Sumedh

- Complete C notes
  - handwritten prog
  - highlighted theory

↓

merged ppt's

(11) What is an algorithm? Give its characteristics.

→ An algorithm is a finite sequence of unambiguous, executable steps or instructions, which, if followed would ultimately terminate and give the solution of the problem.

Properties

- Finite: They must eventually terminate
- Complete: Must always give a solution when one exists.
- Correct & Sound: Always give a correct situation

(12) What is a flowchart? Give the symbols / shapes used in flowchart.

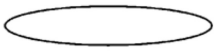
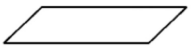
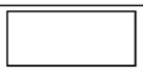
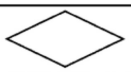
→ A flowchart is a diagramatic representation of a computer program in relation to its sequence of functions.

Symbols as

Terminator

Process

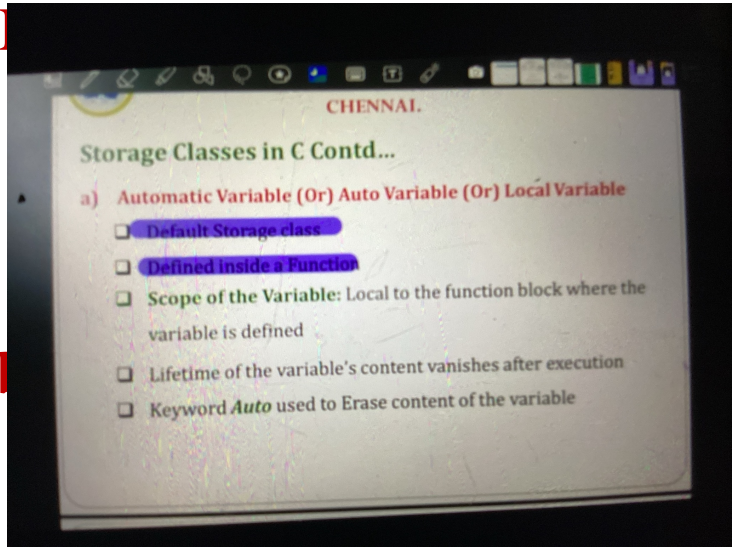
Input / Output

Name	Symbol	Function
Start/End		Used to markup the starting and ending point
Arrows		Used for connection
Input/Output		Used for input and output information
Process		Used to represent single step
Decision		Used for branching or decision making

(13) Discuss pseudocode with and give its importance with an example.

→ Pseudocode is a detailed yet readable description of what a computer program or algorithm must do, expressed in a formally-styled natural language rather than in a programing language.

Pseudocode is similar to programming code and enables the programmer to concentrate on Algorithm. It rather gives a description of the algorithm and if accepted is transformed into actual program code.



Storage Classes in C Contd...

b) External Variable (Or) Global Varial

- Available to all the Functions
- Defined outside the Function
- Keyword ***Declare*** is used to define the global variable (Optional)
- **Scope of the Variable:** Global to all the function blocks
- Lifetime of the variable's content vanishes after the entre program is executed

```
/* Program to Demonstrate External / Global Variables*/
#include<stdio.h>
#include<conio.h>
int n = 10;
void main( )
{
      block1( );
      block2( );
      clrscr( );
      printf("In Main Block n=%d", n);
      getch( );
}
block1( )
{
      printf("In Block 1 n=%d", n);
      return;
}
```

```
block2( )
{
    printf("In Block 2 n=%d", n);
    return;
}
```

Output

```
In Block 1 n=10

In Block 2 n= 10

In Main Block n=10
```

Storage Classes in C Contd…

c) Static Variables

- Keyword ***Static*** is used to define the variable (Compulsory)
- Variable declared as static is initialized to NULL
- Value of the Static variable remains the same throughout the program
- **Scope of the Variable:** Local or Global depending on where it is declared
- **Static Global:** Defined outside the Function
- **Static Local:** Defined inside the Function

```
/* Program to Demonstrate Static Variables*/
#include<stdio.h>
#include<conio.h>
void main( )
{
    int x;
    static int y;
    clrscr( );
    printf("x=%dy=%d", x, y);
    getch( );
}
```

Output

x = 28722

y = 0

Storage Classes in C Contd…

d) Register Variables

- Variables stored in the CPU registers instead of Memory
- Keyword ***Register*** is used to define the variable
- **Scope of the Variable:** Local
- **Advantages**
  - CPU register access is faster than memory access
- **Disadvantages**
  - Number of CPU registers is less
  - Less Number of variables can be stored in CPU registers

```
/* Program to Demonstrate Register Variables*/
#include<stdio.h>
#include<conio.h>
void main( )
{
    register int n=1;
    clrscr( );
    for(n=1; n<=10; n++)
    printf("%d", n);
    getch( );
}
```

Output

1 2 3 4 5 6 7 8 9 10

Sumedh

Q.1] Local and Global declration

Soln ① Variables that are declared inside the function or block is called local declaration

They can be only used by statements inside the function

eg code :-

```
# include <conio.h>
int main ()
{
   int a, b;
   int c,

   a = 10; b = 20;
   c = a+b;
   printf ("value of a = %d, b = %d, c = %d \n", a,b,c)

   return 0;

}
```

② Variables declared outside the function mostly on the top of the program is called as Global declaration

They can be accessed by any function.

eg code :-

```
#include <stdio.h>

int g

int main ()

{
   int a, b;

   a = 10; b = 20;
```

float a;
a = (double)(

```
    g = a+b ;
    printf("value of a = %d , b = %d , g = %d \n", a,b,g);

    return 0;

    }
```

8.2] Bitwise, logical and relational operators with eg. program

Soln ① Relational operators are used to compare two value in C.

