USING POINTER

We will first see a program, and then we will try to understand.

```
#include<stdio.h>

int main() {
  int num = 8;
  int * pt = &num;
  *pt = *pt + 10; // value at the address pt = value at the address pt + 10
  printf("The new value in num is: %d \n", num);
  return 0;
}
```

Read the comment, and the given line will become clear to you.

TYPES OF POINTERS

All the data types that exist in C can have a pointer. Including the user defined data types also. In the following program, we will use a pointer of type char and float.

```
#include<stdio.h>

int main() {
  char myChar = 'A';
  char * myPointer = &myChar;
  *myPointer = 'B';
  //
  float myFloat = 3.5;
  float * myFloatPointer = &myFloat;
  *myFloatPointer = 4.6;
```

```
  printf("The values contained in myChar and myFloat are: %c and %f \n", myChar, myFloat);
  return 0;
}
```

POINTERS IN FUNCTION CALLS

In C, by default function calls are call by value. Which means that if you pass a variable to C program as an argument, then that function will make a local copy of that variable. If the function changes the value in its own body, the original variable will remain unaffected.

But, if you pass a pointer of a variable to a function call, and the called function changes the value at that pointer, then the original variable will gets changed. This is quite logical. We all know that pointers are also a variable, a variable that can hold address. When you pass a pointer to a function, the function will have its own local copy of the pointer variable, but the address that this pointer points to remain the same. If you go to that address and make some change, that change will be permanent. This may sound a bit confusing at first reading, but will become clear with time. Let's see an example.

```
#include<stdio.h>
void doubleTheNum(int *pt) {
  *pt = *pt * 2;
}
int main() {
  int num = 7;
  doubleTheNum(&num);
  printf("New value at num = %d \n", num);
  return 0;
}
```

Did you notice some difference in the pointer treatment. Instead of creating a new pointer variable, we just passed &num to the function that accepts a pointer to an integer.

OPERATIONS ON POINTER

What will happen if you add 1 to an integer pointer? Will in increate by one?

The answer is No. When you add something, let's say A to a pointer, the pointer first find out the size of the data type to which it points to, let's say it is 4. Then it calculates a number, say N (= 4 * A), which would be the space required to write A number of integer in memory. And then it moves N steps forward in memory. In our case, it will be A * 4.

If the integer pointer points to 4096 and size of int is 4 bytes. Then adding one to the given pointer will make it 4100. Let's see it through an example.

```
#include<stdio.h>

int main() {
  int num = 90;
  int * p = &num;
  printf("p points to: %p \n", p);
  p = p + 1;
  printf("Now, p points to: %p\n", p);
  return 0;
}
```

The output on my system is:

```
p points to: 0x7ffeee6f89b8
Now, p points to: 0x7ffeee6f89bc
```

If I run the program again, the output will be different. But, the difference between the two hex values printed will remain the same. On my system, integer in C is of four bytes. So, increasing the pointer by one has increased the pointer value by 4.

You can subtract a point from another pointer of compatible type, you will get the value which is difference of the pointers values divided by the size of the data types it points to. Whereas, you cannot add two pointers.

7. ARRAYS

How will you store the marks of each student in a class? Will you have a variable corresponding to every student? If yes, then won't it become tough to manage? Don't worry, the answer is No. And the solution lies in Array.

INTRODUCTION

Arrays are a single variable that holds multiple data of same types. Let's see an example to understand it better.

```
#include<stdio.h>

int main() {
  int marks[] = { 2, 3, 90, 43, 67 };
  for(int i = 0; i < 5; i++) {
    printf("Value at index %d is %d\n", i, marks[i]);
  }
  return 0;
}
```

The output would be:

```
Value at index 0 is 2
Value at index 1 is 3
Value at index 2 is 90
Value at index 3 is 43
Value at index 4 is 67
```

marks is an array which contains 5 integers, and those integers can be accessed using their indices. Let's learn it in detail.

ARRAY DECLARATION

Array declaration is similar to variable declaration, except that here we have to add a pair of brackets after the array name with the size of the array. Example:

```
int marks[10];
```

If you want to define array while declaring then you can skip explicitly giving the size. In that case the syntax will become:

```
int marks[] = { 2, 3, 90, 43, 67 };
```

If you don't assign values to array while declaring, then you will have to assign values to each elements of array later on.

To access elements of an array, you can write array name with [], with the index of the element inside []. Note that, in C, arrays are 0 indexed. So, first element is at 0, second element at index 1, and so on. Let's see these concepts through an example:

```
#include<stdio.h>

int main() {
  int battingAverage[5];
  for(int i = 0; i < 5; i++) {
    battingAverage[i] = 30 + i;
  }
  for(int i = 0; i < 5; i++) {
    printf("Batting average of %dth player is: %d\n", i + 1, battingAverage[i]);
  }
  return 0;
}
```

You can have arrays on any data types, int, char, struct or any other data types.

ARRAY REPRESENTATION IN MEMORY

Arrays in C are nothing but a series of data put together in memory on continuous locations. So, if you get a pointer to the first element of the array, and you increase the pointer by 1, your pointer will now point to the second element of the array.

C doesn't care to store any other info about arrays. If you create an array of size 10 and you try to assign a value to 11th element, C will let you do so. It will overwrite the data stored at the location next to 10th element of the array with the value provided by you. If that location had something important, then that would be lost. This is considered to be a very big disadvantage with C language. For example: this program which is accessing values outside arrays will compile fine and won't give any error.

```
#include<stdio.h>

int main() {
  int battingAverage[5];
  for(int i = 0; i < 7; i++) {
    battingAverage[i] = 30 + i;
  }
  for(int i = 0; i < 10; i++) {
    printf("Batting average of %dth player is: %d\n", i + 1, battingAverage[i]);
  }
```

```
  return 0;
}
```

Since, arrays has no bounds checking mechanism, the program is responsible for managing and respecting the boundary of the array.

ARRAYS AND POINTERS

Arrays and pointers and more similar than you can think intuitively. We will learn the similarity one by one with program examples.

They both can be accessed using same syntax.

```
#include<stdio.h>

int main(){
  int d[] = {2, 93, 4};
  int *p = d;

  printf("Accessing array using pointer with array syntax\n");
  for(int i = 0; i < 3; i++){
    printf("%d\n", p[i]);
  }
  printf("\nNow accessing array using array name with pointer syntax\n");
  for(int i = 0; i < 3; i++){
    printf("%d\n", *(p +i));
  }

  return 0;
}
```

The output would be:

```
Accessing array using pointer with array syntax
2
93
4

Now accessing array using array name with pointer syntax
2
93
4
```

DIFFERENCE BETWEEN ARRAY AND POINTERS

You can perform arithmetic on pointers be adding one to it, but you cannot do the same to an array. Example:

```
#include<stdio.h>

int main() {
  int arr[] = {3, 4, 5, 6};
  int *pt = arr;
  pt += 1;// works fine
  printf("2nd value of arr using pointer is: %d\n", *pt);

  // arr += 1;// Gives compilation error
}
```

Sizeof operator returns size of entire array when operated upon array, but in case of pointers, it returns the size of the pointer. Example:

```
#include<stdio.h>

int main() {
  int arr[] = {3, 4, 5, 6, 8, 2, 34};
  int *pt = arr;
  printf("sizeof array is: %lu\n", sizeof(arr));

  printf("sizeof pointer is: %lu\n", sizeof(pt));
}
```

The output on my system is:

```
sizeof array is: 28
sizeof pointer is: 8
```

Assigning address to an array variable is not allowed.

```
#include<stdio.h>

int main() {
  int arr[] = {3, 4};
  int myInt = 8;
  // arr = &myInt; // This will cause compilation error
  return 0;
}
```

If the character array holds a string value, then the string can be modified. But, is the character pointer points to a string literal, then the value of the string cannot be modified. Example:

```
#include<stdio.h>

int main() {
  char myStr[] = "myFirstString";
  printf("%s\n", myStr);
  myStr[1] = 'R'; // This is allowed
  printf("%s\n", myStr);

  char *charP = "mySecondStr";
  printf("%s\n", charP);
  // *(charP + 1) = 'T'; // This will compile fine, but won't run properly
  printf("%s\n", charP);
  return 0;
}
```

