ENVISION™
Computer Training Institute
An **ISO 9001:2008** Certified Company

C Programming

First published on 3 July 2012
This is the 7h Revised edition

Updated on:
03 August 2015

DISCLAIMER

- The data in the tutorials is supposed to be one for reference.
- We have made sure that maximum errors have been rectified. Inspite of that, we (ECTI and the authors) take no responsibility in any errors in the data.
- The programs given in the tutorials have been prepared on, and for the IDE Microsoft Visual Studio 2013.
- To use the programs on any other IDE, some changes might be required.
- The student must take note of that.

History of C Programming

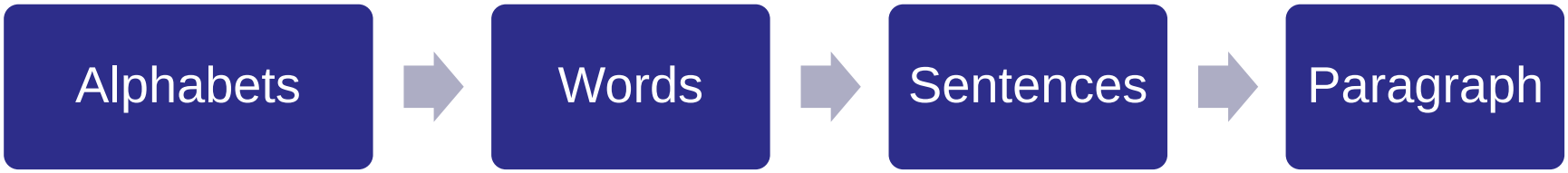
- C Programming Language was introduced by Mr. Dennis Ritchie in 1972. He invented this language for the internal use of AT & T Bell Labs. But due to the features of C Programming Language it became very popular.
- Problems with languages existing before C –
  - They were categorized in **Lower Level Languages and Higher Level Languages**. The Lower Level Languages were designed to give better machine efficiency, i.e. faster program execution. The Higher Level Languages were designed to give better programming efficiency i.e. faster program development.
  - The languages before C were Application Specific e.g. FORTRAN (FORmula TRANslation) was built for Engineering Applications Development, COBOL (COmmon Business Oriented Language) was developed for Commercial Application Development.

History of C Programming

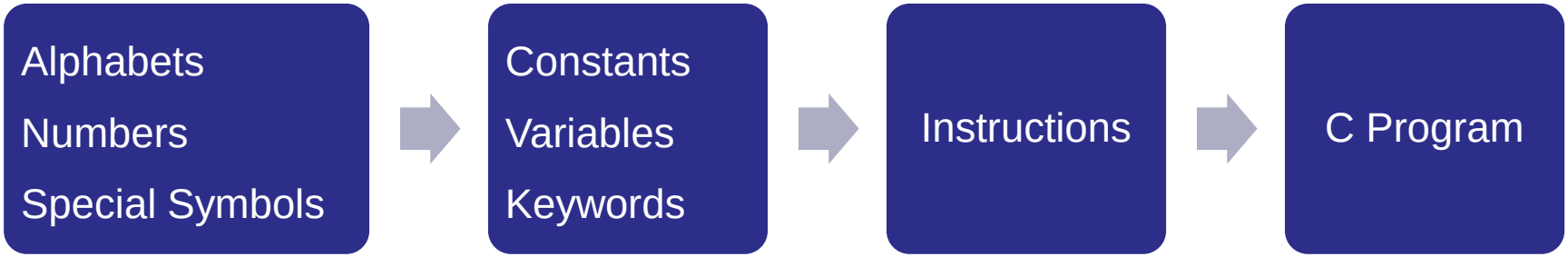
- Dennis Ritchie thought about giving features of both Lower Level Languages and Higher Level Languages. So he introduced the concept of **Compilers** in C. The function of compiler is to **"Convert the code from Human Readable Language to Machine Readable Language and Vice-a-Versa"**.

Basics of C Programming

Steps in Learning English Language:



Steps in Learning C Language:



Basics of C Programming

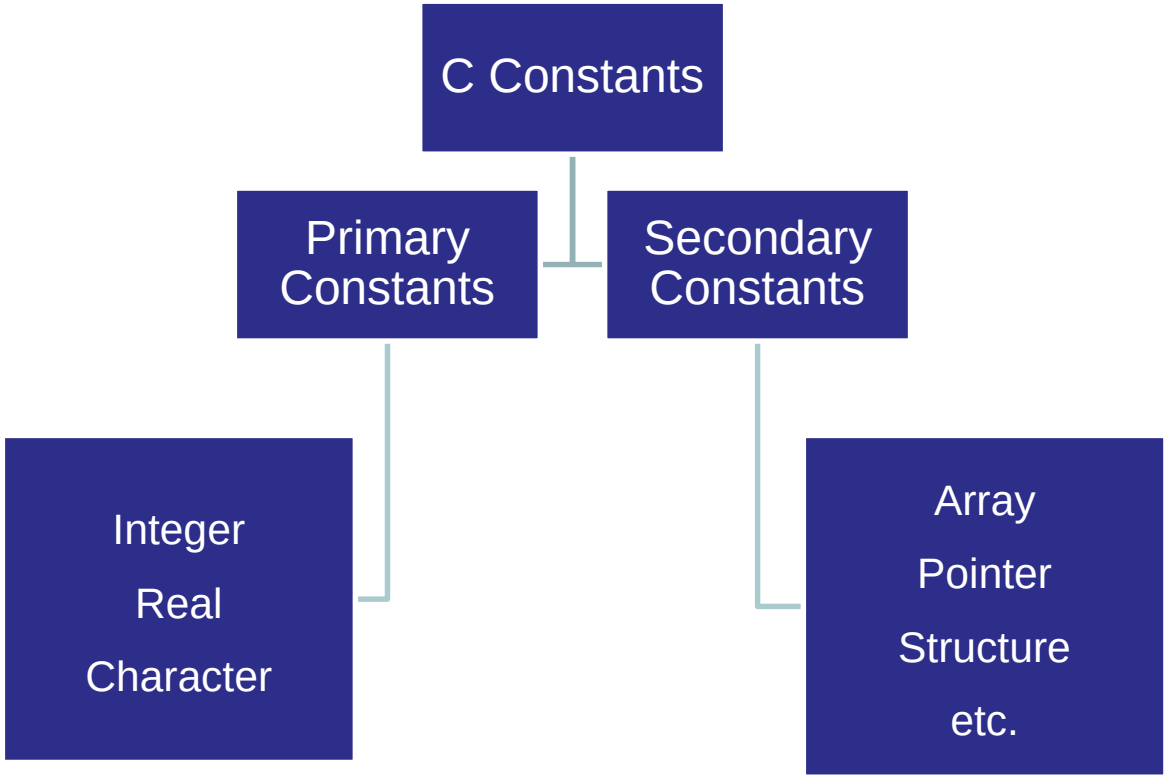
'C' Alphabets:

Alphabets	A, B, ……, Y, Z
	a, b, ……., y, z
Numbers	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Special Symbols	~ ' @ # % ^ & ( ) _ - + | \ { } [ ] : ; " ' < > , . ? /

Basics of C Programming

'C' Constants:

- A constant is an entity that never changes. There are two type of C Constants – **Primary Constants, Secondary Constants.**



Basics of C Programming

Rules for Constructing Integer Constants:

- An Integer Constant must have at least one digit.
- It must not have a decimal point.
- It can be either positive or negative.
- If no sign precedes an Integer Constant, it is assumed to be positive.
- No commas or blanks are allowed in integer constant.
- The allowable range is: **-2,14,74,83,648 to 2,14,74,83,647**.

Rules for Constructing Real Constants:

- A Real Constant must have at least one digit.
- It must have at least one decimal point.
- It can be either positive or negative.
- Default sign is positive.
- No commas or blanks are allowed within Real Constants.
- It's range is**: 3.4e-38 to 3.4e38**.

Basics of C Programming

- **Rules for Constructing Character Constants**:
  - A character constant is a single alphabet, a single digit or a single special symbol enclosed within single inverted commas. Both the inverted commas should point to the left.

'C' Variables:

- An entity that **may vary** during the program execution is called as **Variable**.
- Variable names are names given to the memory locations.
- These locations can hold integer, real or character constants.
- The **integer** variable needs **4 bytes**, **float** variable needs **4 bytes**, **double** variable needs **8 bytes** and **char** variable needs **1 byte memory space**.

Basics of C Programming

- **Rules for Constructing a Variable Name**:
  - The variable name **should not exceed 30 characters**. Some compilers allow variable names upto 247 characters. But restrict variable name, as it adds to the typing effort.
  - The variable name can be **Alpha-numeric** e.g. no1
  - But it should start with an alphabet.
  - It should not contain any spaces in between.
  - It should not contain any special symbols except underscore (_) in between e.g. area_circle
  - It should not contain any Keyword.

Basics of C Programming

Keywords:

- The words which are assigned with special meaning to them in C Compiler are called as keywords. As they are assigned with special meaning, they can not be used as variable names. There are only **32 keywords** available in C.