Various relational operators are >, <, >=, <=, ==, !=

eg code :-

```
#include <stdio.h>
int main()

{
  int a, b;
  printf (" Enter 2 values \n");
  scanf ( " %d %d", &a , &b);

  printf (" a>b is %d \n" , (a >b));

  printf ( " a<b is %d \n " , (a < b) );

  return 0;

}
```

Output :

```
4
2
a > b  is  1
a < b  is  0
```

② logical operators are used to evaluate truth value of an expression or a condition, either gives the value 0 or 1

Various logical operators && , || , !

eg code :-

```
# include <stdio.h>
int main ()

{
  int age, height ;

  printf (" enter age \n.") ;
  scanf ( "%d", age );

  printf (" enter height \n") ;
  scanf ( "%d", height);

 if ( (age >= 18) && (height >= 5))

    { printf (" Candidate selected ") ; }

   else { printf (" Candidate not selected");}
```

Output :-
```
enter age = 18
enter height = 6
Candidate selected
```

③ In Bitwise operator basic mathematical expressions like addⁿ, subtractⁿ, mul. & div are carried at bit-level in the ALU

Some Bitwise operators are &, |, ~, >>, etc

eg code :-
```
# include <stdio.h>
int main ()
{
    int a = 12, b = 25

  printf ( " output = %d" , a & b);
```

```
/*          128  64  32  16  8  4  2  1
   12       0    0   0   0   1  1  0  0
   25       0    0   0   1   1  0  0  1
&  ---------------------------------------
            0    0   0   0   1  0  0  0    = 8 */

    return 0;

}
```

Output = 8

Q.4] Switch Case statement with example program

Soln Switch Case statement allows to make decisions from a number of choices.

As per evaluated value, the control jumps to the corresponding case.

Syntax :

```
switch (choice)
{
    case 1:     statement; break;
      :
    case n:     Statement; break;
    default :   statements ;
```

eg code :-

```
# include <stdio.h>
 int main()

{
    int a, b , choice

printf (" \n Enter two values ");
scanf (" %d %d", &a , &b);

printf (" \n Enter 1 for Add");

printf (" \n Enter 2 for sub);

printf (" \n Enter 3 for mul);

printf (" \n Enter 4 for div);

printf (" Enter your choice);
scanf ("Choice = %d", choice);

switch ( choice) {

case 1 :
        printf ("sum is %d ", a+b);
        break;

case 2 :
        printf ( " sub is %d", a-b);
        break;

case 3 :
        printf (" mul is %d" , a*b);
        break;

case 4 :
        printf ( " div is %d", a/b);
        break;

 defaut :
        printf ("invalid choice ");

  }
```

```
    getch();
}
```

Output :

```
Enter 2 no's
10
5
Enter 1 for Add
      2     Sub
      3     Mul
      4     div
Enter your choice = 2
  Sub = 5
```

Q.5] Unary, binary and ternary operators with example operators

Soln ① Unary operators are the operators that perform operations on a single operand to produce new value

Some Unary operators are ++, --,

eg code :-

```
# include <stdio.h>
int main()

{
   int i = 1;
   while (i++ < 5) {

        printf ("%d", i);

   }
  return 0;
}
```

② Binary operators are the operators that perform operations on 2 operands to produce a new value

Some Binary operators are >, <, >=, !=, etc

eg code :-

```
# include <stdio.h>
int main()

{
  int a, b;
  printf (" Enter 2 values \n");
  scanf (" %d %d", &a , &b);

  printf (" a>b is %d \n" , (a >b));

  printf (" a<b is %d \n" , (a < b) );

  return 0;

}
```

Output :

```
4
2
a >b  is  1
a< b  is  0
```

③ Ternary operators are operators which require 3 operands to produce a new value.

Also known as Conditional operator and denoted by '?'

```
#include<stdio.h>
int main( )
{
    int x, y;
    scanf("%d", &x);
    y=(x > 5 ? 3 : 4);
    printf("%d", y);
    return 0;
}
```

```
#include<stdio.h>
int main( )
{
    int x, y;
    scanf("%d", &x);
    if(x >5)
        y=3;
    else
        y=4;
    printf("%d", y);
    return 0;
}
```

Operators in C Contd…

- **Classification of Operators**
  a) Increment & Decrement Operators
  b) Comma Operator
  c) Arrow Operator
  d) Assignment Operators
  e) Bitwise Operators
  f) Sizeof Operator

Operators in C Contd...

a) Increment and Decrement Operators Contd...

- **Classification**
  - Pre Increment Operator
  - Post Increment Operator
  - Pre Decrement Operator
  - Post Decrement Operator

Operators in C Contd...

a) Increment and Decrement Operators Contd...

S. No	Operator type	Operator	Description
1	Pre Increment	++i	Value of i is incremented before assigning it to variable i.
2	Post Increment	i++	Value of i is incremented after assigning it to variable i.
3	Pre Decrement	-- i	Value of i is decremented before assigning it to variable i.
4	Post Decrement	i --	Value of i is decremented after assigning it to variable i.

/* Program for Post Increment*/

```
#include<stdio.h>
#include<conio.h>
void main( )
{
      int i = 1;
     while (i++<5)
     {
          printf("%d", i);
     }              }
getch ( );
}
```

Output

1 2 3 4

Operators in C Contd...

b) Comma Operator

- Special operator which separates the declaration of multiple variables
- Has Lowest Precedence i.e it is having lowest priority so it is evaluated at last
- Returns the value of the rightmost operand when multiple comma operators are used inside an expression
- Acts as Operator in an Expression and as a Separator while Declaring Variables

Operators in C Contd...

b) Comma Operator Contd...

```
#include<stdio.h>
int main( )
{
int i, j;
i=(j=10, j+20);
printf("i = %d\n j = %d\n" , i,j );
return 0;
}
```

Operators in C Contd...

c) Arrow Operator (->)

- Arrow operator is used to access the structure members when we use pointer variable to access it
- When pointer to a structure is used then arrow operator is used

c) Arrow Operator (->)

```
#include <stdio.h>

#include<stdlib.h>

struct abc {

int a;};

int main(){

struct abc * g;    g=(struct abc *)malloc(sizeof(struct abc));

g->a=19;

printf("%d",g->a);

return 0;}
```

Operators in C Contd…

d) Assignment Operators

- Assigns result of expression to a variable
- Performs Arithmetic and Assignment operations
- Commonly used Assignment operator: **=**
- ***Syntax*** ***variable = expression;***
- ***Examples***
  - num = 25; age = 18; pi = 31.4; area = 3.14 * r * r;

Operators in C Contd...