PASSING ARRAYS TO A FUNCTION

When you pass an array to a function, the called function will receive a pointer, even if the callee has passed an array and the callee's formal argument is declared using square brackets.

```
#include<stdio.h>

void arrayProp(int arr[]) {
  int arrSize = sizeof(arr);
  printf("sizeof arr in arrayProp: %d \n", arrSize);
  arr += 1;// okay because arr is treated as pointer
  printf("2nd element of array is: %d\n", *arr);
}

int main(){
  int arr[] = {4, 1, 0, 89, 60, 2, 24};
  int sizeOfArr = sizeof(arr);
  printf("sizeof arr in main: %d \n", sizeOfArr);
  // arr += 1;// This line will give compilation error because arr is an array
  // printf("second element of array is: %d \n", *arr);

  arrayProp(arr);
  return 0;
}
```

The output on my system is:

```
sizeof arr in main: 28
sizeof arr in arrayProp: 8
2nd element of array is: 1
```

ARRAYS OF DYNAMIC SIZE

Just the way, we have been initialising arrays of constant size, we can also initialise arrays of dynamic size. //TODO: performance analysis

```c
#include<stdio.h>

int main() {
  int n;
  printf("Enter the size of array you want: ");
  scanf("%d", &n);
  int myArr[n];

  for(int i = 0; i < n; i++) {
    myArr[i] = 30 + i;
  }
  for(int i = 0; i < n; i++) {
    printf("Value at index %d is: %d\n", i, myArr[i]);
  }
  return 0;
}
```

There is one more way to initialise an array of dynamic size. That is though pointers.

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

int main(){
  int *p;
  int n;
  printf("Enter the size of array you want: ");
  scanf("%d", &n);
  p = (int *)malloc(sizeof(int *) * n);

  for(int i = 0; i < n; i++) {
    p[i] = 30 + i;
  }
  for(int i = 0; i < n; i++) {
    printf("Value at index %d is: %d\n", i, p[i]);
  }
  return 0;
}
```

Some time back (not long back), dynamic sized arrays were possible only through pointer method. So, if you are writing code that will be compiled on old compilers, then you should prefer pointer method.

MULTIDIMENSIONAL ARRAYS

Arrays don't need to be just one dimensional. It can be 3D, 4D and so on. A 2D array is called a matrix. Let's look at the syntax for initialising a 2D array.

```
#include<stdio.h>

int main() {
  int my2DArr[2][3] = {
    { 2, 5, 3 },
    { 5, 2, 5 }
  };
  printf("my2DArr: %d\n", my2DArr[1][2]);
  //
  int myNext2DArray[2][3] = {1, 5, 4, 0, 7, 3 };
  printf("myNext2DArray: %d\n", myNext2DArray[1][2]);
  //
  int myThird2DArr[][2] = {
    2, 3, 4, 7
  };
  printf("myThird2DArr: %d\n", myThird2DArr[1][1]);
  return 0;
}
```

The way we have declared my2DArr is the most popular and recommended way of declaring 2D arrays. But all the syntaxes shown above are valid syntaxes.

Notice the way we declared myThird2DArr array, we omitted the size of first dimension. In multi-dimensional arrays, if we are initialising an array while declaring, then we can skip the size of first dimension. C will infer it from the number of elements provided.

8. STRINGS

In the world away from computers, every information is stored as a text. And the way we store those texts in computer is string. So, string is a very important concept in any programming language.

STRINGS IN C

Let's have a look at a program utilising Strings in C. The declaration happens by writing string name followed by [].

```c
#include<stdio.h>

int main() {
  char myString[] = "Apna Vidyalaya";
  printf("%s\n", myString);
  return 0;
}
```

STRINGS AND ARRAYS

In C, strings are nothing but an array of characters, terminated by a null character, '\0'.

Example:

```c
#include<stdio.h>

int main() {
  char myString[] = { 'R', 'a', 'v', 'i', '\0' };
  printf("%s\n", myString);
  return 0;
}
```

The following lines, necessarily means the same

```c
char myString[] = { 'R', 'a', 'v', 'i', '\0' };
```

and

```c
char myString[] = "Ravi";
```

The syntax in the second case is available just for convenience, internally the C treats strings an an character array only, as evident in the first case. :

STRINGS AND POINTERS

The way we have been using pointers to access integer arrays, we can also do so for strings. Let's see an example:

```
#include<stdio.h>

int main() {
  char myString[] = "Apna Vidyalaya";
  char * charPtr = myString;
  while(*charPtr != '\0') {
    printf("%c", *charPtr);
    charPtr++;
  }
  printf("\n");
  return 0;
}
```

One thing you might have notices is that, we are checking for '\0'. '\0' is a special character in C and it is known as null character. A string in C must terminate with this character.

MANIPULATING CHARACTERS OF A STRING

The way we have been accessing and manipulating individual elements of an array, we can do so for strings also. Because, strings are internally character arrays. Let's see an example.

```
#include<stdio.h>

int main() {
  char myString[] = "Sun";
  printf("Original String is: %s\n", myString);
  myString[0] = 'R';
  printf("Modified string is: %s\n", myString);
  char * charPointer = myString;
  *charPointer = 'B';
  printf("Further modified string is: %s\n", myString);
  return 0;
}
```

STANDARD LIBRARY FUNCTIONS FOR STRING

There are multiple standard library functions in C, which gives useful information about C. To use standard library functions for strings, we have to include string library in our program. In this section, we will learn about few of the string library functions.

strlen: This function returns the length of a string. Example:

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

```c
int main() {
  char myString[] = "Sun";
  int length = strlen(myString);
  printf("The length of the given string is: %d \n", length);
  return 0;
}
```

strcpy: This function copies content of one string into another. Let's see an example:

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

int main() {
  char myString[] = "Sun";
  char newString[20];
  strcpy(newString, myString);
  printf("New copied string is: %s \n", newString);
  return 0;
}
```

strcmp: This function compares two strings and returns the difference of first un-matching character. Let's see an example:

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

int main() {
  char myString[] = "Sun";
  char newString[] = "Run";
  int compareResult = strcmp(newString, myString);
  printf("Comparison of %s and %s gives: %d \n", newString, myString, compareResult);
  return 0;
}
```

The result would be:

```
Comparison of Run and Sun gives: -1
```

9. STRUCTURES

Arrays are a way to handle similar data, but what if the data is not simple like an integer or character. How will we store data of a book, such as publisher, price etc. They are all related, but their data type is not similar.

WHY STRUCTURES?

To handle data, which are not similar, but are closely related, C provides us with structures. The way integers and characters are a data type that are defined by C, structures are user defined data type. Once you have created a data type using struct, you can then go on to create variables and even arrays of this data type.

Let's see how we can create a struct.

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

struct book {
  char title[20];
  float price;
  char author[20];
};

int main() {
  struct book c_programming;
  strcpy(c_programming.author, "Ravi Shankar");
  strcpy(c_programming.title, "C Programming");
  c_programming.price = 999;

  printf("Books details are: \nTitle: %s\nAuthor: %s\nPrice: Rs %.2f\n",
c_programming.title, c_programming.author, c_programming.price);
  return 0;
}
```

As you can see, we created a data type called book. And we created a variable called c_programming of this data type. And individual members of c_programming are accessed by using a dot (.) notation.

At the beginning, structures seems a bit overwhelming. But, with time, it becomes very easy. This topic is very important, because structures are heavily used when we implement data structures using C/C++.

ARRAY OF STRUCTURES

The way we create arrays of other data types, we can also create arrays of structs. Let's see an example to understand this.