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Basics of C Programming

'C' Instructions:

- **Type Declaration Instruction**: These instructions are used to declare the type of variables used in a C Program.
  - **int** no1, no2, sum;
  - **float** radius, area;
  - **char** ch;
- **Arithmetic Instructions**:
  - **Arithmetic Operators** –

Addition	+	5 + 2 = 7
Subtraction	-	5 – 2 = 3
Multiplication	*	5 * 2 = 10
Division	/	5 / 2 = 2
Modulus	%	5 % 2 = 1

In normal maths, we write instruction as follows –

A + B = C

In C, we write the same as –

C = A + B

In C '=' is used to assign values to variables.

Basics of C Programming

- **Priority of operators –**

Priority	Operators	Description
1st	( )	Brackets
2nd	* / %	Multiplication, Division, Modulus
3rd	+ -	Addition, Subtraction
4th	=	Assignment

- **Control Instructions:** In C Programming the execution of program happens from top to bottom. There are certain instruction in C which can change the flow of the program. These instructions are called as control instructions. e.g. while, for, goto etc.

Basics of C Programming

Basic Structure of a C Program:

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

void main()
{
    ……….;
    ……….;
    ……….;
}
```

- # – preprocessor instruction
- include<x> – To include a file x to the C Program
- stdio.h / conio.h – Header Files
- void main() – Start of a program
- { } – Scope of a Program

Rules for Writing a C Program

- Each Instruction in C Program is written as a separate statement.
- The statements in program must appear in the same order which we wish them to be executed, unless the logic of the program demands a deliberate 'jump' or transfer of control to a statement, which is out of sequence.
- Blank spaces my be inserted between two words to improve the readability of the statement.
- All syntax should be entered in small case letters.
- Every C statement must end with a ;. Thus ; acts as a statement terminator.
- There is no specific rule for the position at which the statement is to be written. That's why it is often called as free-form language.

First C Program

```
#include<stdio.h>

void main()
{
    printf("This is my first C Program"); /*to print the statement*/
}
```

- The program should be properly **documented**. The statements enclosed in /* */ are called as **comments**.
- Single line comments can also be given using **//**.
- The program should be properly **indented**.
- **printf()** is a function which prints the string embedded in between " " on the console.

Declaring & Assigning Values to Variables

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

void main()
{
    int no1;
    float no2;
    char ch;
    no1 = 10;
    no2 = 25.75;
    ch = 'A';
    printf("Integer = %d, Real = %f, Character = %c", no1, no2, ch);
    getch();          /*waits for user to press any key on console*/
}
```

Note: You can assign the values to the variables at the time of declaration also e.g. **int no1 = 10**;
%d is format specifier for **integers**.
%f is format specifier for **real**.
%c is format specifier for **character**.

scanf() Function

- To receive the values from user scanf() function is used.
- You can receive the single or multiple values within a single scanf.
- e.g. **scanf("%d", &no1)** or **scanf("%d%d“, &no1, &no2)**;
- Generally before any scanf() statement printf() statement should be written to guide the user for giving input.
- **&** means address of the variable.

```
#include<stdio.h>

void main()
{
    int no1;
    printf(“\nEnter any number: ");
    scanf("%d", &no1);      /*receives the value from console*/
}
```

The Decision Control Structure

if statement

- Many times we have to take actions based on certain conditions. C provides **if** or **if-else statement** to allow you to execute statements based on the conditions.
- Before going into details of conditional statements you must know the **Relational Operators** –

less than	<	x < y
greater than	>	x > y
less than equal to	<=	x <= y
greater than equal to	>=	x >= y
equal to	==	x == y
not equal to	!=	x != y

Forms of if-else Statements

- if(condition)
  ……..;
- if(condition)
  {
  ……..;
  ……..;
  }
- if(condition)
  ……..;
  else
  ……..;
- if(condition)
  {
  ……..;
  ……..;
  }
  else
  ……..;

- if(condition)
  ……..;
  else
  {
  ……..;
  ……..;
  }
- if(condition)
  {
  ……..;
  ……..;
  }
  else
  {
  ……..;
  ……..;
  }

Nested if-else Statements

```
if(condition)
{
        if(condition)
        {
                ……..;
                ……..;
        }
        else
        {
                ……..;
                ……..;
        }
}
```

```
else
{
    if(condition)
    {
            ……..;
            ……..;
    }
    else
    {
            ……..;
            ……..;
    }
}
```

Logical Operators

- C contains Logical Operators && (AND), || (OR) and ! (NOT).
- && and || allow to join multiple conditions.
- ! Operator reverses the result of the expression it operates on.

Examples -

- if(!flag) /*will reverse value of flag*/
```
{
        ……..;
        ……..;
}
```

- if(cond1 && cond2 && …… && condN)
```
{
        ……..;
        ……..;
}
```
If all the conditions joined by && are true then the statements will be executed.

- if(cond1 || cond2 || …… || condN)
```
{
        ……..;
        ……..;
}
```
If any of the conditions joined by || is true then the statements will be executed.

- if((cond1 && cond2) || (cond3 && cond4))
```
{
        ……..;
        ……..;
}
```

The else if Statement

```
void main()
{
    int no1=15, no2=10, no3=8;
    if (no1 > no2 && no1 > no3)
        printf("The no1 is maximum");
    if(no2 > no1 && no2 > no3)
        printf("The no2 is maximum");
    if(no3 > no1 && no3 > no2)
        printf("The no3 is maximum");
    getch();
}
```

In the above example even if the first number is maximum the other two if statements are also checked and this will increase the execution time of the program.

```
void main()
{
    int no1=15, no2=10, no3=8;
    if (no1  >no2 && no1 > no3)
        printf("The no1 is maximum");
    else if(no2 > no1 && no2 > no3)
        printf("The no2 is maximum");
    else
        printf("The no3 is maximum");
    getch();
}
```

In the above example if the fist 'if' goes false then only second if will get executed and if the second 'if' goes false then the last else will get executed.

The Conditional Operators

- The conditional operators **?** and **:** are sometimes called as **ternary operators** since they take three arguments. Their general form is –

  expression1 ? expression 2 : expression3

  The above expression suggests "if expression1 is true (that is, if it's value is non-zero), then the value returned will be expression2, otherwise value returned will be expression3.

```
int x, y;
scanf("%d",  &x);
y = (x > 5 ? 3 : 4)
```

This expression will store 3 in y if x is greater than 5 otherwise it will store 4 in y.

```
int i;
scanf("%d",&i);
If(i==1 ? printf("Amit") : printf("Rohit);

char a = 'z';
printf("%c", (a >= 'a' ? a : '!'));

int big a, b, c;
big=(a>b ? (a>c ? 3 : 4) : (b>c ? 6 : 8))
```

The Loop Control Structure

The while loop:

- In programming sometimes you want to run a block of code repeatedly till certain number of times. The while loop helps you to achieve this task.
- The general form of while loop is –

initialize the loop counter;

while(test the loop counter using a **condition**)

{

.....;

.....;

increment / **dec**rement loop counter;

}

The loop counter is initialized before going inside the loop. If the condition is true, it will execute all the statements inside the loop. The counter will be incremented or decremented while going outside the loop. The control is again passed to while. **This will happen till the condition is true.**

More Operators

++ and -- operators:

- When you want to do increment by one i.e. if you want to write **i = i + 1**, you can use **i++** or **++i**; Same way if you want to do decrement by on i.e. if you want to write **i = i – 1**, you can use **i--** or **--i**.
- There is a difference between pre-increment / post-increment or pre-decrement / post-decrement operators. e.g.

```
int i = 10, j;
j = i++;
printf(" %d  %d", i, j);  /* 11  10*/
```

```
int i = 10, j;
j = ++i;
printf(" %d  %d", i, j);  /* 11  11*/
```

In the first example the value of i is first assigned to j and then the value of i gets incremented.

In the second example the value of i is first incremented and then it is assigned to j.

Compound Assignment Operators

i = i + 5	**i+=5**	i = i – 5	**i-=5**
i = i * 5	**i*=5**	i = i / 5	**i/=5**
i = i % 5	**i%=5**		

Nesting of while loop

- The way we have seen nesting of 'if' statements, we can do nesting of loops. Nesting of loop means loop into another loop. Example –

```
*
**
***
****
```

- In the above example, you have to print multiple stars in columns for each row. Such times you need nesting of loops.

```
void main()
{
        int r = 1, c;
        while(r <= 4)
        {
                c = 1;
                while (c <= r)
                {
                        printf("*");
                        c++;
                }
                printf("\n");
                r++;
        }
        getch();
}
```

do while Loop

- The difference between while and **do while** is the do while loop **at least gets executed single time**.

```
initialize the loop counter;
do
{
        .....;
        ……;
        increment / decrement loop
        counter;
} while (condition);
```

- do while loop is generally used with **odd loops**. When the programmer does not know how many time the loop will get executed the odd loops are used.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int no1, no2, sum;
    char ch;
    do
    {
        printf("Enter 2 nos: ");
        scanf("%d%d", &no1, &no2);
        sum = no1 + no2;
        printf("Sum = %d", sum);
        printf("Want to continue <y/n>: ");
        flushall();
        scanf("%c", &ch);
    } while(ch == 'y');
}
```

for Loop

- for loop specify three things about the loop in a single line.
- The general form of for statement is as follows –

for(initialize; condition; increment / decrement)

```
{
        ..........;
        ..........;
}
```

e.g.