- Shorthand Assignment Operators

Simple Assignment Operator	Shorthand Operator
a = a + 1	a+=1
a = a – 1	a-=1
a = a * 2	a*=2
a = a / b	a/=b
a = a % b	a%=b
c = c * (a + b)	c *= (a + b)
b = b / (a + b)	b /=(a + b)

/* Program for Assignment Operations*/

```
#include<stdio.h>
#include<conio.h>
void main( )
{
      int a;
     a = 11;
     a+ = 4;
          printf("Value of A is %d\n",a);
     a = 11;
     a- = 4;
          printf("Value of A is %d\n",a);
     a = 11;
     a* = 4;
          printf("Value of A is %d\n",a);
     a = 11; a/ = 4;
```

Datatypes Contd…

a) Integer Data Type

- Whole numbers with a range
- No fractional parts
- Integer variable holds integer values only
- **Keyword:** int
- **Memory:** 2 Bytes (16 bits) or 4 Bytes (32 bits)
- **Qualifiers:** Signed, unsigned, short, long
- **Examples:** 34012, 0, -2457

Datatypes Contd…

a) Integer Data Type

```
#include<stdio.h>
int main()
{
  int a;
scanf("%d",&a);
printf("%d",a);
return 0;}
```

Datatypes Contd…

b) Floating Point Data Type

- Numbers having Fractional part
- Float provides precision of 6 digits
- Integer variable holds integer values only
- **Keyword:** float
- **Memory:** 4 Bytes (32 bits)
- **Examples:** 5.6, 0.375, 3.14756

Datatypes Contd…

b) Floating Point Data Type

```
#include<stdio.h>
int main()
{
 float g;
scanf("%f",&g);
printf("%f",g);
return 0;
}
```

Datatypes Contd…

c) Double Data Type

- Also handles floating point numbers
- Double provides precision of 14 digits
- Integer variable holds integer values only
- **Keyword:** double
- **Memory:** 8 Bytes (64 bits) or 10 Bytes (80 bits)

Datatypes Contd…

c) Double Data Type

```
#include<stdio.h>
int main()
{
   double g;
  scanf(“%ld”,&g);
printf(“%ld”,g);
return 0;
}
```

Datatypes Contd…

d) Character Data Type

- handles one character at a time
- **Keyword: char**
- **Memory:** 1 Byte (8 bits)

Datatypes Contd…

d) Character Data Type

```
#include<stdio.h>
int main()
{
 char g;
scanf("%c",&g);
printf("%c",g);
return 0;
}
```

Conditional Statements/decision-making statements

1. **Simple if statement**

```
Odd*/ #include<stdio.h>
int main( )
{
    int number;
    printf("Enter the Number:
    "); scanf("%d, &number);
    if(number%2==0)
    {
        printf("The Number is Even");
    }
    return 0;
}
```

Output

```
Enter a value :
10342 The
number is Even
```

2. **if..else Statement**

Example:// if..else

```
#include<stdio.h>
void main()
{
int age=21;
if(age>18)
{
printf("Eligible to Vote");
}
else
{
printf("Not eligible to Vote");
}
}
```

iii. Nested if else statement

Example:

```
#include <stdio.h>
int main () {
   int a = 100;
   int b = 200;
   if( a == 100 ) {
        if( b == 200 ) {
        printf("Value of a is 100 and b is 200\n" );
     }
   }
     printf("Exact value of a is : %d\n", a );
   printf("Exact value of b is : %d\n", b );
   return 0;
}
```

Output:

```
Value of a is 100 and b is 200
Exact value of a is : 100
Exact value of b is : 200
```

iv. else if

Example:// simple if

```
#include<stdio.h>
void main()
{
int m;
printf("Input m value");//Month
scanf("%d",&m);
if(m==1 || m==2 || m==12)
{
printf("Winter");
}
else if(m==3 || m==4 || m==5)
{
printf("Spring");
}
else if(m==6 || m==7 || m==8)
{
printf("Summer");
}
```

```
else if(m==9 || m==10 || m==11)
{
printf("Autumn");
}
else
{
printf("Invalid Month");
}
}
```

Output:
Input m value5
Spring

v. switch case

Example: // Switch case

```
#include<stdio.h>
int main()
{
int day;
printf("Enter the day\n");
scanf("%d",&day);
switch(day)
{
case 1:
printf("Sunday");
break;
case 2:
printf("Monday");
break;
case 3:
printf("Tuesday");
break;
case 4:
printf("Wednesday");
break;
case 5:
printf("Thursday");
break;
case 6:
printf("Friday");
break;
case 7:
printf("Saturday");
break;
default:
printf("Invalid day");
}
}
```

Output:

```
Enter the day
1
Sunday
```

Looping Control Statements

i. for

Example:

```
#include<stdio.h>
void main()
{
int i,num=5;
for(i=1;i<=10;i++)
{
printf("%d*%d=%d\n",i,num,(i*num));
}
}
```

```
1*5=5
2*5=10
3*5=15
4*5=20
5*5=25
6*5=30
7*5=35
8*5=40
9*5=45
10*5=50
```

ii. While

Example:

```
#include<stdio.h>
void main()
{
int x=1;
while(x<=10)
{
printf("%d\n",x);
x++;
}
}
```

```
1
2
3
4
5
6
7
8
9
10
```

iii. Do…while

Example:

```
#include<stdio.h>
void main()
{
int i=18;
do
{
printf("i=%d: \n",i);
i=i+1;
}while(i<20);
}
```

```
i=18:
i=19:
```

Un-conditional Control Statements

i. Break

Example-2: // break

```
#include<stdio.h>
void main()
{
for(int i=1;i<=5;i++)
{
   if(i==3)
   {
      break;
   }
printf("%d\n",i);
}
}
```

Output:
1
2

ii. Continue

Example:

```
#include<stdio.h>
void main()
{
for(int i=1;i<=5;i++)
{
   if(i==3)
   {
      continue;
   }
printf("%d\n",i);
}
}
```

Output:
1
2
4
5

iii. Goto

Example:

```
#include <stdio.h>
int main()
{
 int num,i=1;
 printf("Enter the number whose table you want to print?");
 scanf("%d",&num);
 table:
 printf("%d x %d = %d\n",num,i,num*i);
 i++;
 if(i<=10)
 goto table;
}
```