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

struct book {
  char title[20];
  float price;
  char author[20];
};

int main() {
  struct book my_books[2];

  for(int i = 0; i < 2; i++) {
    printf("Enter title, author and price of the book");
    scanf("%s %s %f", my_books[i].title, my_books[i].author, &my_books[i].price);
  }

  for(int i = 0; i < 2; i++) {
    printf("Books details are: \nTitle: %s\nAuthor: %s\nPrice: Rs %.2f\n",
my_books[i].title, my_books[i].author, my_books[i].price);
  }

  return 0;
}
```

STRUCTURE DECLARATION

When we declare structure, it doesn't reserve any space in memory. A space in memory is only reserved when we declare a variable of the above structure type.

The following, will only define the form of a user defined data type:

```
struct book {
  char title[20];
  float price;
  char author[20];
};
```

While declaring structures, we can also declare variables of this data type. Let's see an example:

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

struct book {
  char title[20];
  float price;
  char author[20];
} book1, book3, *book3;

struct {
  char name[20];
} nameStruct;
```

As you can see that, while declaring structure book, we also declared three two variables and one pointer of type 'struct book'.

The second struct that we declared, we didn't give it a name. But, we created a variable, 'nameStruct' of this type.

NESTING OF STRUCTURE

Well, a structure can have a variable of type structure which can again have a variable of type structure and so on. Let's understand through an example.

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

struct Address {
  char village[20];
  char state[20];
};

struct Person{
  char name[20];
  struct Address address;
};
```

STRUCTURE COPY

A structure can be copied into another structure of same type in two ways, piece-meal or at one shot. Let's see an example:

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

struct book {
```

```c
  char title[20];
  float price;
  char author[20];
};

void displayBookInfo(struct book b) {
  printf("Books details are: \nTitle: %s\nAuthor: %s\nPrice: Rs %.2f\n", b.title, b.author,
b.price);
}

int main() {
  struct book book1, book2, book3;
  strcpy(book1.author, "Ravi Shankar");
  strcpy(book1.title, "C Programming");
  book1.price = 999;

  book2 = book1;

  // book3.author = book1.author; // This will cause error
  strcpy(book3.author, book1.author);
  strcpy(book3.title, book1.title);
  book3.price = book1.price;

  printf("\nDisplaying details of book1\n");
  displayBookInfo(book1);

  printf("\nDisplaying details of book2\n");
  displayBookInfo(book2);

  printf("\nDisplaying details of book3\n");
  displayBookInfo(book3);


  return 0;
}
```

The output would be:

```
Displaying details of book1
Books details are:
Title: C Programming
Author: Ravi Shankar
Price: Rs 999.00

Displaying details of book2
Books details are:
Title: C Programming
Author: Ravi Shankar
Price: Rs 999.00
```

```
Displaying details of book3
Books details are:
Title: C Programming
Author: Ravi Shankar
Price: Rs 999.00
```

PASSING STRUCTURE TO A FUNCTION

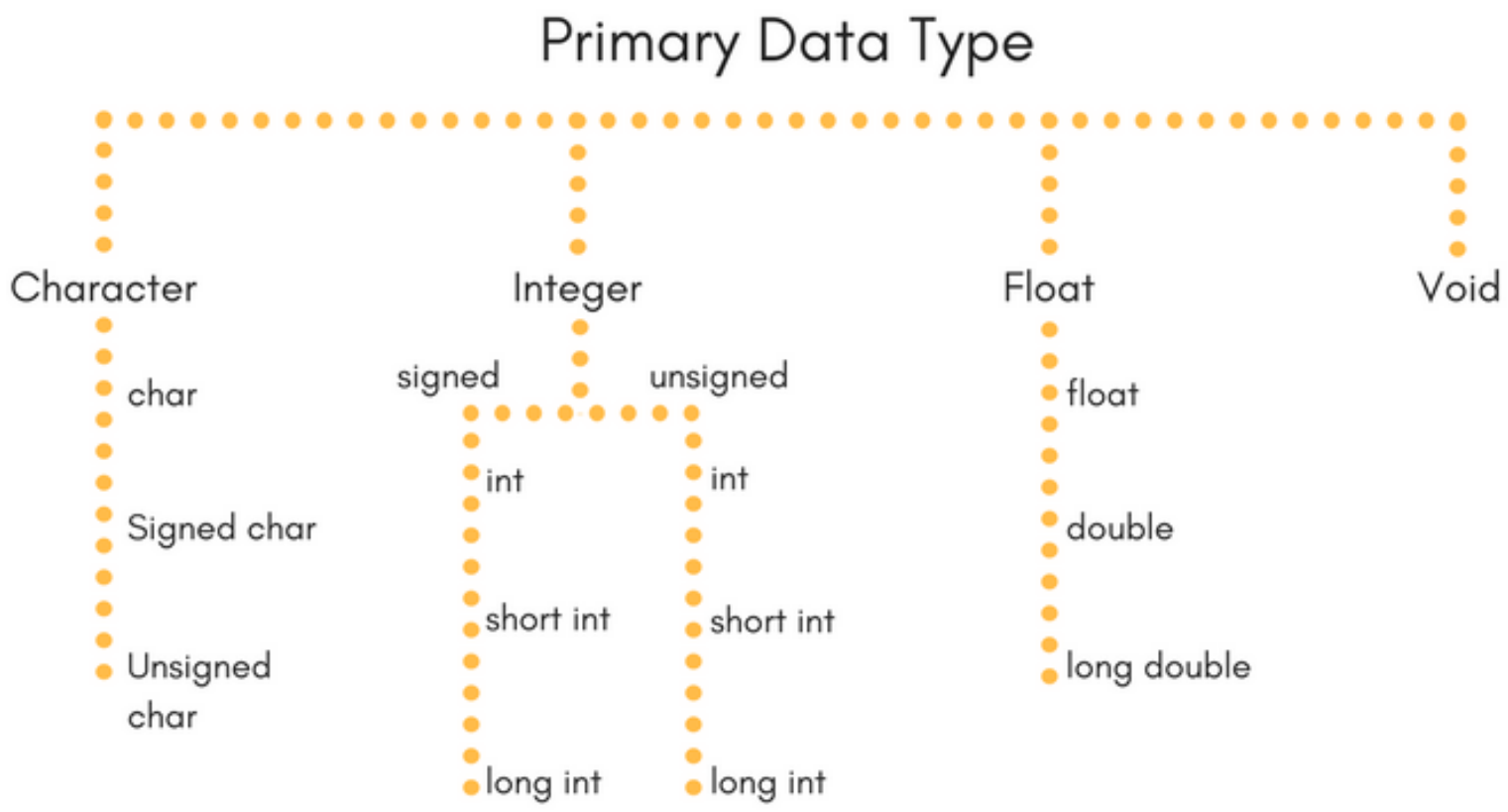
The way we pass normal variables to a function, struct can also be passed. Please refer to the example in the last section to understand it better.

10. DATA TYPES IN C

In this chapter, we will get introduced to data types in C. Data in C is basically categorised into two parts, primary data types and secondary data types.

PRIMARY DATA TYPES

There are four types of primary data types available: Integers, characters, reals and void.



SECONDARY DATA TYPES

Secondary data type basically includes arrays, pointers, structures and enums. They are all being dealt at length in different chapters on this notes.

STORAGE CLASS IN C

We don't need to specify storage class of a variable in C because the defaults are automatically implied. But, for any cases where we want non-defaults, we have to specify. A storage class gives the influence the following properties of data on which it is specified.

- Storage
- Initial value
- Scope
- Lifetime

There are four storage class available in C. For brevity, we will study them by comparing them in a table.

Storage Specifier	Storage	Initial Value	Scope	Life
auto	stack	Garbage	Within block	End of block
extern	Data Segment	Zero	Global Multiple files	Till end of program
static	Data Segment	Zero	Within block	Till end of program
register	CPU register	Garbage	Within block	End of block

If nothing is specified, then auto storage class is implied. Let's look at a program to understand the syntax.

```
#include<stdio.h>
void registerStorageClassExample() {
  register char b = 'A';
  printf("Value of the variable 'b' declared as register: %d\n", b);
}

int main() {
  registerStorageClassExample();
  return 0;
}
```

11. BITS AND BIT OPERATIONS

Computers don't understand human language. All they understand is 0s and 1s. They don't understand even the numbers like 10 or 20. Everything is converted into 0s and 1s before computer could store it or save it. In this chapter, we will learn how numbers and converted to their binary form(0s and 1s), and how binary numbers are converted to decimals and hexadecimals (explained later).