```
for(i = 0; i <= 10; i++)
{
        printf("%d", i);
}
```

- In for loop the initializations are done only at the first time. Then condition is checked and while going outside the loop the loop counter is incremented or decremented.
- You can do multiple initializations or multiple increments / decrements using comma separation.

e.g.

```
for(i = 0, j = 5; i < j; i++, j--)
{
        printf("%d %d", i, j);
}
```

break, continue Statement

- **break** statement is used to jump out of the loop instantly.

e.g.

```
for(i = 0; i <= 10; i++)
{
        if(i == 5)
                break;
        printf("%d  ", i);
}
```

The output of the above program is:

0 1 2 3 4

- **continue** statement takes the control of the program to the beginning of loop.

e.g.

```
for(i = 0; i <= 10; i++)
{
        if(i == 5)
                continue;
        printf("%d  ", i);
}
```

The output of the above program is:

0 1 2 3 4 6 7 8 9 10

The goto Keyword

- **goto** keyword allows to jump the program from one location to another.
- Though it is easy to use, but it makes the program unreliable, unreadable and hard to debug. So avoid using it or use it in a very unavoidable circumstances.
- The word followed by goto keyword is called as **label**. It takes the control to that specific label.

```
void main()
{
        int i;
        for(i = 0; i < 10; i++)
        {
                if(i == 5)
                        goto xyz;
                else
                        printf(" %d", i);
        }
        xyz:
        printf("\nI am out of main");
}
```

The output is:

0 1 2 3 4

I am out of main

switch case Control Structure

- The **switch-case-default** allows us to make decision from many choices.
- The expression followed by switch should be **integer** or **character**.
- The keyword case is followed by **integer or character constants**.
- When user gives some choice the switch statement takes the control to the specified case.
- If no such case exists, then control goes to **default**.
- Each case ends with **break.**

- You cannot use logical **&& or ||** operators with cases.

```
switch(choice)
{
        case 1:
                        ……..;
                        break;
        case 2:
                        ……..;
                        break;
        case 3:
                        ……..;
                        break;
        default:
                        ……..;
}
```

Functions

- Sometimes there is a code which occurs repeatedly inside main. To avoid this repetition we write functions. Functions helps to reduce length of code.
- A function is a block of code that performs a specific assigned task.
- You can call one function from another function but you cannot write one function's definition into another.
- The scope of the variables is limited to the function.
- Two different functions can have variables with same name. But their memory locations are different.

```
#include<stdio.h>
void pune();
void mumbai();
void main()
{
      printf("\nI am in main");
      pune();
      mumbai();
      printf("\nI am back in main");
      getch();
}
void pune()
{
      printf("\nI am in Pune");
}
void mumbai()
{
      printf("\nI am in Mumbai");
}
```

Call to a function

Function Definitions

Functions (Passing Parameters)

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

void calsum();

void main()
{
      calsum();
      getch();
}
void calsum()
{
      int no1, no2, add;
      printf("\nEnter two numbers: ");
      scanf("%d%d", &no1, &no2);
      add = no1 + no2;
      printf("\nThe sum is: %d", add);
}
```

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

void calsum(int, int);

void main()
{
      int no1, no2;
      printf("\nEnter two numbers: ");
      scanf("%d%d", &no1, &no2);
      calsum(no1, no2);
      getch();
}
void calsum(int a, int b)
{
      int add;
      add = a + b;
      printf("\nThe sum is: %d", add);
}
```

In this program we are **parameters** to the function. These functions are called as **Call by Value** functions.

Actual Arguments

Formal Arguments

Functions (Returning Value)

- The function can **return** a value to a calling function.
- The function can **return only one value** at one call.
- The return statement immediately passes control to the calling function.
- Replace void before the function definition by **int**, **float** or **char**.
- There can be more than one return statements in a function but at a time only one will get executed.

```
#include<stdio.h>
#include<conio.h>
int calsum(int, int);
void main()
{
    int no1, no2, sum;
    printf("\nEnter two numbers: ");
    scanf("%d%d", &no1, &no2);
    sum = calsum(no1, no2);
    printf("\nThe sum is: %d", sum);
    getch();
}
int calsum(int a, int b)
{
    int add;
    add = a + b;
    return add;
}
```

In this program the sum function returns a integer value back to main. So **int** is written instead of **void**

Recursive Functions

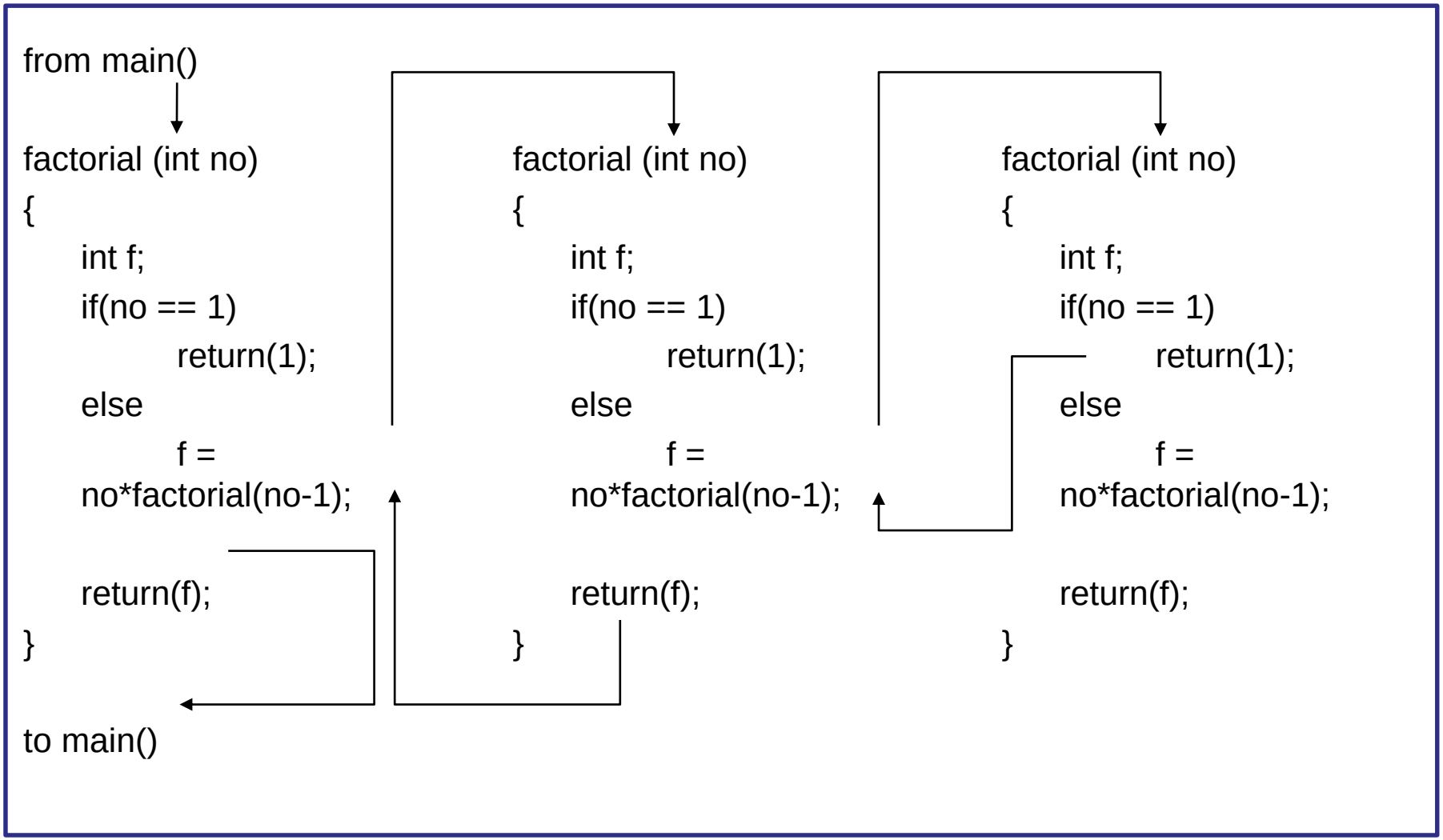
- You can give a call to a function from the same function it is called as recursive function.