If the condition is true, then control is transferred to label

Enter the number whose table you want to print?7

7 x 1 = 7
7 x 2 = 14
7 x 3 = 21
7 x 4 = 28
7 x 5 = 35
7 x 6 = 42
7 x 7 = 49
7 x 8 = 56
7 x 9 = 63
7 x 10 = 70

2. 3 Arrays Contd...

a) Array Declaration

- Syntax

Datatype arrayname [size/subscript];

- **Data Type:** int, float, double, char, structure, union
- **Array Name:** Name given to the Array variable
- **Size / Subscript:** Number of values an Array can hold
- **Examples**

int numbers[5]; float marks[50];

char name[20]; double a[i];

2. 3 Arrays Contd...

b) Array Initialization

- **Initialization:** Assigning values to array elements
  - Values specified in curly braces separated by commas
- Examples

```
int a[ 5] = {1, 2, 3, 4, 5};

float b[3] = { 40.5, 59.0, 98.5};

char name[6] = ” SRMIST”;
```

- Array element index start from 0

2. 3 Arrays Contd...

c) Getting Input for Arrays

- Use for loops to get input in arrays
- Use for loops with regard to the Array's dimension
- Input for One Dimensional Arrays – 1 for loop

```
for(i = 0; i < 5; i++)
{
        scanf("%d", &a[i]);
}
```

2. 3 Arrays Contd...

- Input for Two Dimensional Arrays – 2 for loops

```
for(i=0;i<5;i++)
    {
        for(j=0;j<5;j++)
        {
            scanf("%d",&a[i][j]);
        }
    }
```

2. 3 Arrays Contd...

d) Printing Output in Arrays

- Use for loops to print array output
- Use for loops with regard to the Array's dimension
  - Printing One Dimensional Array Output – 1 for

```
loop  for(i=0;i<5;i++)

{

        printf("%d",a[i]);

}
```

2. 3 Arrays Contd...

- Printing Two Dimensional Array Output – 2 for loops

```
for(i = 0; i < 5; i++)
    {
        for(j=0; j < 5; j++)
        {
            printff("%d", a[i][j]);
        }
    }
```

2. 3 Arrays Contd...

i. One Dimensional Array

- Data stored under a single variable using one subscript
- 1-D Array Declaration – Syntax

datatype arrayname [size/subscript];

- **Example:** int a [5];

- 1-D Array initialization – Syntax

datatype arrayname [size] = { list of values};

Example: int a [5] = { 10, 20, 30, 40, 50};

2. 4 - Initializing and Accessing 2D Array

Initialization of 2D Array in C

In the 1D array, we don't need to specify the size of the array if the declaration and initialization are being done simultaneously.

However, this will not work with 2D arrays. We will have to define at least the second dimension of the array. The two-dimensional array can be declared and defined in the following way.

```
int arr[4][3]={{1,2,3},{2,3,4},{3,4,5},{4,5,6}};
```

2. 5 - Pointer and address of operators

A pointer variable is another variable that holds the address of the given variable to be accessed. Pointers provide a way of accessing a variable with out referring to variable directly.

- Declaration: type *ptr_name;
- Initialiazation: int x, *p = & x;

Gives Address

- A pointer should be initialized before use.
- The unary operator * is termed as dereferencing operator or indirection operator, as it allows to access the value of a variable indirectly.

2. 5 - Pointer and address of operators

Types of pointer

Null pointer: Null pointer is a constant with a value of zero defined in several standard libraries.

Dangling pointer: If any pointer is pointing to any address at given point in a program and later on the variable has been deleted from that specific memory location, although the pointer is still referencing to the memory location. These pointers are called dangling pointers

2. 5 - Pointer and address of operators

Genetic Pointers: Pointers which doesn't have any specific data type is known as genetic pointers.

Eg: void *the_data;

Wild Pointers: Uninitialized pointers are known as wild pointers because they point to some arbitrary memory location and may cause a program to crash or behave badly.

Eg: int main()
{
Int *ptr;
Printf("%d", *ptr);
}

/* Array Using Pointers */

```
#include<stdio.h>
Int main()
{
int data[5],I;
printf("Enter elements: ");
for(i=0;i<5;++i)
scanf("%d", data + i);
printf("you entered: \n");
for(i=0; i<5; i++)
printf("%d", *(data + i));
return 0;
}
```

```
Output:
Enter elements: 1
2
3
5
4

You entered: 1
2
3
5
4
```

2. 7 – Void Pointers

- A void pointer is a pointer that has no associated data type with it. A void pointer can hold address of any type and can be typecasted to any type.

```
#include<stdio.h>
int main()
{
   int a[2] = {1, 2};
   void *ptr = &a;
   ptr = ptr + sizeof(int);
   printf("%d", *(int *)ptr);
   return 0;
}
```

Output: 2

2. 9 – Pointer based Array Manipulation

Pointer and Arrays in C

Replacing the printf("%d", *p); statement of the above example, with below mentioned statements. Lets see what will be the result.

printf('%d", a[i]); ⇨ prints the array, by incrementing index

printf('%d", i[a]); ⇨ this will also print elements of array

printf('%d", a+i); ⇨ this will prints address of all the array elements.

printf('%d", *(a+i)); ⇨ will print value of array element

printf('%d", *a); ⇨ will print value of a[0] only

a++ ⇨ compile time error, we cannot change the base address of the array.