BINARY, DECIMAL, OCTAL AND HEXADECIMAL NUMBERS

The numbers which we use in daily life have 10 digits, 0 to 9. Since there are 10 digits in total, this system of number is called decimal number. The computers converts these numbers to 0 and 1, they use a total of 2 digits, so this system is called binary system. There is a system of numbers that use just 8 and 16 digits, 0 to 7 and 0 to F respectively, and these are called octal and hexadecimal respectively.

Examples:

Binary $1100_{(2)}$ -> $12_{(10)}$ in decimal.

Octal $17_{(8)}$ -> $15_{(10)}$ in decimal

Hexadecimal $FF_{(16)}$ -> $256_{(10)}$ in decimal

UNDERSTANDING THE DIFFERENT BASES OF NUMBERS

Let's take an example os $2145_{(10)}$. It cal also be written as

$2145_{(10)} = 2000 + 100 + 40 + 5$

$2145_{(10)} = 2 * 1000 + 1 * 100 + 4 * 10 + 5 * 1$

$2145_{(10)} = 2 * (10 \hat{} 3) + 1 * (10 \hat{} 2) + 4 * (10 \hat{} 1) + 5 * (10 \hat{} 1)$

You can see that every digit in 2145 is multiplied with certain powers of 10 to get the value of final numbers. The reason why they are multiplied by a power of 10 is generally not obvious.

Remember how we write numbers in incremental order, we start from 1 and go on till 9. Once we reach 9, we make the unit's digit as 0 and prepend it with 1, which makes it 10. We

again start from 10 and move till 19, on reaching 19, we change the unit's digit to zero and increment the ten's digit by 1. Again after 10 steps, the digits in units place will become 0 and the digit in tens place will increase by 1.

Another way to look at this is that, if the digits in units place increase by 1, overall increment in number is one. But, if the digits in tens place increase by one, overall increment in the number is 10. Similarly, every increment in hundreds place amount to an increment of 100 to the overall number.

This is so obvious. right?

Can there be a system where every increment in the number next to units place doesn't amount to an increase of 10? The answer is yes.

Let's understand octal system. Here, the counting starts from 1 and go till 7. On reaching 7, the rightmost digit becomes 0 and the we add one to the left of right most digit.

Let's write a few continuous numbers in octal system starting from 0.

0, 1, 2, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14, 15, 16, 17, 20, 21, 22, 23, 24, 25, 26, 27, 30, 31, 32

Here,

How many places apart are 20 and 10? The answer would be 8.

How many places apart are 31 and 21? The answer would be 8.

See, for every increment in 2nd digit from right, the overall number increases by 8. Similarly, for every increment in 3rd digit from right, the overall number increases by 64 i.e 8^2 So, this number said to be in base 8.

OCTAL TO DECIMAL

Now, the question is, how to get the absolute value of the number written in octal format? The way we get absolute value of decimal number by multiplying every digit by a power of 10, we get absolute value of octal number by multiplying every digit by a power of 8.

Let's see how we get value $2501_{(8)}$ in decimal.

$$2501_{(8)} = 2 * 8^3 + 5 * 8^2 + 0 * 8^1 + 1 * 8^0$$

$$2501_{(8)} = 2 * 512 + 5 * 64 + 0 * 8 + 1 * 1$$

$2501(_8) = 1024 + 320 + 0 + 1$

$2501(_8) = 1355(_{10})$

HEX TO DECIMAL

Let's convert $1AF(_{16})$ to decimal.

$1AF(_{16}) = 1 * 16^2 + A * 16^1 + F * 16^0$

$1AF(_{16}) = 1 * 16^2 + 10 * 16^1 + 15 * 16^0$ (converting F to 15 and A to 10)

$1AF(_{16}) = 1 * 256 + 10 * 16 + 15 * 1$

$1AF(_{16}) = 256 + 160 + 15$

$1AF(_{16}) = 431(_{10})$

BINARY TO DECIMAL

Let's convert 110101111 to decimal.

$110101111(2) = 1*2^8 + 1*2^7 + 0*2^6 + 1*2^5 + 0*2^4 + 1*2^3 + 1*2^2 + 1*2^1 + 1*2^0$

$110101111(2) = 1*256 + 1*128 + 0*64 + 1*32 + 0*16 + 1*8 + 1*4 + 1*2 + 1*1$

$110101111(2) = 256 + 128 + 0 + 32 + 0 + 8 + 4 + 2 + 1$

$110101111(2) = 431(_{10})$

0 TO 15 IN BINARY, OCTAL AND HEX

Decimal	Binary	Octal	Hex
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4

Decimal	Binary	Octal	Hex
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

The above table may sound intimidating on the first reading, but with time, this will become obvious.

OCTAL TO BINARY

Octal to binary conversion is very simple. Take each digit of an octal number and replace it by **binary equivalent in a group of three**. Example: 321

$321_{(8)} = 011\ 010\ 001_{(2)}$

$321_{(8)} = 11010001_{(2)}$

HEX TO BINARY

Hex to binary conversion is also simple just like octal. Take each digit of a hex number and replace it by **binary equivalent in a group of four**. Example: 321

$1AF_{(16)} = 0001\ 1010\ 1111_{(2)}$

$1AF_{(16)} = 110101111_{(2)}$

CONVERTING DECIMAL TO BINARY

We converted Octal and Hex to binary using a trick. But, that trick will not work for decimal. Here, we will have to do using the generic way. To convert any number(in any base)

in binary form, we have to represent it in such a form that it is represented in power of 2. Example:

$12(_{10}) = 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 0 * 2^0$

Once, the number are represented in the above format, take the coefficients and ignore the other parts. So, in the above case, we will pick 1100 and ignore the other parts.

DECIMAL TO BINARY (LONG DIVISION METHOD)

Well, being able to represent number in power of 2 is not always easy. So, we will take help of long division method.

Divisor(2)	Divident(Number)	Remainder
2	13	1
2	6	0
2	3	1
2	1	1
	0	

Here, we will pick the remainder on the right column (From bottom to top) to get the binary number. In case of 13, that would be 1101.

HEX TO BINARY(USING LONG DIVISION)

Divisor(2)	Divident(Number)	Remainder
2	1AF	1
2	D7	1
2	6B	1
2	35	1
2	1A	0
2	D	1
2	6	0
2	3	1
1	1	1
	0	

Taking remainders (rightmost column) from bottom to top, we get $110101111(_2)$, which is the binary equivalent of $1AF(_{16})$.

Hex to binary conversion is not so simple using long devision. When we have to carry a digit, it is getting multiplied by 16 rather than 10 as in standard long division that we are used to. We have added this section just for understanding.

C PROGRAM FOR DECIMAL TO BINARY AND VICE-VERSA

Since, this is a C programming notes, we should automate everything using a C program. We will see conversion of Decimal to binary and vice versa. Other conversions are a bit complex and we have left it for enthusiastic students to do on their own.