```
#include<stdio.h>
#include<conio.h>
int factorial(int );
void main()
{
    int no, fact;
    printf("\nEnter any number: ");
    scanf("%d", &no);
    fact = factorial(no);
    printf("\nThe factorial is: %d",
    fact);
    getch();
}
```

```
int factorial (int no)
{
    int f;
    if( no == 1)
        return(1);
    else
        f = no * factorial (no – 1);
    return(f);
}
```

- **Make sure that the recursive call is conditional. An unconditional recursive call will result to a infinite call.**

Recursive Function



Data Types Revised

Data Type	Range	Bytes	Format Specifiers
char	-128 to 127	1	%c
unsigned char	0 to 255	1	%c
short signed int	-32,768 to 32,767	2	%d
short unsigned int	0 to 65,535	2	%u
signed int	-32768 to 32767	4	%d
unsigned int	0 to 65535	4	%u
long signed int	-2147483648 to 2147483647	4	%ld
long unsigned int	0 to 4294967295	4	%lu
float	-3.4e38 to 3.4e38	4	%f
double	-1.7e308 to 1.7e308	8	%lf
long double	-1.7e4932 to 1.7e4932	8	%Lf
Note: The sizes in this figure are for 32 bit compiler			

C Preprocessor

- There are several steps involved from the stage of writing a C Program to the stage of getting it executed. The combination of these steps is known as the '**Build Process**'.
- Before the C Program is compiled it is passed through another program called as '**Preprocessor**'.
- C program is often known as '**Source Code**'. The Preprocessor works on Source Code and creates '**Expanded Source Code**'. If source file is Prog01.C, the expanded source code is stored in file Prog01.I. The expanded source code is sent for compilation.
- The preprocessor directives begin with # symbol. Generally they are placed at the start of the program, before function definitions.
- Following are the preprocessor directives –
  - Macro Expansion
    - Simple Macros
    - Macros with Arguments
  - File Inclusion
  - Conditional Compilation
  - Miscellaneous Directives

C Preprocessor

Macro Expansion

#define PI 3.142

- The above expression is called as '**macro definition**'. During the preprocessing PI will be replaced with 3.142.
- **PI** in above example is called as '**macro template**' and 3.142 is called as '**macro expansion**'.
- When we compile the program, before the source code passes to the compiler, it is examined by C preprocessor for macro definitions. And when it finds the macro templates they replace it with appropriate macro expansion.

Some examples -

- #define AND &&

  We can use AND instead of &&
- #define OR ||

  We can use OR instead of ||
- #define RANGE (a > 25 && a < 50)

  We can use RANGE instead of (a>25 && a < 50)
- #define FOUND printf("Location Found");

  We can use FOUND instead of printf("Location Found")

Macros with Arguments

- We can pass arguments to macros.

```
#include<stdio.h>
#define AREA(x) (3.142 * x * x)
void main()
{
        float r = 5.25, a;
        a = AREA(r);
        printf("\nThe area is : %f", a);
}
```

When the above code is passed to preprocessor –

a = AREA(r);

will get converted to –

a = 3.142 * r * r;

Multiline Macros

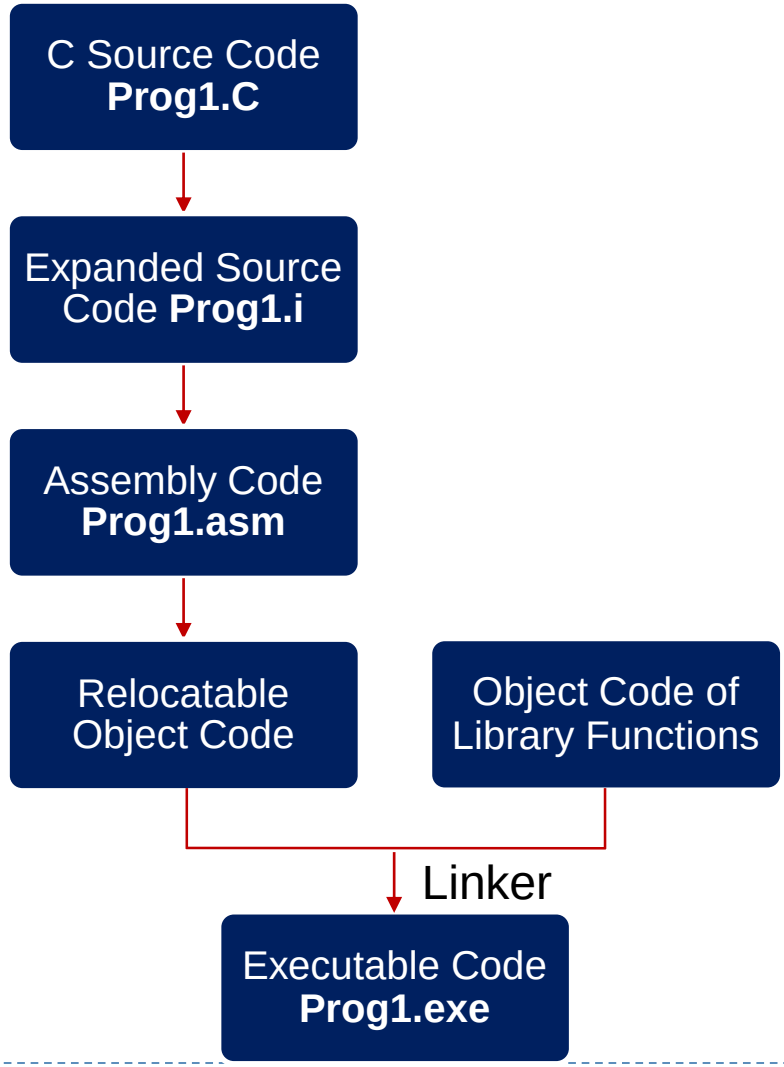
- Macros can be split into multiple lines with a '\' (back slash) present at the end of each line.

```
#include<stdio.h>
#define HLINE for(i=0; i<79; i++) \
                        printf("%c", 196);
#define VLINE(X, Y) { \
            gotoxy(X, Y); \
            printf("%c", 179); }
void main()
{
    int i, y;
    gotoxy(1, 12);
    HLINE;
    for (y = 1; y < 25; y++)
            VLINE(39, y);
}
```

File Inclusion

- This preprocessor directive allows us to include one file into other.
- The large programs can be divided into different files, each containing a set of functions. These files can be included at the beginning of main program file.
- The files to be included should have .h extension. This extension stands for a '**header file**'.
- Each header file generally contains related set of functions.
  - e.g. **math.h** will contain all functions related to mathematics.
- You can include these files into main using **#include** statement.
  - #include "filename"
  - #include<filename>
- **#include "filename"** will search filename into current directory as well as specified list of directories mentioned in the include search path.
- **#include <filename>** will search the filename into specified list of directories only.

The Build Process



- **Preprocessing**: The C code is expanded on preprocessor directives like #define, #include etc.
- **Compilation**: If expanded source code is error free then compiler translates the expanded source code into equivalent assembly language program.
- **Assembling**: The job of Assembler is to translate .asm code into relocatable object code. Thus a **.asm** file gets converted to **.obj** file.
- **Linking**: Linking creates an executable program. It finds the definitions of all external functions, finds definitions of all global variables, combines data sections into a single data section and combines code sections into single code section. And finally **.exe** file is created.

Arrays

- Array is a collection of variables of same data type.
- E.g. if we want to store roll numbers of 100 students, then you have to define 100 variables and which is very difficult to manage. Instead you can define one array which can hold 100 variables.
- Declare Array:

  int arr[5];

  int arr[5] = {10,20,30,40,50};

  int arr[ ] = {10,20,30,40,50};
- When you declare an array a consecutive memory allocation is done for 5 integers.
- You can **initialize** the array only at the time of **declaration**.
- Array index always starts from **zero**.
- If you want to refer array elements you can use **arr[0], arr[1], arr[2]** etc.
- In the following array 10 is a value at array index 0 having address 6231.
- Memory Map of Array:

	0	1	2	3	4
arr	10	20	30	40	50
	6231	**6235**	**6239**	**6243**	**6247**

Arrays

Accept and Print Array:

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

void main()
{
	int arr[5], i;
	printf("\nEnter five numbers: ");
	for(i = 0; i < 5; i++)
		scanf("%d", &arr[i]);
	printf("\nYou have entered: ");
	for(i = 0; i < 5; i++)
		printf("  %d", arr[i]);
	getch();
}
```

- As array holds multiple locations you have to use for loop while accepting and printing the array.
- You can define an array of bigger size and accept the required no. of values from user. e.g. you can define an array of arr[100] and use only 10 locations.
- In C Language the **bounds checking** for array is not done. So the programmer has to be very careful while accepting or printing the values from array.