2. 9 – Pointer based Array Manipulation

Array of Pointers

Pointers are very helpful in handling character arrays with rows of varying lengths.

```
char *name[3] = {
                    "Adam",
                    "chris",
                   "Deniel“
};
//without pointer
char name[3][20] = {"Adam",    "chris",    "Deniel"};
```

SRM Institute of Science and Technology, Chennai

21CSS101J –Programming for Problem Solving
Unit-III

SRM Institute of Science and Technology, Chennai

Prepared by:

Dr. P. Robert

Assistant Professor

Department of Computing Technologies

SRMIST-KTR

SRM Institute of Science and Technology, Chennai

LEARNING RESOURCES	
S. No	**TEXT BOOKS**
1.	*Zed A Shaw, Learn C the Hard Way: Practical Exercises on the Computational Subjects You Keep Avoiding (Like C), Addison Wesley, 2015*
2.	*W. Kernighan, Dennis M. Ritchie, The C Programming Language, 2nd ed. Prentice Hall, 1996*
3.	*Bharat Kinariwala, Tep Dobry, Programming in C, eBook*
4.	*http://www.c4learn.com/learn-c-programming-language/*

Unit-III

String Basics - String Declaration and Initialization - String Functions: gets(), puts(), getchar(),putchar(), printf() - Built-inString Functions: atoi, strlen, strcat, strcmp -String Functions: sprint, sscanf, strrev, strcpy, strstr, strtok - Operations on Strings - Function prototype declaration, function definition - Actual and formal parameters - Function with and without Arguments - Function with and without return values - Call by Value, Call by Reference - Passing Array to Function - Passing Array elements to Function - Function Pointers.

String Basics

In C programming, a string is a sequence of characters terminated with a null character \0 .

For example: char c[] = "c string";

When the compiler encounters a sequence of characters enclosed in the double quotation marks, it appends a **null character \0** at the end by default.

In C language strings are stored in array of char type along with **null terminating character \o**.

Synatx:

char fname[4];

The above statement declares a string called fname that can take up to 3 characters.

fname[]={'J','O','E'}

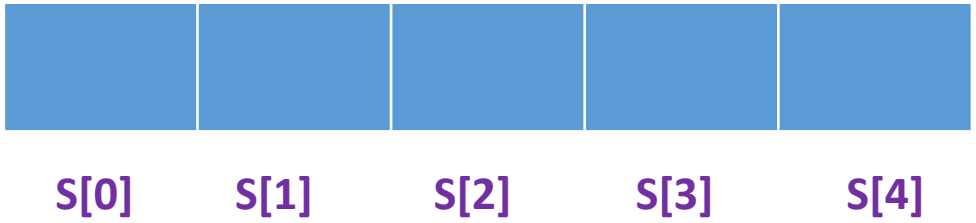
J O E \0

char name[10]; can store maximum of 9 characters.

String Declaration and Initialization

How to declare a string?

char s[5];



How to initialize strings?

char s[5];

char s[] = "abcd";

char s[50] = "abcd";

char s[] = {'a', 'b', 'c', 'd', '\0'};

char s[5] = {'a', 'b', 'c', 'd', '\0'};

String Functions

1. gets()
2. puts()
3. getchar()
4. putchar()
5. Printf()

String Functions

1. gets()

It is used to take a single input at a time but can be used to input a complete sentence with spaces unlike scanf(). It stops reading character when the newline character is read or end-of the file is reached.

Declaration

char *gets(char *str)

str- This is the pointer to an array of characters where the C string is stored.

Return Value:

This function returns str on success, and NULL on error or when end of file occurs.

String Functions

Example:

```
#include <stdio.h>

int main () {

   char str[50];

   printf("Enter a string : ");

   gets(str);

   printf("You entered: %s", str);

   return(0);

}
```

Output:

```
Enter a string : Robert Jose
You entered: Robert Jose
```

String Functions

2. puts()

This function writes strings or lines to stdout, i.e, the output stream. The string is printed with newline and an integer value is returned.

Declaration

int puts(const char* string)

Example:

```
# include<stdio.h>
int main(){
        // Initializing the string.
        char string[] = "SRM University";
        // Writing our string to stdout.
        puts(string);
    return 0;
}
```

Output:

```
SRM University
```

String Functions

3. getchar()

The getchar() function is the part of the <stdio.h> header file in C. It is used when **single character** input is required from the user.

Note:

The function reads the input as an unsigned char; then it casts and return as an int or an EOF.

EOF is returned if the end of the file is reached or an error is occurred.

Declaration

int getchar(void)

The getchar function takes **no parameter**.

String Functions

Example:

```
#include<stdio.h> // Including header file
 int main(){
  int myChar; // creating a variable to store the input
  myChar = getchar(); // use getchar to fetch input
  printf("You entered:  %c", myChar); // print input on screen
  return 0;
}
```

Output:

```
a
You entered:  a
```

String Functions

4. putchar()

This function is used for printing character to a screen at current cursor location.

Declaration

putchar(character_variable);

Example:

```
#include<stdio.h>
void main()
{
  int ch;
  printf("Press any character\n");
  ch = getchar();
  printf("Pressed character is\n");
  putchar(ch);
  getch(); /* Holding output */
}
```

Output:

```
Press any character
a
Pressed character is
a
```

String Functions

5. Printf()

It is one of the main output function. This function sends formatted output to the screen.

Declaration

Printf("format String", argument list);

The format string can be

%d	Integer
%c	Character
%f	Float
%s	String

String Functions

5. Printf()

Example:

```
#include<stdio.h>

int main(){

int num;

printf("Enter a number:");

scanf("%d",&num);

printf("Cube of number is:%d ",num*num*num);

return 0;

}
```

Output:

```
Enter a number:3
Cube of number is:27
```

Built-in String Functions

- atoi
- strlen
- strcat
- strcmp

Built-in String Functions

❖ atoi

The atoi() function **converts a character string to an integer value**. The input string is a sequence of characters that can be interpreted as a numeric value of the specified return type. The function stops reading the input string at the first character that it cannot recognize as part of a number.

Syntax:

int atoi(const char *str)

Where

str- String passed to the function

Return value:

If str is a valid input, the function returns the integer number equal to the passed string number. If str has no valid input, the functions return zero value.