Let's look at a program that converts a decimal number to its binary form

```
/**
 * C program for converting decimal to binary
 * Apna Vidyalaya
 */

#include<stdio.h>

int main() {
  int binaryEquivalent = 0;
  int decimalEquivalent = 0;
  printf("Enter the number you want to convert to binary: ");
  scanf("%d", &decimalEquivalent);
  int tempDecimal = decimalEquivalent;
  int indexMultiplier = 1;
  while(tempDecimal) {
    int temp = tempDecimal % 2;
    binaryEquivalent = binaryEquivalent + indexMultiplier * temp;
    indexMultiplier *= 10;
    tempDecimal /= 2;
  }
  printf("%d in decimal when converted to binary looks like %d\n", decimalEquivalent,
binaryEquivalent);
  return 0;
}
```

Now, lets look at a program that converts binary to its decimal equivalent

```
/**
 * C program for converting binary to decimal
 * Apna Vidyalaya
 */

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

```c
int main() {
  int binaryEquivalent = 0;
  int decimalEquivalent = 0;
  printf("Enter the number you want to convert to decimal: ");
  scanf("%d", &binaryEquivalent);
  int tempBinary = binaryEquivalent;
  int powerMultiplier = 0;
  while(tempBinary) {
    int temp = tempBinary % 10;
    decimalEquivalent = decimalEquivalent + temp * pow(2, powerMultiplier);
    powerMultiplier += 1;
    tempBinary /= 10;
  }
  printf("%d in binary when converted to decimal looks like %d\n", binaryEquivalent,
decimalEquivalent);
  return 0;
}
```

Both, the above programs are self-evident and probably don't require explanation.

REPRESENTATION OF NUMBERS IN MEMORY

We will just talk about integers, for others data types please refer online sources. Many of the modern day computers use 4 byte for integers, but for simplicity we will represent int as 1 byte in our examples.

Positive integers, irrespective of how many digits they have in their binary form, will occupy 1 byte(i.e 8 bits). The bit positions which are not occupied, will be padded with 0.

Example:

$12_{(10)}$ is equivalent to $1100_{(2)}$ in its binary form. But, in memory $12_{(10)}$ will be represented at $00001100_{(2)}$.

Negative numbers are generally represented by using 2's complement method. In this method, we first write the number is its positive form, then take reverse each bit, and then add one to the number. Let's see for $-12_{(10)}$

$12_{(10)} = 00001100_{(2)}$

After reversing all the bits, we get 11110011

After adding 1 to the output, we get 11110100.

So, $-12_{(10)} = 11110100_{(2)}$

The obvious question might be, how do we know the sign of a number in binary form. Well, the most significant bit represents the sign of a number.

CONVERTING FROM 2'S COMPLEMENT TO DECIMAL

In the last section, we have already seen how to convert decimals to 2's compliment. In this section we will learn the reverse.

In 2's complement. the most significant bit is negative bit, and all other bits are positive bits. So, if the most significant bit is non-zero(i.e. 1), then you have to subtract its value from the overall number. Example: $11110100_{(2)}$.

Since the most significant bit is 1, its value will be subtracted and values of all other bits will be added. Let's calculate it step by step for more clarity.

$11110100_{(2)} = -1*(2^7) + 1*(2^6) + 1*(2^5) + 1*(2^4) + 0*(2^3) + 1*(2^2) + 0*(2^1) + 0*(2^0)$

$11110011_{(2)} = -128 + 64 + 32 + 16 + 0 + 4 + 0 + 0$

$11110011_{(2)} = -128 + 116$

$11110011_{(2)} = -12_{(10)}$

For positive numbers, since the most significant bit is zero, the number to be subtracted becomes zero, hence the calculation is a bit simpler.

INTRODUCTION TO BIT OPERATIONS

We played a lot with bits in this chapter. It is time now to take the game a bit deeper. We will see how we can manipulate the bits of a number using c program. Let's look at the representation of 12 in binary form.

$12_{(10)} = 00001100_{(2)}$

If we, flip the second bit from right, what is the number that we will get? The answer is 14. Can it be done using C programming? Of course, yes. This is the type of operations that we would be doing in the rest of the chapters.

But, for that let is get introduced to bitwise operators.

BITWISE OPERATORS

Operators	Meaning
~	One's complement
>>	Right shift
<<	Left shift
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR(Exclusive OR)

These operators can be applied to integers and characters but not on float and doubles.

ONE'S COMPLEMENT OPERATOR

You remember the binary representation of 12? What will it become if we were to flip all the bits. On almost all the computers, it should become -13.

$12_{(10)} = 00001100_{(2)}$

After Flipping all the bits, it will become

$11110011_{(2)}$

After converting it to decimal we get,

$11110011_{(2)} = -13_{(10)}$

Let's write the code for bit flipping.

```c
#include<stdio.h>

int main() {
  int b = 12;
  int bitFlipped = ~b;
  printf("%d becomes %d applying ones complement operator\n", b, bitFlipped);
  return 0;
}
```

SHIFTING BITS LEFT AND RIGHT

If we shift all the bits of a number towards left using << operator, then the bits on the left falls off and 0s are added on the right side for the bits that falls off.

Example:

if A = 11001101

and we shift bits towards left using A = A << 1, then A would become 10011010.

Similarly, if we shift bits towards right using >> operator, then the bits on the right falls off and bits are added on the left for the bits that falls off. Here, the bits added on the left depends on the sign bit. If, the sign bit is zero then 1 will be added and if the sign bit is zero, then 0 will be added. Example:

If A = 11001101 and B = 01100111

and we shift bits towards right using

B = B >> 1 and A = A >> 1

then A and B will become

A = 11100110 and B = 00110011

Note that, if the value of the right operand is negative or greater than the number of bits present in the left operand, then the behaviour is undefined. The compiler may choose to define certain behaviour in this case, but, we can't rely on this functionality.

```c
#include<stdio.h>

int main() {
  int b = 1;
  int s = 2;
  int c = b<<s;
  printf("%d\n,", c);
  //output should be 4
}
```

BITWISE AND, OR, XOR

If we perform bitwise AND, OR, XOR on two BITs, then we get the following results.

|	0	1
0	0	1
1	1	1

&	0	1
0	0	0
1	0	1

^	0	1
0	0	1
1	1	0

If we perform bitwise & or | of ^operation between any two numbers, then their BITs are ANDed ORed of XORed individually according to the above table, and the results thus obtained is returned.

GETTING VALUE AT A PARTICULAR BIT POSITION OF A NUMBER.

Binary representation of 1 is 0000 0001. If we perform left shift on 1 by 2 we get 0000 0100.

If we perform & between any other number and this number, then the output thus returned will be the BIT value of the third bit from right of that number. If the third(from right) bit is non-zero then the output would be non-zero otherwise the output would be zero.

Using this trick, we can get bit value of any bit present at any position of an integer. Let's see a function that prints BIT values of any given number.

```
#include<stdio.h>

void displayBits(int n) {
  printf("Bit representation of %d is: ", n);
  for(int i = 7; i >= 0; i--) {
    int mask = 1 << i ;
    int isBitOn = n & mask;
    char bitVal = isBitOn ? '1': '0';
    printf("%c", bitVal);
  }
  printf("\n");
}

int main() {
  int d = 10;
  displayBits(d);
  return 0;
}
```

12. INPUT/OUTPUT

C Language doesn't have any provision for input and output. But, it does provide a standard library for handling I/O operations. The I/O functionalities can be divided into two parts, Console I/O and File I/O. In this chapter, we intend to get briefly introduced to them.

CONSOLE I/O

Console I/O deals with input and output that involves active participation from user, ie screen and keyboard. Console I/O is further divided into two parts, Formatted I/O and unformatted I/O.

FORMATTED CONSOLE I/O

Formatted I/O as the name suggests, deals with input and output where the data is formatted according to the requirement. They let us provide input in a fixed format and they output the data in a fixed format. They also gives us functionality to handle the I/O formatting. Basically the function printf() and scanf() falls under this category.

PRINTF()

The general form of printf looks like this:

```
printf("format string", list of variables);
```

The format string can contain characters, format specifier such as %d and escape sequences. Let's see an example of printf(), and then we will understand in detail.