Passing Array to a Function

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

void display(int [ ], int);

void main()
{
      int arr[5], i;
      printf("\nEnter five numbers: ");
      for(i = 0; i < 5; i++)
            scanf("%d", &arr[i]);
      display(arr, 5);
      getch();
}
```

```
void display (int a[ ], int s)
{
      int i;
      printf("\nYou have entered: ");
      for(i = 0; i < s; i++)
            printf("  %d", a[i]);
}
```

Two Dimensional Arrays

- Array can have two or more dimensions.
- Declare 2D-Array:

  int a[2][2];

  int a[ ][2] = {10,20,30,40};

  int a[ ][2] = {{10,20},{30,40}};

- Memory Map of 2D-Array:

	a[0][0]	a[0][1]	a[1][0]	a[1][1]
a	10	20	30	40
	6231	**6235**	**6239**	**6243**

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int arr[3][3], i, j;
        printf("\nEnter nine numbers: ");
        for(i = 0; i < 3; i++)
                for(j = 0; j < 3; j++)
                        scanf("%d", &arr[i][j]);
        printf("\nYou have entered: ");
        for(i = 0; i < 3; i++)
                for(j = 0; j < 3; j++)
                        printf("  %d", arr[i][j]);
        getch();
}
```

Passing 2D-Array to a Function

```
#include<stdio.h>
#include<conio.h>
void display(int [ ][3], int, int);
void main()
{
      int arr[3][3], i, j;
      printf("\nEnter nine numbers: ");
      for(i = 0; i < 3; i++)
            for(j = 0; j < 3; j++)
                  scanf("%d", &arr[i][j]);

      display(arr, 3, 3);
      getch();
}
```

```
void display (int a[ ][3], int r, int c)
{
      int i, j;
      printf("\nYou have entered: ");
      for(i = 0; i < r; i++)
            for(j = 0; j < c; j++)
                  printf("  %d", a[i][j]);
}
```

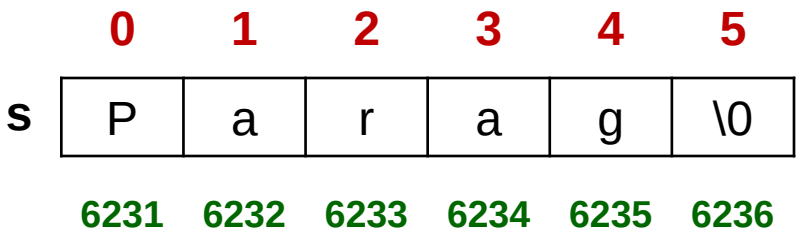
Strings

- String is a collection of character variables. Sometimes you need to store name, address where you have to store more than one characters, you need strings.
- Declare String:

  char s[50];

  char s[ ] = {'P','a','r','a','g','\0'};

  char s[ ] = "Parag";
- Memory Map of a String:



- **'\0'** character is automatically appended at the end while accepting the string.

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

void main()
{
        char name[50];
        int i;
        printf("\nEnter your name: ");
        //scanf("%s", name);
        gets(name);
        printf("\nYour name is: %s", name);
        printf("\nYour name is: ");
        for(i = 0; name[i] != '\0'; i++)
                printf("%c", name[i]);
        getch();
}
```

Strings

- %s is special format specifier to accept the strings. But it cannot accept string with spaces. So use **gets(string-name)** function to accept string.
- You can print the string using **puts(string-name)** function.
- You are not needed to accept the size of strings as you require in case of arrays because C automatically appends the '\0' character at the end of the string. Hence you know when a '\0' character appears, that is the end of the string.

Standard Library Functions

Function	Use
strlen	Finds the length of the string
strlwr	Converts string to lowercase
strupr	Converts string to uppercase
strcpy	Copies string to another string
strcat	Appends one string at the end of another string
strcmp	Compares two string
strrev	Reverses the string

Passing String to Function

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

void display(char [ ]);

void main()
{
      int name[50], i;
      printf("\nEnter your name: ");
      gets(name);
      display(name);
      getch();
}
```

```
void display (char str[ ])
{
      int i;
      printf("\nYour name is: %s", str);
      printf("\nYour name is: ");
      for(i = 0; str[i] != '\0' ; i++)
            printf("%c", str[i]);
}
```

Output:

```
Enter your name: Rohit

Your name is: Rohit
Your name is: Rohit
```

Two Dimensional Strings

- Strings also can have 2 or more dimensions. When you want to store multiple names, addresses that time you need 2-D string.
- Declare 2D-String:

char s[2][10];

char s[ ][10] = {"Parag",
"Pooja",
"Sarang"};

- Memory Map of 2D-String:

6231	P	a	r	a	g	\0				
6241	P	o	o	j	a	\0				
6251	S	a	r	a	n	g	\0			

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

void main()
{
        char name[3][50];
        int i;
        printf("\nEnter three names: ");
        for(i = 0; i < 3; i++)
                gets(name[i]);
        printf("\nThe names are: ");
        for(i = 0; i < 3; i++)
                puts(name[i]);
        getch();
}
```

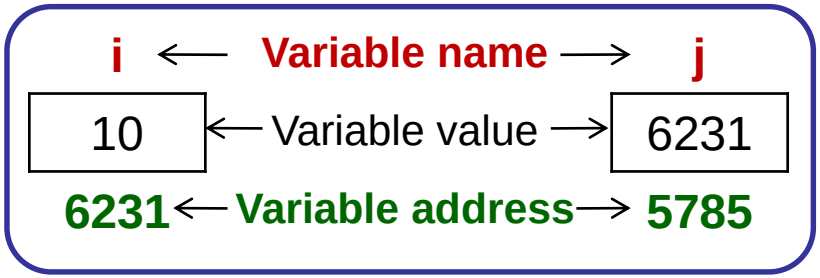
Passing 2D-String to a Function

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

void display(char [ ][50], int);

void main()
{
        char name[3][50];
        int i;
        printf("\nEnter three names: ");
        for(i = 0; i < 3; i++)
                gets(name[i]);
        display(name, 3);
        getch();
}
```

```
void display (char n[ ][50], int s)
{
        int i;
        printf("\nThe names are: ");
        for(i = 0; i < s; i++)
                puts(n[i]);
}
```

Pointers

- Pointer is a variable which can store address of another variable.
  - **& means address**
  - *** means value at that address (except declaration)**



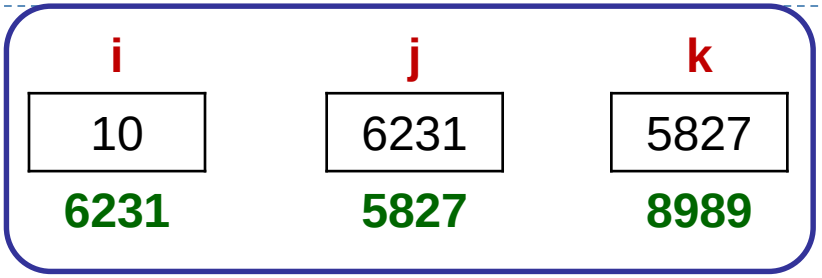

```
#include<stdio.h>
void main()
{
        int i = 10, *j;
        j = &i;
        printf("\nAddress of i = %u", &i);
        printf("\nAddress of i = %u", j);
        printf("\nAddress of j = %u", &j);
        printf("\nValue of j = %u", j);
        printf("\nValue of i = %d", i);
        printf("\nValue of i = %d", *(&i));
        printf("\nValue of i = %d", *j);
        getch();
}
```

- **The output of above program**:

```
Address of i = 6231        Value of i = 10
Address of i= 6231         Value of i = 10
Address of j = 6233        Value of i = 10
Value of j = 6231
```

Pointers

```
#include<stdio.h>
void main()
{
        int i = 10, *j, **k;
        j = &i;
        k = &j;
        printf("\nAddress of i = %u", &i);
        printf("\nAddress of i = %u", j);
        printf("\nAddress of i = %u", *k);
        printf("\nAddress of j = %u", &j);
        printf("\nAddress of j = %u", k);
        printf("\nAddress of k = %u", &k);
        printf("\nValue of j = %u", j);
        printf("\nValue of k = %u, k);
        printf("\nValue of i = %d", i);
        printf("\nValue of i = %d", *(&i));
        printf("\nValue of i = %d", *j);
        printf("\nValue of i = %d", **k);
        getch();
}
```



The output of the program:

```
Address of i = 6231
Address of i = 6231
Address of i = 6231
Address of j = 6233
Address of j = 6233
Address of k =6235
Value of j = 6231
Value of k = 6233
Value of i = 10
Value of i = 10
Value of i = 10
Value of i = 10
```

Call By Reference Functions

- We have already seen the call by value type of functions. The disadvantage of call by value is you can return only one value and only once.
- To overcome the disadvantage of call by value we can use call by reference type of functions.
- When you pass address of a variable to function that time the function is called as call by reference.
- In the below example though we have not returned the value back to calling function, we get the value back to sum variable due to pointer.

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

void calsum(int, int, int *);

void main()
{
        int no1, no2, sum;
        printf("\nEnter two numbers:");
        scanf("%d%d", &no1, &no2);
        calsum(no1, no2, &sum);
        printf("\nSum = %d");
        getch( );
}
void calsum(int x, int y, int *add)
{
        *add = x + y;
}
```

Passing Array to Function using Pointers

- When we want to pass the array to function using pointers we pass it as **display(arr, size)** and catch it in function using **void display(int *b, int s)**.
- So if arr is as follows –

	0	1	2	3	4
arr	10	20	30	40	50
	6231	6235	6239	6243	6247

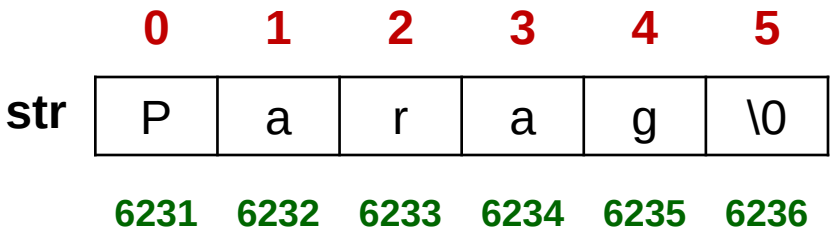
So the base address of arr will be stored in b. So **b** will hold **6231**.

- When you do pointer++ it will be increased by no. of bytes of it's datatype.