Built-in String Functions

Example:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main()
{
   int val1,val2;
   char string1[20] = "Jose";
   val1 = atoi(string1);
   printf("String value = %s\n", string1);
   printf("Integer value = %d\n", val1);
   char string2[20] = "1984";
   val2 = atoi(string2);
   printf("String value = %s\n", string2);
   printf("Integer value = %d\n", val2);
   return (0);
}
```

Output:

```
String value = Jose
Integer value = 0
String value = 1984
Integer value = 1984
```

Built-in String Functions

❖ strlen

The strlen() function returns the length of the given string. It doesn't count null character '\0'.

Syntax:

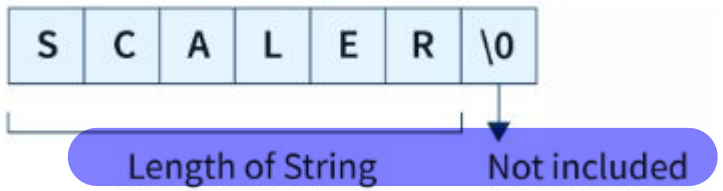
int strlen(const char *str)

Where

Str is the parameter passed to the function.

Return value:

The length of the string is returned in integer type excluding the NULL character.



Built-in String Functions

Example:

```
#include<stdio.h>

#include<string.h>

int main() {

  char name[10] = "Jordin";

  printf("The length of the string is %d", strlen(name));

  return 0;

}
```

Output:

```
The length of the string is 6
```

Built-in String Functions

❖ strcat

The strcat(first_string, second_string) function concatenates two strings and result is returned to first_string.

Syntax:

char *strcat(char *dest, const char *src)

Where

dest- The pointer to the destination array

src- The pointer to the source array

Return value:

This function returns a pointer to the resulting string dest.

Built-in String Functions

Example:

```
#include <stdio.h>

#include <string.h>

int main() {

   char str1[100] = "Welcome to  ", str2[] = "India";

   strcat(str1, str2);

   puts(str1);

   puts(str2);

   return 0;

}
```

Output:

```
Welcome to  India
India
```

Built-in String Functions

❖ **strcmp**

Strings can be compared either by using the string function or without using string function. The string function which is pre-defined in a **string.h** header file is a **strcmp()** function. The strcmp() function consider two strings as a parameter, and this function returns an integer value where the integer value can be **zero**, **positive** or **negative**.

Syntax:

int strcmp (const char* str1, const char* str2);

Where

Str1- a string

Str2 – a string

Return value

0- If both strings are equal

> 0- It returns a value greater than zero when the matching character of left string has greater ASCII value than the character of the right strin

< 0 - It returns a value less than zero when the matching character of left string has lesser ASCII value than the character of the right string

Built-in String Functions

Example:

```
#include<stdio.h>
#include<string.h>
int main() {
   char str1[] = "Hello World!";
   char str2[] = "hello World!";
   int result = strcmp(str1, str2);
   if (result==0)
   printf("Strings are equal");
   else
   printf("Strings are unequal");
   printf("\nValue returned by strcmp() is: %d" , result);
   return 0;
}
```

Output:

```
Strings are unequal
Value returned by strcmp() is: -32
```

String Functions

- sprintf
- sscanf
- strrev
- strcpy
- strstr
- strtok

❖ sprintf

sprintf stands for "string print". In C programming language, it is a file handling function that is used to send formatted output to the string. Instead of printing on console, sprintf() function stores the output on char buffer that is specified in sprintf.

Example:

```
#include <stdio.h>
int main()
{
   char buffer[50];
   int a = 15, b = 25, res;
   res = a + b;
   sprintf(buffer, "The Sum of %d and %d is %d", a, b, res);
   printf("%s", buffer);
   return 0;
}
```

Output:

```
The Sum of 15 and 25 is 40
```

❖ sscanf

In C, sscanf() is used to read formatted data. It works much like scanf() but the data will be read from a string instead of the console.

Return Value

The function returns an int value which represents the total number of items read.

Given there was an error and the string could not be read, an EOF error is returned.

Any other type of error which is encountered while reading is denoted by the function returning -1.

Example:

```
#include<stdio.h>
int main() {
    char* buffer = "Hello";
    char store_value[10];
    int total_read;
    total_read = sscanf(buffer, "%s" , store_value);
    printf("Value in buffer: %s",store_value);
    printf("\nTotal items read: %d",total_read);
    return 0;
}
```

Output:

```
Value in buffer: Hello
Total items read: 1
```

❖ strrev

The strrev(string) function returns reverse of the given string.

Example:

```
#include <stdio.h>
#include <string.h>
int main()
{
   char str[] = "OMAN"; // initialzing a char array
   printf("The string is : %s\n", str); // printing the actual array
   strrev(str); // reversing the char array
   printf("The string after using function strrev() is : %s\n", str); // printing the reversed array
   return 0;
}
```

❖ strcpy

The strcpy(destination, source) function copies the source string in destination.

Example:

```
#include<stdio.h>
#include <string.h>
int main(){
 char ch[20]="Oman";
   char ch2[20];
   strcpy(ch2,ch);
   printf("Value of second string is: %s",ch2);
 return 0;
}
```

Output:

```
Value of second string is: Oman
```

❖ strstr

The strstr() function returns pointer to the first occurrence of the matched string in the given string. It is used to return substring from first match till the last character.

Example:

```
#include<stdio.h>
#include <string.h>
int main(){
  char str[100]="types of machine learning algorithms";
  char *sub;
  sub=strstr(str,"machine");
  printf("\nSubstring is: %s",sub);
 return 0;
}
```

Output:

```
Substring is: machine learning algorithms
```

❖ strtok

The C string manipulation function ***"strtok()"*** is used to split a string into ***tokens*** or smaller strings based on a ***designated delimiter***. The ***"string.h"*** header file contains the function's declaration.

Example:

```
#include <stdio.h>
#include <string.h>
int main() {
   char str[] = "Delhi,Hyderabad,Noida";
   char *token;
   token = strtok(str, ",");
   while(token != NULL) {
      printf("%s\n", token);
      token = strtok(NULL, ",");
   }
    return 0;
}
```

Output:

```
Delhi
Hyderabad
Noida
```

In the above example, the ***comma (,)*** is used as a delimiter to separate the ***str*** string into smaller tokens. The ***strtok() function*** is first invoked with ***str*** as its first argument. The function returns a pointer to the ***first token***, which is kept in the token variable. Once all tokens have been extracted, the loop resumes by calling ***strtok()*** with the delimiter as the ***second argument*** and a ***NULL*** for the ***first argument***. The ***NULL first argument*** tells ***strtok()*** to pick up where the previous call left off.

Function Prototype Declaration

A function is a block of code which only runs when it is called. You can pass data, known as parameters, into a function. Functions are used to perform certain actions, and they are important for reusing code: Define the code once, and use it many times.

Predefined Function