```
#include<stdio.h>

int main() {
  printf("I study at %s at %s\n", "C Programming", "Apna Vidyalaya");
  return 0;
}
```

As you can see, the above printf() is called by a format string and a list of arguments. The format string contains characters, format specifiers(%s) and escape sequence(\n).

During execution, The characters are printed from left to right until a format specifier or an escape sequence is encountered. If it is an escape sequence then appropriate action is taken. Such as for \n, the cursor will be put on the start of next line. If it is a format specifier, then

it will be replaced by the arguments provided. The replacement of format specifier with arguments happen sequentially, in the order in which they occur from left to right.

Let's have a look at the format specifiers available in C.

Data Type		Format specifier
Integer	short signed	%d or %l
	short unsigned	%u
	long signed	%ld
	long unsigned	%lu
	unsigned hexadecimal	%x
	signed octal	%o
Real	float	%f
	double	%lf
	long double	%Lf
Character	signed character	%c
	unsigned character	%c
String		%s

Well, apart from these we can also specify some optional format specifiers.

Specifier	Description
w	Specify width of the field
.	Separates the field from precision using decimal
d	specifies the digits after decimal position
-	used to justify the field towards left in specified field width

Let's see an example to understand the optional format specifiers.

```
#include<stdio.h>

int main() {
  int numberOfStudents = 56;
  printf("There are %d students in the class\n", numberOfStudents);
  printf("There are %2d students in the class\n", numberOfStudents);
  printf("There are %4d students in the class\n", numberOfStudents);
  printf("There are %6d students in the class\n", numberOfStudents);
  printf("There are %-6d students in the class\n", numberOfStudents);
  printf("There are %1d students in the class\n", numberOfStudents);
  return 0;
}
```

The output would be:

```
There are 56 students in the class
```

```
There are 56 students in the class
There are   56 students in the class
There are     56 students in the class
There are 56     students in the class
There are 56 students in the class
```

Let's see another program that deals with float.

```c
#include<stdio.h>

int main() {
  printf("Average marks of students are:\n");
  printf("%5.2f %5.2f\n", 60.0, 56.0);
  printf("%5.2f %5.2f\n", 65.908, 45.33);
  return 0;
}
```

The output would be:

```
Average marks of students are:
60.00 56.00
65.91 45.33
```

We can also specify width specifier for strings. That is left as an assignment for reader to try.

ESCAPE SEQUENCES

We have been using newline character (\n) since the beginning of this book. We have been using this to introduce a new line in text being outputted from printf(). The backslash (\) is called an escape character - it causes an escape from normal interpretation of the characters and treats the next character as a special character. The character coming after (\) have a special meaning. In the case of newline, 'n' which appears after '\' indicates new line. There are other escape sequences too. Let's look at them.

Escape sequence	Description	Escape sequence	Description
\n	new line	\f	form feed
\t	tab	\a	alert
\r	carriage return	\'	single quote
\b	back space	\"	double quote
\\	back slash		

What if you want to output '\' on the terminal? Since, '\' is an escape character, it can't be outputted directly. Hence, we need to use it with an escape sequence, (\\). In this escape

sequence, since the second backslash is coming after an escape character, it assumes its normal behaviour.

UNFORMATTED CONSOLE I/O

Till now in this chapter, we have been learning about formatted I/O. In this section we will learn about few library functions that performs unformatted I/O.

getchar() and putchar(): getchar() receives a single character and putchar prints a single character on the console. Note that, a character has to be followed by enter to be read by getchar()

gets() and puts(): they are similar to getchar() and putchar(), with one difference that they read and write strings. Also note that, gets() will read the line of text only after enter has been pressed.

FILE I/O

If we think objectively file I/O, then then there are only a few things that we need to do with a file. They are:

- create a file
- Open a file
- read from a file
- write to a file
- navigate inside a file
- delete a file
- close a file

We will briefly learn a few of the above operations in this section.

OPENING AND CLOSING AND READING FROM A FILE

Let's see a program for opening and reading a file. But, before we do that, lets create a file called "vidyalaya.txt" with the following text in it:

"My favourite book on C is "The C Programming Language" by Dennis M. Ritchie and Brian W. Kernighan"

Once you are done, you can run this program to see the above text printed on the console.

```
#include<stdio.h>

int main() {
  FILE *fp;
  fp = fopen("vidyalaya.txt", "r");
  while(1) {
    char c = fgetc(fp);
    if(c == EOF) {
      break;
    }
    printf("%c", c);
  }
  fclose(fp);
  printf("\n");
  return 0;
}
```

FILE is a structure provided in stdio library. When we need to open a file, few data about the file need to be maintained, and we do that is a variable which is of type FILE. In our case, we declared a FILE pointer fp to store file information.

fopen() is a standard library function for opening a file. It takes first argument as the file name and second argument as the mode in which file needs to be opened. We will use "r" and "w" modes for programs in this notes. We won't talk much about the other modes in this notes. If you interested, consult online sources.

fgetc() is used to read one character, after reading the character, it moves the pointer one step ahead.

EOF is a macro that indicated end of file.

fclose() is a standard library function that closes a file. It frees up the resources put to maintain details about the file being opened.

WRITING TO AND DELETING A FILE

Let's see a program that write to a file line by line. Once you run the below program, you wil see that a new file is created with some text it it.

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

```c
int main() {
  FILE *fp;
  fp = fopen("apna.txt", "w");
  char arr[100] = "I revise C from 'Notes on C Programming' by Ravi Shankar";
  char *temp = arr;
  while(*temp) {
    fputc(*temp, fp);
    temp++;
  }

  fclose(fp);
  printf("\n");
  return 0;
}
```

Now, let's see a program that delete a file. Note that the file will be deleted permanently and probably you won't be able to recover it. Before you run this program, create a file called "abc.txt" with text of your choice, in the folder where you execute your c programs.

```c
#include <stdio.h>

int main() {
  if (remove("abc.txt") == 0) {
    printf("Deleted successfully");
  } else {
    printf("Unable to delete the file");
  }
  return 0;
}
```

This program will delete the file "abc.txt".

MISCELLANEOUS

sprintf() and sscanf() are two functions which are very similar to printf() and scanf(), with one difference being that they write and read to and from character arrays.

13. PREPROCESSOR AND BUILD PROCESS

The C preprocessor is a macro preprocessor that preprocesses the code before it foes for compilation. It can do things like expanding macros and inclusion of files.

MACROS

There is nothing that can be done with macros and not without macros, but it offers a convenience. Let's look at the following code to see macros in action.

```c
#include<stdio.h>

#define PI 3.14

int main() {
  float radius = 3.0;
  float area = PI * radius * radius;
  printf("Area is: %.3f\n", area);
  return 0;
}
```

Here, you can see that we have defined a macro PI. Let's say we want to use a value of PI tomorrow that is more precise, then we will have to change the value at just one position and all the occurrences will be automatically updated. Well, it is possible to achieve the same effect with global variables, but with global variables, we have a chance of mistakenly corrupting the original value. Macros are more powerful that shown in the example above, lets see one more example.

```c
#include<stdio.h>

#define AREA(x) (3.14 * x * x)

int main() {
  float radius = 3.0;
  float area = AREA(radius);
  printf("Area is: %.3f\n", area);
  return 0;
}
```

Macros can also receive variables and expand accordingly. Note that, during the process of preprocessing, all the macros are replaces by its value. The code which goes for compilation, doesn't contain any macros.

INCLUDING FILES

During preprocessing, the files which we include in our source code, and included. There are two ways to include files in C. First using <> and second using ""; If you include files using <>, C looks for the file at standard location. But, if you include file using "", C will expand its search space to also look into the directory where you are executing the program and the paths provided. Let's look at a program that includes another program.

```
/*
* area.c
*/

float area(int radius) {
  return 3.14 * radius * radius;
}

/*
* area.h
*/

float area(int radius);

/*
include-area.c
*/

#include<stdio.h>
#include "area.h"

int main() {
  int radius = 3;
  float totalArea = area(radius);
  printf("Total area is: %.3f \n", totalArea);
  return 0;
}
```

We have created three files, include-area.c, area.c and area.h

On my system, I use GCC. so I compile the two c files together using the command "gcc include-area.c area.c". You might need to find out in your IDE/editor settings how to compile and link multiple source files together.

CONDITIONAL COMPILATION

Six directives are available to enable conditional compilation in C. They are:

- #if
- #ifdef
- #ifndef
- #else
- #elif
- #endif

#IFDEF, #IFNDEF, #ENDIF

Let's see an example using this feature.