```
#include<stdio.h>
void display (int *, int);
void main()
{
        int arr[5], i;
        printf("\nEnter five numbers: ");
        for(i = 0; i < 5; i++)
                scanf("%d", &arr[i]);
        display(arr, 5);
        getch();
}
void display (int *b, int s)
{
        int i;
        printf("\nYou have entered: ");
        for(i = 0; i < s; i++)
                printf("  %d", *(b+i));
}
```

Passing String to Function using Pointers

- When we want to pass the string to function using pointers we pass it as **display(str)** and catch it in function using **void display(char *s)**.
- So if string is as follows –



So the base address of str will be stored in s. So **s** will hold **6231**.

- There is no need to pass size of the string as the end character of string holds '\0';

```
#include<stdio.h>
#include<conio.h>
void display (char *);
void main()
{
        char str[50];
        printf("\nEnter your name: ");
        gets(str);
        display(str);
        getch();
}
void display (char *s)
{
        printf("\nYou have entered: %s", s);
        printf("\nYou have entered: ");
        for( ; *s != '\0'; s++)
                printf("%c", *s);
}
```

Structures

- If we see real world data it is a combination of integers, characters, floats, arrays, strings etc. So we can call it as a **heterogeneous data**.
- If we want to store data of a student then we need to store her/ his Roll Number, Name, Address, Age, Gender, Phone No., E-mail etc. So to store this kind of data we can use structures.
- Structure is also called as **User Defined Datatype**.

```
struct stud
{
        int roll_no;
        char name[20];
        char gender;
};
```

- Above syntax is used to define a structure.
- Memory Map of Structure:

	roll_no	name	gender
stud	1	Parag Patki	50
	6231	6235	6255

Structures

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

struct stud
{
        int roll_no;
        char name[50], address[100];
        char gender;
};

void main()
{
        struct stud s;
        printf("\nEnter roll number:");
        scanf("%d", &s.roll_no);
        printf("\nEnter name: ");
        flushall( );
        gets(s.name);
        printf("\nEnter address: ");
        flushall();
        gets(s.address);
        printf("\nEnter gender <y/n>: ");
        flushall( );
        scanf("%c", &s.gender);
        printf("\nRoll No. is: %d", s.roll_no);
        printf("\nName is: %s", s.name);
        printf("\nAddress is: %s", s.address);
        printf("\nGender is: %c", s.gender);
        getch();
}
```

Nested Structures

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

struct p_details
{
        int roll_no;
        char name[50];
        char gender;
        char email[50];
};
struct  e_student
{
        int m1, m2;
        struct p_details p;
};
void main()
{
        struct e_student e;
        printf("\nEnter roll number:");
        scanf("%d", &e.p.roll_no);
        printf("\nEnter Name: ");
        flushall( );
        gets(e.p.name);
        printf("\nEnter Gender <y/n>: ");
        flushall( );
        scanf("%c", &e.p.gender);
        printf("\nEnter E-mail: ");
        flushall( );
        gets(e.p.email);
        printf("Enter marks of three subjects:");
        scanf("%d%d", &e.m1, &e.m2);
        printf("\nName = %s", e.p.name);
        printf("\nRoll No. = %d", e.p.roll_no);
        printf("\nGender = %c", e.p.gender);
        printf("\nEmail = %s", e.p.email);
        printf("Marks = %d %d", e.m1, 2.m2);
        getch( );
}
```

Array of Structures

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

struct student
{
        int roll_no;
        char name[50];
        int m1, m2;
};
typedef struct student STUD;

void main()
{
        STUD s[50];
        int i, size;
        printf("\nEnter no. of records: ");
        scanf("%d", &size);
        printf("\nEnter %d records:", size);
        for(i=0; i<size; i++)
        {
                printf("\nEnter roll no.:");
                scanf("%d", &s[i].roll_no);
                printf("\nEnter Name: ");
                flushall( );
                gets(s[i].name);
                printf("\nEnter marks:");
                scanf("%d%d", &s[i].m1, s[i].m2);
        }
        printf("\n\n\nRecords are:");
        for(i=0; i<size; i++)
        {
                printf("\nRoll no.: %d", s[i].roll_no);
                printf("\nName: %s", s[i].name);
                printf("\nMarks:%d %d", s[i].m1, s[i].m2);
        }
        getch( );
}
```

Passing Structure to Function

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

typedef struct stud
{
        int roll_no;
        char name[50];
        int m1, m2;
}STUD;
void display(STUD);

void main( )
{
        STUD s;
        printf("\nEnter Roll Number: ");
        scanf("%d", &s.roll_no);
        printf("\nEnter Name: ");
        flushall( );
        gets(s.name);
        printf("\nEnter Marks:");
        scanf("%d%d", &s.m1, &s.m2);
        display(s);
        getch( );
}

void display(STUD s1)
{
        printf("\nRoll No.: %d", s1.roll_no);
        printf("\nName: %s", s1.name);
        printf("\nMarks: %d %d", s1.m1, s1.m2);
}
```

Passing Array of Structures to Function

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

struct stud
{
        int roll_no;
        char name[50];
        int m1, m2;
};
void display(struct stud[ ] );

void main( )
{
        int I;
        struct stud s[5];
        for(i=0;i<5;i++)
        {
                printf("\nEnter Roll Number: ");
                scanf("%d", &s[i].roll_no);
                printf("\nEnter Name: ");
                flushall( );
                gets(s[i].name);
                printf("\nEnter Marks:");
                scanf("%d%d", &s[i].m1, &s[i].m2);
        }
        display(s);
        getch( );
}
void display(struct stud s1[ ])
{
        for(int i=0;i<5;i++)
        {
                printf("\nRoll No.: %d", s1[i].roll_no);
                printf("\nName: %s", s1[i].name);
                printf("\nMarks: %d %d", s1[i].m1, s1[i].m2);
        }
}
```

Structure to Pointer

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

struct stud
{
        int roll_no;
        char name[50];
        int m1, m2;
};
void display(struct stud *);

void main( )
{
        struct stud s;
        printf("\nEnter Roll Number: ");
        scanf("%d", &s.roll_no);
        printf("\nEnter Name: ");
        flushall( );
        gets(s.name);
        printf("\nEnter Marks:");
        scanf("%d%d", &s.m1, &s.m2);
        display(&s);
        getch( );
}

void display(struct stud *s1)
{
        printf("\nRoll No.: %d", s1->roll_no);
        printf("\nName: %s", s1->name);
        printf("\nMarks: %d %d", s1->m1, s1->m2);
}
```

Passing Array of Structures to Pointer

```
#include<stdio.h>
#include<conio.h>
struct stud
{
      int roll_no;
      char name[50];
};
void display(struct stud *, int);
void main()
{
      struct stud s[3];
      int i;
      printf("\nEnter 3 records: \n\n");
      for(i=0; i<3; i++)
      {
            printf("\nEnter roll no.: ");
            scanf("%d", &s[i].roll_no);
            printf("\nEnter Name: ");
            flushall();
            gets(s[i].name);
            printf("\n\n");
      }
      display(s,3);
      getch();
}

void display(struct stud *s1, int n)
{
      int i;
      printf("\nYou have entered: \n\n");
      for(i=0; i<n; i++, s1++)
      {
            printf("\nEnter roll no.: %d", s1->roll_no);
            printf("\nEnter Name: %s", s1->name);
      }

}
```

Input / Output in C

- There are two types of I/O.
  - Console I/O
  - File I/O
- Till now whatever programs we have seen, in those programs data vanishes when the program ends. So to store the data on the disk we need **file I/O** operations.
- All the data on the disk is stored in binary form. But the process of storing data varies from OS to OS. The programmer uses library function in a particular OS to perform I/O.
- There are different operations that can be carried out on the file:
  - Creation of new file
  - Opening an existing file
  - Reading from a file
  - Writing to a file
  - Moving to a specific location in the file
  - Closing a file

Display Contents of File on Console

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
void main( )
{
        FILE *fp;
        char ch;
        fp = fopen("Source.txt", "r");
        if(fp == NULL)
        {
                puts("Cannot open source file");
                exit(1);
        }
        while(1)
        {
                ch = fgetc(fp);
                if(ch == EOF)
                        break;
                printf("%c", ch);
        }
        fclose(fp);
        getch();
}
```

- **FILE** is a **inbuilt structure** defined in stdio.h header file.
- **fopen()** function searches the file on disk to be opened, if file found it loads it into buffer and it sets up a character pointer that points to the first character of buffer. **If file not found** it returns **NULL**.
- **exit()** function terminates the program.
- **while(1)** is a infinite loop.
- **fgetc(fp)** function returns the character present at the pointer fp and moves the pointer to the next location.
- **EOF** stands for **End Of File character** which indicates the end of file.
- **fclose()** function closes the file.