```
int main() {
  printf("Hello World!");
  return 0;
}
```

For example, main() is a function, which is used to execute code, and printf() is a function; used to output/print text to the screen:

Function Prototype Declaration

Function Prototype Declaration

```
return_type function_name(parameter list) {
  // function body
}
```

The return_type specifies the type of value that the function will return. If the function does not return anything, the return_type will be void.

The function_name is the name of the function, and the parameter list specifies the parameters that the function will take in.

How to Declare a Function in C

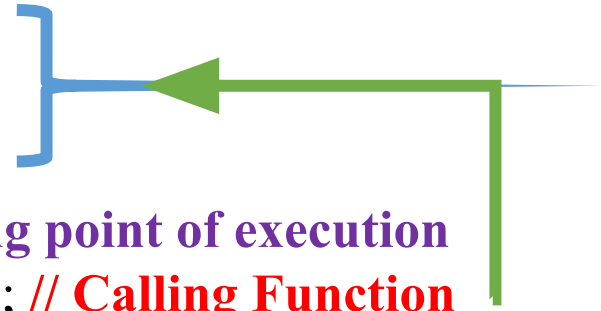
```
return_type function_name(parameter_list);
```

Example: `int add(int num1, int num2);`

Function Prototype Declaration

Example:

```
#include <stdio.h>
/* function statement */
int add(int a, int b);  //Function declaration
/* function definition */
int add(int a, int b) {
    return a + b;
}
int main() {   // Starting point of execution
    int result = add(2, 3); // Calling Function
    printf("The result is %d\n", result);
    return 0;
}
```



Function definition:

It contains the actual statements which are to be executed. It is the most important aspect to which the control comes when the function is called. Here, we must notice that only one value can be returned from the function.

Function Prototype Declaration

Function Declaration:

A function must be declared globally in a c program to tell the compiler about the function name, function parameters, and return type.

Function Call:

Function can be called from anywhere in the program. The parameter list must not differ in function calling and function declaration. We must pass the same number of functions as it is declared in the function declaration.

Actual Parameters

Actual parameters are values that are passed to a function when it is invoked.

Formal Parameters

Formal parameters are the variables defined by the function that receives values when the function is called.

Actual and Formal Parameters

Difference between Actual and Formal Parameters

Actual vs Formal Parameters	
The Actual parameters are the values that are passed to the function when it is invoked.	The Formal Parameters are the variables defined by the function that receives values when the function is called.
Related Function	
The actual parameters are passed by the calling function.	The formal parameters are in the called function.
Data Types	
In actual parameters, there is no mentioning of data types. Only the value is mentioned.	In formal parameters, the data types of the receiving values should be included.

Actual and Formal Parameters

Example

```
#include <stdio.h>
void addition (int x, int y) { // Formal parameters
int addition;
addition = x+y;
printf("%d",addition);
}
void main () {
addition (2,3); //actual parameters
addition (4,5); //actual parameters
}
```

Different aspects of function calling

Example

```
#include<stdio.h>
void printName();
void main ()
{
   printf("Hello ");
   printName();  // function without arguments and without return value
}
void printName()
{
   printf("Students");
}
```

Output:

```
Hello Students
```

Different aspects of function calling

Example

```
#include<stdio.h>
int sum();
void main()
{
    int result;
    printf("\nGoing to calculate the sum of two numbers:");
    result = sum();  //Function without arguments and with return value
    printf("%d",result);
}
int sum()  //Function definition
{
    int a,b;
    printf("\nEnter two numbers");
    scanf("%d %d",&a,&b);
    return a+b;
}
```

Output:

```
Going to calculate the sum of two numbers:
Enter two numbers 2 3
5
```

Different aspects of function calling

Example

```
#include<stdio.h>
int sum();
void main()
{
    int result;
    printf("\n sum of two numbers:");
    result = sum(2,3);  //Function with arguments and with return value
    printf("%d",result);
}
int sum(int a,int b)
{
    return a+b;
}
```

Output:

```
sum of two numbers:5
```

Different aspects of function calling

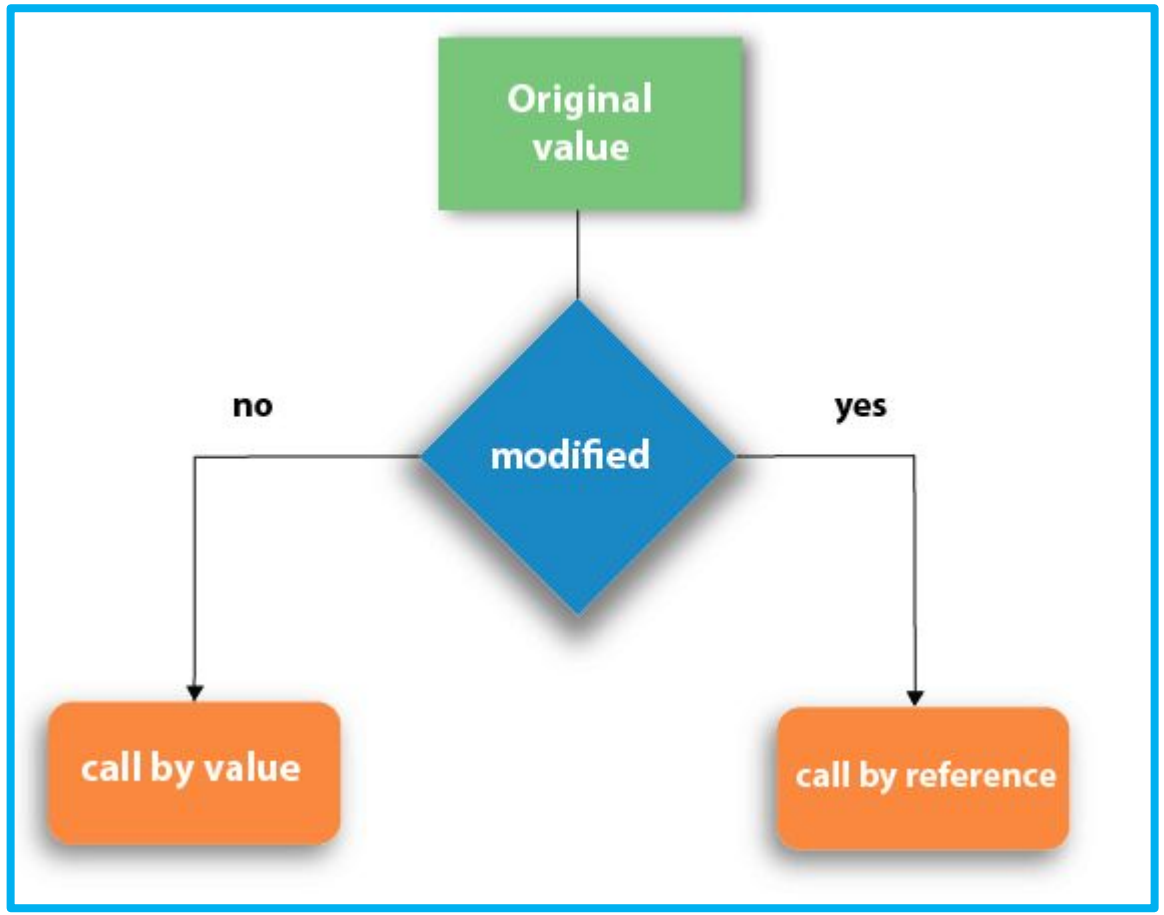
Example