```
#include<stdio.h>
#define MACBOOK
// #define WINDOWS
int main() {

  #ifdef MACBOOK
    printf("Running on MacBook\n");
  #endif
  #ifndef WINDOWS
    printf("Not Running on Windows\n");
  #endif
  return 0;
}
```

Lets see another similar program

```
#include<stdio.h>
// #define MACBOOK
int main() {

  #ifdef MACBOOK
    printf("Running on MacBook\n");
  #else
    printf("Not Running on MacBook\n");
  #endif
  return 0;
}
```

#IF, #ELIF AND OTHER DIRECTIVES

#If and #elif are similar to #ifdef, but with a difference that they test a conditional

```
#include<stdio.h>
#define number 5
int main() {

  #if number == 5
    printf("Number = 5\n");
  #elif number == 7
    printf("Number = 7\n");
  #else
    printf("Number not equal to 5 and 7\n");
  #endif
  return 0;
}
```

Please note that you can only use a macros for condition testing in #if or #elif directive.

There is also an another way to test whether a macro is defined or not, that is by using define directive. Lets see an example:

```
#include<stdio.h>
#define number
int main() {
  #if defined(number) && !defined(WINDOWS)
    printf("Number on non-windows system\n");
  #endif
  return 0;
}
```

#UNDEF AND #PRAGMA

If a macro has been defined and we want to undefine it, then #undef comes to our rescue.

#pragma is a special purpose directive that turns feature on and off. For example: disabling warning in certain cases.

THE BUILD PROCESS

The build process happens in four steps in C language. Let's walk through each part of the step.

In the first step is called preprocessing. In this step, first the preprocessor removes comments and joins lines ending with (\).

Then, lines starting with # are interpreted as processor commands. A new language semantics is introduced in this step. This is basically done to avoid repetition of source code by providing functionality to inline files, define macros and conditionally omit code.

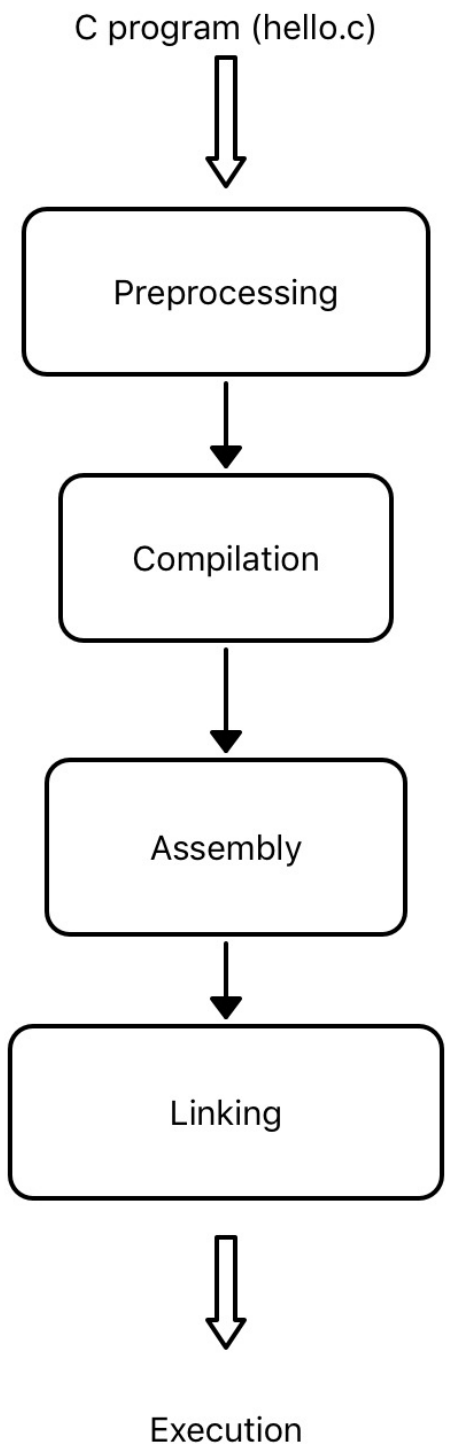
If you want to see the output of this step, pass an argument -E to the compiler. In out case, the program is named hello.c, so, we will run the following command to see the output of preprocessor step.

gcc -E hello.c

Second step is compilation step. In this step, preprocessed code is converted to assembly code specific to the process architecture of the target machine. Assembly code is a human readable code. To see the output of this step, we pass -S option to the compiler. In our case, the command is:

gcc -S hello.c

Third step is conversion of assembly code to object code. Object code is basically machine instruction to be run on the target machine. To get the output of assemble stage, we have to pass -c to the compiler. In our case the command is:



gcc -c hello.c

Fourth stage is linking. The object code generated in the assembly stage consists of machine instructions to be run on target machine, but it contains some missing or out of order pieces. The linking step rearranges the machine instructions and fill in the missing pieces so that the program can execute. This is the final stage and the output of this stage is an executable file. When we compile a c program without any option, a file called a.out is generated. But, we can pass an option -o followed by target executable file name to the compiler to save the output of this step to a file. In our case the command is:

gcc -o hello-demo hello.c

This command will create a file called hello-demo, which is an executable file.

14. MISCELLANEOUS

There are multiple functionalities of C which we didn't cover. We will be covering a few of them them here in this chapter.

ENUMS

Enums are a way to define user data types. For example, we can define a data type called marital_status which only receives married, unmarried, divorced as the value. Let's see an example:

```
#include<stdio.h>

int main() {
  enum marital_status {
    married,
    unmarried,
    divorced
  };
  enum marital_status person_status;
  person_status = married;
  if (person_status == married) {
    printf("Person is married\n");
  } else {
    printf("Person is either unmarried or divorced\n");
  }
  return 0;
}
```

Here, first we defined a data type called "enum marital_status", then we declared a variable of this data type, called person_status. It should be noted that internally person_status is treated an an integer, so, you can give it some integer value also. Also note that when we define enumerated data types, each list of permissible values gets assigned an integer starting from 0 onwards. If is possible to override the default assignment by giving our own values to enumerations. Let's see an example

```
#include<stdio.h>

int main() {
  enum marital_status {
    married = 200,
    unmarried,//it will be automatically given 201, ie 1 more than married
    divorced = 500,
    separated//it will get 501
  };
```

```
  enum marital_status person_status;
  printf("Integer values of enumerations are:\n");
  printf("married: %d\n", married);
  printf("unmarried: %d\n", unmarried);
  printf("divorced: %d\n", divorced);
  printf("separated: %d\n", separated);
  return 0;
}
```

The output would be:

```
Integer values of enumerations are:
married: 200
unmarried: 201
divorced: 500
separated: 501
```

UNION

Union can be thought of as container for multiple data types. A union with fields of data type X and data type Y represents a single data that is either X or Y. If we define a union with an int and char, it will have a single data in memory and the same data can be accessed as char or int variable. It sounds confusing at first, but will become clear with time. Let's look at an example:

```
#include<stdio.h>

union test_data {
  int x;
  char y;
};

int main() {
  union test_data test;
  test.x = 65;
  printf("The value of char test.y is: %c\n", test.y);
  return 0;
}
```

The output is:

```
The value of char test.y is: A
```

In the example above, we declared a variable called test which is of union test_data type. When the variable is declared, C will find out which field among the union fields require the

maximum space, and then reserve that much memory for the variable. In our case, int requires 4 byte, and char require 1 byte, so test variable will be given 4 bytes in memory.

All the fields of union, in our case x and y will refer to the same memory. When we assigned 65 to integer variable, and accessed it using char variable, the character 'A' got printed, whose ASCII value is 65.

TYPECASTING

Converting a data of one type into another type is called typecasting. For example, an int can be converted to float or a char can be converted to int etc. There are two types of casting available in C. Implicit type casting and explicit type casting.

IMPLICIT TYPECASTING

When we convert data from one type to another compatible type such that there is no data loss, then it is called implicit typecasting. Assigning an int to a variable of type float is implicit.