Writing From One File to Another

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
void main( )
{
        FILE *fs, *ft;
        char ch;
        fs = fopen("Source.txt", "r");
        if(fs == NULL)
        {
                puts("Cannot open source file");
                exit(1);
        }
        ft = fopen("Target.txt", "w");
        if(ft == NULL)
        {
                puts("Cannot open target file");
                fclose(fs);
                exit(2);
        }
        while(1)
        {
                ch = fgetc(fs);
                if(ch == EOF)
                        break;
                else
                        fputc(ch, ft);
        }
        fclose(fs);
        fclose(ft);
        printf("File copied successfully");
        getch();
}
```

- **fputc(ch, ft)** writes the character ch to the file location ft and moves pointer ahead to next location.

Writing String Received From Keyboard to File

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#include<stdlib.h>
void main( )
{
      FILE *fp;
      char s[80];
      fp = fopen("Target.txt", "w");
      if(fp == NULL)
      {
            puts("Cannot open file");
            exit(1);
      }
      printf("\nEnter few lines: ");
      while(strlen(gets(s)) > 0)
      {
            fputs(s, fp);
            fputs("\n", fp);
      }
      fclose(fp);
      printf("Data writing successful");
      getch();
}
```

- The function **fputs()** writes the given string to the file. Since fputs() does not add newline character automatically we have to add it explicitly.

Reading String From File and Printing on Console

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
void main( )
{
        FILE *fp;
        char s[80];
        fp = fopen("Target.txt", "r");
        if(fp == NULL)
        {
                puts("Cannot open file");
                exit(1);
        }
        printf("\nThe file data is:\n");
        while(fgets (s, 79, fp) != NULL)
                        printf("%s", s);
        fclose(fp);
        getch();
}
```

- The function **fgets()** requires three arguments. The **first** is the **address where the string is stored**, the **second** is **maximum length of string** and the **third** argument is the **pointer to the structure FILE**. When all the lines from the file have been read, fgets() returns **NULL**.

Write Records to File Using Structure

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
void main( )
{
        FILE *fp;
        char ch;
        struct emp
        {
                char name[50];
                int age;
                float sal;
        };
        struct emp e;
        fp = fopen("Data.txt", "w");
        if(fp == NULL)
        {
                puts("Cannot open file");
                exit(1);
        }
        do
        {
            printf("\nEnter name, age, sal:");
            scanf("%s%d%f", e.name, &e.age, &e.sal);
            fprintf(fp, "%s %d %f\n", e.name, e.age,
            e.sal);
            printf("\nAdd another record <y/n>: ");
            fflush(stdin);
            ch = getche();
        }while(ch == 'y');
        fclose(fp);
        printf("\nData written successfully:");
        getch();
}
```

- **fprintf()** writes the structure to file. The first argument is FILE pointer,

Read Records From File Using Structure

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
void main( )
{
        FILE *fp;
        char ch;
        struct emp
        {
                char name[50];
                int age;
                float sal;
        };
        struct emp e;
        fp = fopen("Data.txt", "r");
        if(fp == NULL)
        {
                puts("Cannot open file");
                exit(1);
        }
        while(fscanf (fp, "%s %d %f",
        e.name, &e.age, &e.sal) != EOF)
                printf("\n%s %d %f", e.name,
                e.age, e.sal);
        fclose(fp);
        getch();
}
```

Writing Records Using fwrite()

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
void main( )
{
        FILE *fs;
        char another;
        struct emp
        {
                char name[50];
                int age;
                float sal;
        }e;

        fs = fopen("Source.txt", "wb");
        if(fs == NULL)
        {
                puts("Cannot open source file");
                exit(1);
        }
        do
        {
                printf("\nEnter name, age, sal:");
                scanf("%s%d%f", e.name, &e.age, &e.sal);
                fwrite(&e, sizeof(e), 1, fs);
                printf("\nAdd another record <y/n>?: ");
                fflush(stdin);
                another = getche();
        }while(another == 'y');
        fclose(fs);
        printf("\nData written successfully to file");
        getch();
}
```

- In **fwrite()** function the first argument is the address of the structure, the second argument is size of structure in bytes, the third argument is the number of elements to be written and the last argument is the pointer to file where we want to write to.

Reading Records Using fread()

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
struct emp
{
        char name[50];
        int age;
        float sal;
}e;

void main( )
{
        FILE *fs;
                fs = fopen("Source.txt", "rb");
        if(fs == NULL)
        {
                puts("Cannot open source file");
                exit(1);
        }
        while(fread (&e, sizeof(e), 1, fs) == 1)
                printf("\n%s %d %f", e.name,
                e.age, e.sal);
        fclose(fs);
}
```

Command Line Arguments

- In C it is possible to accept command line arguments. Command-line arguments are given after the name of a program in command-line operating systems and are passed in to the program from the operating system. To use command line arguments in a program, we first see the full declaration of the main function, which previously has accepted no arguments.

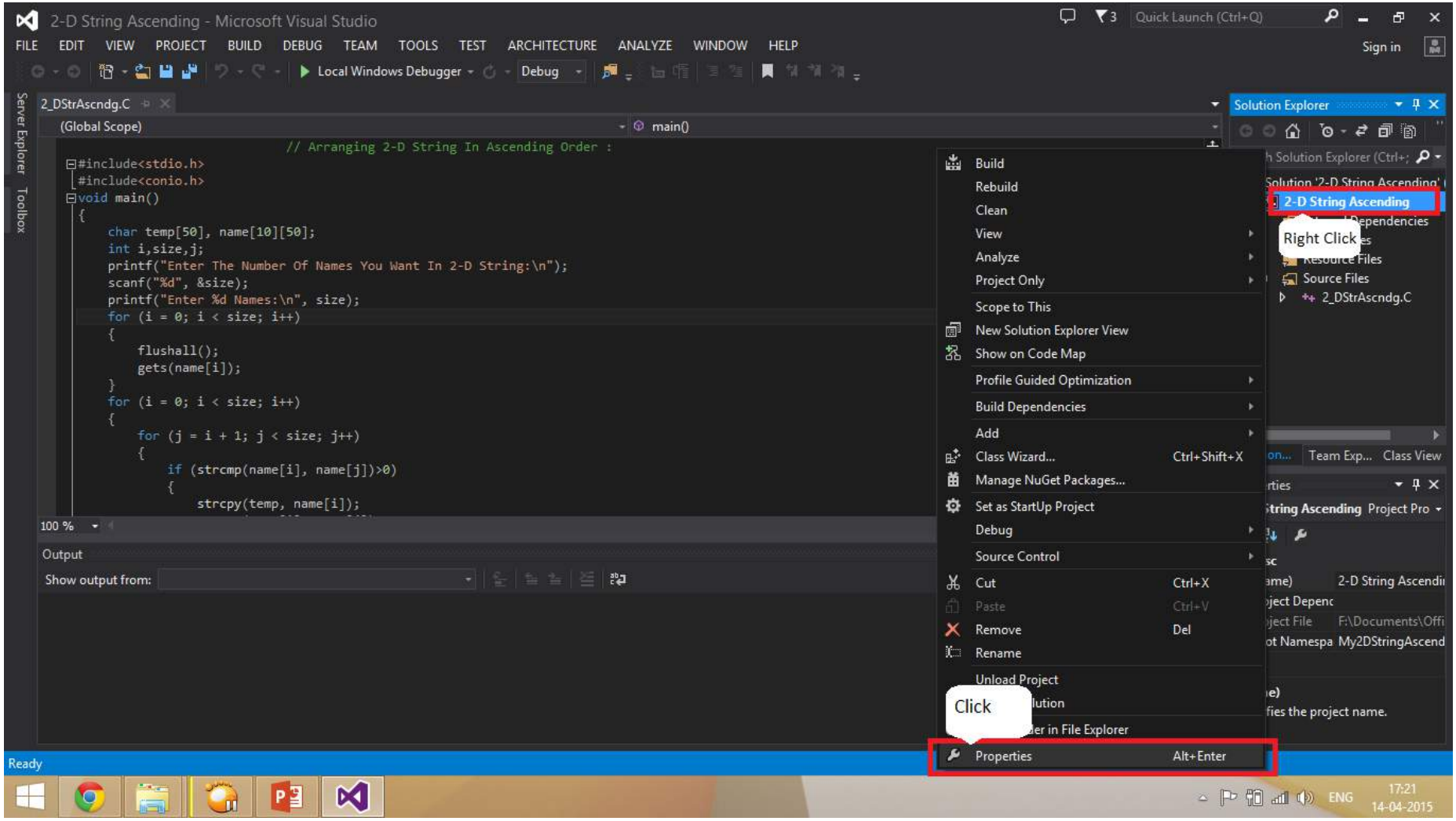
void main(int argc, char *argv[])

- The integer, argc is the **ARGument Count**. It is the number of arguments passed into the program from the command line, including the name of the program.
- The character pointer, *argv[] is the **ARGument Values** list. It has elements specified in the previous argument, argc.

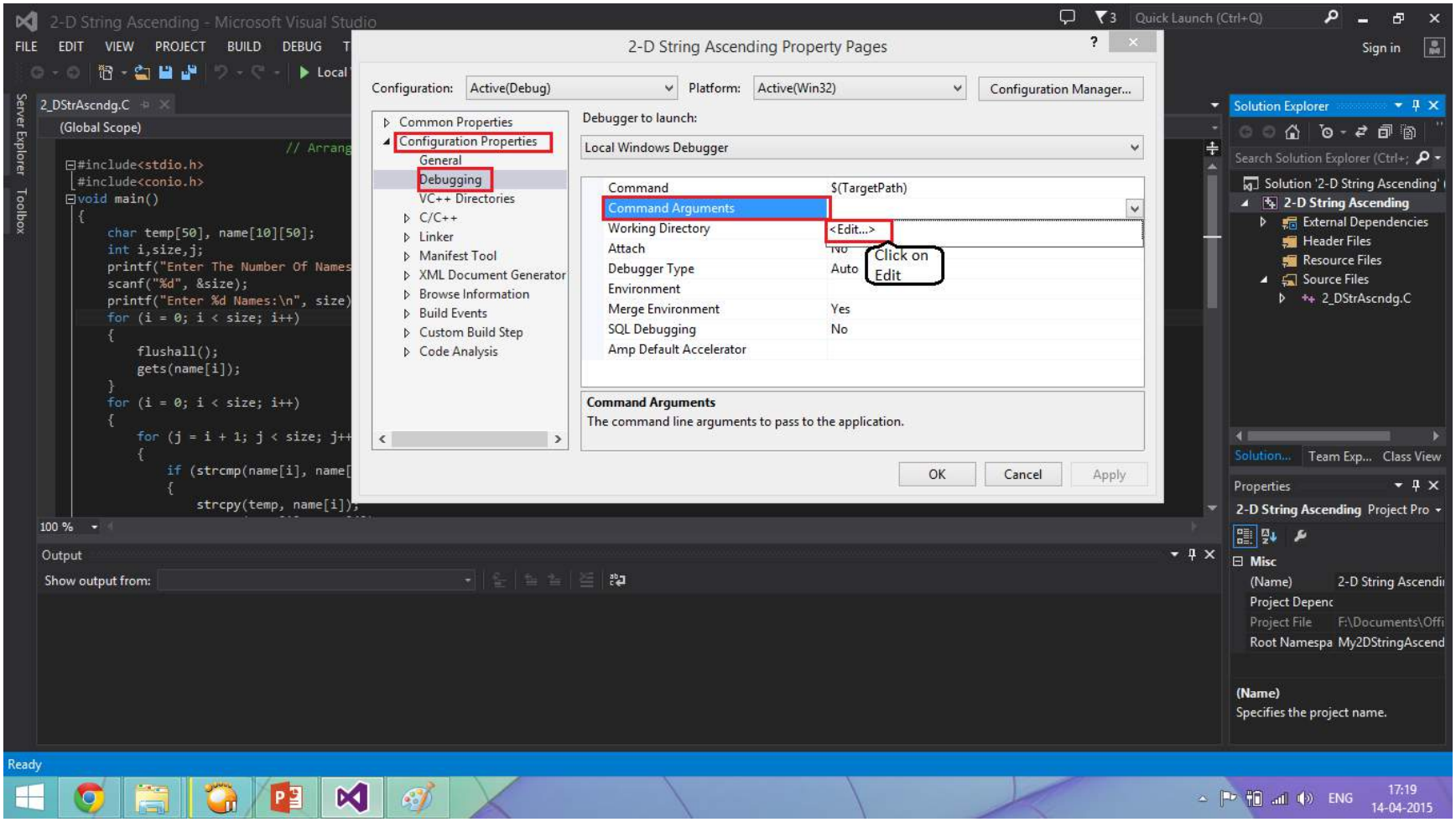
Command Line Arguments

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
void main( int argc, char *argv[])
{
        FILE *fs, *ft;
        char ch;
        if(argc != 3)
        {
                puts("Improper no. of arguments");
                getch();
                exit(1);
        }
        fs = fopen(argv[1], "r");
        if(fs == NULL)
        {
                puts("Cannot open source file");
                getch();
                exit(2);
        }
        ft = fopen(argv[2], "w");
        if(ft == NULL)
        {
                puts("Cannot open target file");
                fclose(fs);
                getch();
                exit(2);
        }
        while(!feof(fs))
        {
                ch = fgetc(fs);
                fputc(ch, ft);
        }
        fclose(fs);
        fclose(ft);
        printf("File copied successfully");
        getch();
}
```