```
#include<stdio.h>
void sum();
void main()
{
   int result;
   sum(2,3);  //Function with arguments and no return value
}
void sum(int a,int b)
{
   printf("Addition of two numbers %d",(a+b));
}
```

Output:

```
Addition of two numbers 5
```

Call by Value & Call by Reference



Call by Value

The call by value method of passing arguments to a function copies the actual value of an argum
the formal parameter of the function. In this case, changes made to the parameter inside the f
have no effect on the argument.

Example:

```
#include <stdio.h>
void swap(int x, int y);
int main () {
   int a = 100;
   int b = 200;
   printf("Before swap, value of a : %d\n", a );
   printf("Before swap, value of b : %d\n", b );
   swap(a, b);
   printf("After swap, value of a : %d\n", a );
   printf("After swap, value of b : %d\n", b );
   return 0;
}
void swap(int x, int y) {
   int temp;
   temp = x; /* save the value of x */
   x = y;    /* put y into x */
   y = temp; /* put temp into y */
   return 0;  }
```

Output:

```
Before swap, value of a : 100
Before swap, value of b : 200
After swap, value of a : 100
After swap, value of b : 200
```

Call by Reference

The **call by reference** method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. It means the changes made to the parameter affect the passed argument.

Example:

```
#include <stdio.h>
int main () {
   int a = 100;
   int b = 200;
   printf("Before swap, value of a : %d\n", a );
   printf("Before swap, value of b : %d\n", b );
   swap(&a, &b);
   printf("After swap, value of a : %d\n", a );
   printf("After swap, value of b : %d\n", b );
   return 0;
}
void swap(int *x, int *y) {
   int temp;
   temp = *x; /* save the value of x */
   *x = *y;    /* put y into x */
   *y = temp; /* put temp into y */
   return 0; }
```

Output:

```
Before swap, value of a : 100
Before swap, value of b : 200
After swap, value of a : 200
After swap, value of b : 100
```

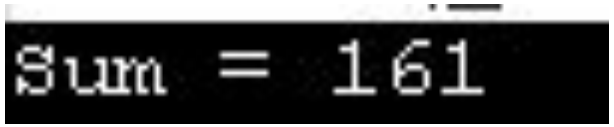
Why do we need functions in C programming?

- Enables reusability and reduces redundancy
- Makes a code modular
- Provides abstraction functionality
- The program becomes easy to understand and manage
- Breaks an extensive program into smaller and simpler pieces

Passing array to functions

```
#include <stdio.h>
int Sum(int num[]);
int main() {
int result, num[] = {23, 55, 22, 3, 40, 18};
  result = Sum(num); //Passing array to function
  printf("Sum = %d", result);
  return 0;
}
int Sum(int num[]) {
  int  sum = 0;
  for (int i = 0; i < 6; ++i) {
    sum += num[i];
  }
  return sum;
}
```

Output:



Passing array elements to functions

```
#include <stdio.h>
void display(int age1, int age2) {
  printf("%d\n", age1);
  printf("%d\n", age2);
}
int main() {
  int ageArray[] = {2, 8, 4, 12};
  // pass second and third elements to display()
  display(ageArray[1], ageArray[2]);
  return 0;
}
```

Output:

Function Pointers

A block of memory is reserved by the compiler when we declare a variable. To store these memory addresses, C allows programmers to use the concept of pointers that can hold the address of any other data type. Pointers can be de-referenced using the **asterisk *** operator to get the value stored in an address.

Function Pointers

A block of memory is reserved by the compiler when we declare a variable. To store these memory addresses, C allows programmers to use the concept of pointers that can hold the address of any other data type. Pointers can be de-referenced using the **asterisk *** operator to get the value stored in an address.

Declaring a Function Pointer in C

return_type (* pointer_name) (datatype_arg_1, datatype_arg_1, ...);
float (*ptr) (int, int);
float add(int, int);
ptr=add;
Here, pointer *ptr is a function pointer and stores the memory address of a function add that takes two arguments of type int and returns a value of data type float.

Function Pointers

Example:

```
#include<stdio.h>
int areaRectangle(int, int);
int main() {
    int length, breadth, area;
    int (*fp)(int, int);
    printf("Enter length and breadth of a rectangle\n");
    scanf("%d%d", &length, &breadth);
    fp = areaRectangle;
    area = (*fp)(length, breadth);
    printf("Area of rectangle = %d", area);
    return 0;
}
int areaRectangle(int l, int b) {
    int area_of_rectangle = l * b;
    return area_of_rectangle;
}
```

Output:

Output:

```
Enter length and breadth of a rectangle
2 3
Area of rectangle = 6
```

Thank You