```
#include<stdio.h>

int main() {
  float myFloat;
  int myInt = 8;
  myFloat = myInt;
  printf("Value of myFloat is: %f\n", myFloat);
  return 0;
}
```

In the example above, an integer can be assigned to a variable of type float and so this is an implicit typecasting.

EXPLICIT TYPECASTING

When we typecast one data type to another data type where data or a part of data may be lost, we do explicit typecasting. Example, from an int to short.

```
#include<stdio.h>

int main() {
  short i;
  int myInt = 70000000;
```

```
  i = (short)myInt;
  printf("Value of i is: %d\n", i);
  return 0;
}
```

TYPEDEF

Typedef allows us to define an alias data types in C. For example, if we using using long long a lot in our program, then we can create an alias LL for long long, and it can make out code simpler. Let's see an example.

```
#include<stdio.h>

typedef long long LL;
struct node {
  LL data;
};

int main() {
  typedef struct node N;
  N myNode;
  myNode.data = 5;
  printf("myNode.data is: %lld \n", myNode.data);
  return 0;
}
```

For structures and other derived data types, we can create an alias at the time of declaration also. See an example.

```
#include<stdio.h>

typedef struct node {
  int data;
} NODE;

int main() {
  NODE myNode;
  myNode.data = 25;
  printf("myNode.data is: %d \n", myNode.data);
  return 0;
}
```

15. PROJECTS

In this section, we will look at few console projects

CALCULATOR PROGRAM

Below is the code for a very simple calculator program.

```
#include<stdio.h>
enum operation {
  SUBTRACTION = 0,
  ADDITION =  1,
  MULTIPLICATION = 2,
  DIVISION = 3
};

int main() {
  enum operation userOperation;
  int operand1, operand2, result;
  printf("\nWelcome to our Calculator project!\n\n");
  printf("Enter %d for Subtraction\n", SUBTRACTION);
  printf("Enter %d for Addition\n", ADDITION);
  printf("Enter %d for Multiplication\n", MULTIPLICATION);
  printf("Enter %d for Division\n", DIVISION);
  printf("Enter your choice: ");
  scanf("%d", &userOperation);

  switch (userOperation)
  {
    case SUBTRACTION:
      printf("Enter two numbers: ");
      scanf("%d%d", &operand1, &operand2);
      result = operand1 - operand2;
      break;
    case ADDITION:
      printf("Enter two numbers: ");
      scanf("%d%d", &operand1, &operand2);
      result = operand1 + operand2;
      break;
    case MULTIPLICATION:
      printf("Enter two numbers: ");
      scanf("%d%d", &operand1, &operand2);
      result = operand1 * operand2;
      break;
    case DIVISION:
      printf("Enter two numbers: ");
      scanf("%d%d", &operand1, &operand2);
      result = operand1 / operand2;
      break;
```

```
    default:
      printf("You entered wrong choice for operation, please run the program again\n");
      return 0;
      break;
  }

  printf("Output is: %d\n\n", result);
  return 0;
}
```

TIC TAC TOE GAME

```
#include<stdio.h>
enum gameoutput {
  progress,
  draw,
  over
};
int playerTurn = 1;
enum gameoutput output = progress;
int playerWon = -1;

char matrix[3][3] = {
  {'_', '_', '_'},
  {'_', '_', '_'},
  {'_', '_', '_'}
};

void displayBoard() {
  for(int i = 0; i < 3; i++){
    for(int j = 0; j < 3; j++) {
      printf("%c  ", matrix[i][j]);
    }
    printf("\n\n");
  }
}

int isValidMove(int row, int col) {
  if (!(row >= 0 && row <3) || !(col >= 0 && col <3)) {
    return 0;
  }
  if (matrix[row][col] == '_') {
    return 1;
  }
  return 0;
}
```

```
void isGameDraw() {
  int found = 0;
  for(int i = 0; i < 3; i++) {
    for(int j = 0; j < 3; j++) {
      if(matrix[j][i] == '_') {
        found = 1;
      }
    }
  }
  if(!found) {
    output = draw;
  }
}

void isGameOver() {
  char charToTest = '0';
  int found;
  if (playerTurn == 0) {
    charToTest = 'x';
  }
  for(int i = 0; i < 3; i++) {
    found = 1;
    for(int j = 0; j < 3; j++) {
      if(matrix[i][j] != charToTest) {
        found = 0;
      }
    }
    if (found) {
      output = over;
      playerWon = playerTurn;
      return;
    }
  }
  for(int i = 0; i < 3; i++) {
    found = 1;
    for(int j = 0; j < 3; j++) {
      if(matrix[j][i] != charToTest) {
        found = 0;
      }
    }
    if (found) {
      output = over;
      playerWon = playerTurn;
      return;
    }
  }

  if(
    (matrix[1][1] == charToTest) &&
    (
```

```
      (matrix[0][0] == matrix[1][1] &&  matrix[1][1]== matrix[2][2]) ||
      (matrix[0][2] == matrix[1][1] &&  matrix[1][1]== matrix[2][0])
    )
  ) {
      output = over;
      playerWon = playerTurn;
      return;
  }


  isGameDraw();
}

void makeMove() {
  int row, column, validMove = 1;
  printf("Enter row and column: ");
  scanf("%d%d", &row, &column);
  validMove = isValidMove(row, column);
  while(!(row >= 0 && row <3) || !(column >= 0 && column <3) || !validMove) {
    printf("Enter correct row and column: ");
    scanf("%d%d", &row, &column);
    validMove = isValidMove(row, column);
  }
  if(playerTurn == 0) {
    matrix[row][column] = 'x';
  } else {
    matrix[row][column] = '0';
  }
  isGameOver();
}

int main() {
  while(output == progress){
    playerTurn = (playerTurn + 1) % 2;
    printf("Current status of board is: \n\n");
    displayBoard();
    printf("\nPlayer %d, please make your move: \n", playerTurn);
    makeMove();
  }
  if(output == draw) {
    printf("\n\nThe game ended in a draw, look at the board below\n");
  } else if(output == over) {
    printf("\n\nPlayer %d has won the game.\nCongratulations!!\nThe final board is: \n",
playerTurn);
  }
  displayBoard();
  return 0;
}
```

ABOUT THE AUTHOR

RAVI SHANKAR

I am an NIT Allahabad graduate in Computer Science and Engineering discipline.

I love challenges, so, right after college I started working for a very early stage startup called THB. At THB, I played a major role in moving the company's core software from a monolith to micro-services architecture. I also helped transition the team and the software from vanilla JavaScript to TypeScript. THB is a medical startup, which builds tool for hospitals and labs. It had a very unique UI requirement. Although Angular 2 was the best fit this requirement, but this also was only partially fitting to the necessity. So, I wrote a wrapper on top of Angular 2, to satisfy the unique requirement for this company.

During my interaction with the VP of engineering of UrbanClap and founders of UrbanClap, I got attracted to the company. I joined UrbanClap in January 2017. At UrbanClap, I initially worked on front-end in React, improving SEO and building certain features of the product. After a while, I shifted to back-end team of UrbanClap, where I works with match-making team. The tools that I uses at UrbanClap are Node.JS, React.JS, JavaScript, AWS, Jenkins, python, elasticSearch, MongoDB, mySql, Alexa etc

I has been very passionate about teaching from the very beginning. I taught Node.JS and web development to my juniors at college. I also ran a startup, Coder Pigeon in Allahabad during my college days. Coder Pigeon was building teaching tool for educational Institutions. To showcase the tool, I started teaching students at Coder Pigeon. For some reason, Coder Pigeon had to be shut down, but I realised my extreme passion for teaching.

In 2019, I started Software Training Institute named Apna Vidyalaya in Greater Noida. Apna Vidyalaya, as the name suggests, aims to be everyones institute. So, at Apna Vidyalaya, we provide very high quality training at a very affordable price. We also release educational materials such as this text, for free. Please check back our website, www.apnavidyalaya.com to get further details. For further details about me, visit www.ravishankarkumar.com