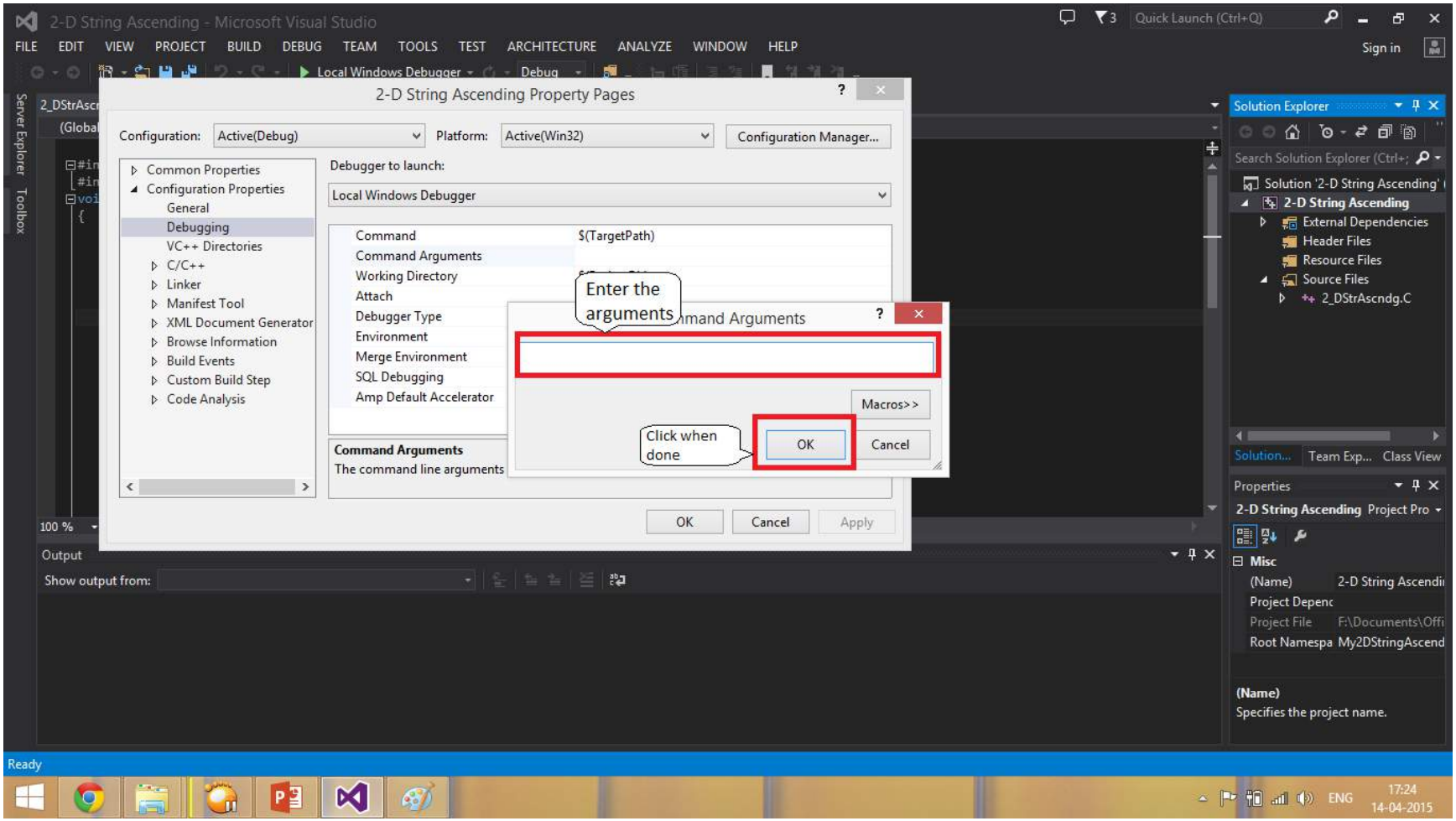
Debugging an "argc-argv" Program Step I



Step 2



Step 3



Continued..

- Execute the program and check the output.
- The arguments will be taken as given by you.
- NOTE: You can give strings as arguments within “ “.
- Make sure that: hi how are you, has 4 arguments whereas: hi “how are” you has just 3.

END OF BASICS IN C

- Thus, the basics of C programming ends here.
- We hope you are satisfied with the theory provided.
- Feel free to share, distribute or use it in any form you wish to. **IT IS FOR YOU**. ☺

END OF BASICS IN C

For advance C programming course or for any doubts in this tutorial, please contact us on any of the following details:

Call us on: 02065000419 / 02065111419

E-mail us at: prog@ecti.co.in

Web URL: www.ecti.co.in

For any suggestions of complaints regarding the theory or programs in the tutorial, write us at:

complaints@ecti.co.in