Algorithm and Flowchart

Pratik Shah

II-CSE (Div A)

02/01/25 – 21/04/25

Algorithm

- Algorithm is a step – by – step procedure which is helpful in solving a problem.
- If, it is written in English like sentences then, it is called as ‘PSEUDO CODE’.
- An algorithm is a finite set of instructions, if followed and accomplishes a particular task.
- It is a sequence of computational steps that transform the input into a valuable or required output.

Properties of an Algorithm

- An algorithm must possess the following five properties −
  - Input
  - Output
  - Finiteness
  - Definiteness
  - Effectiveness

1. **Input:** An algorithm should have some inputs.

2. **Output:** At least one output should be returned by the algorithm after the completion of the specific task based on the given inputs.

3. **Definiteness:** Every statement of the algorithm should be unambiguous.

4. **Finiteness:** No infinite loop should be allowed in an algorithm.

Example:

```
while(1<2)
{
        number=number/2;
}
```

5. **Effectiveness:** Writing an algorithm is a **priori process** of actual implementation of the algorithm. So, a person should analyze the algorithm in a finite amount of time with a pen and paper to judge the performance for giving the final version of the algorithm.

Algorithm 1: Add two numbers entered by the user

Step 1: **Start**

Step 2: Declare variables num1, num2 and sum.

Step 3: Read values num1 and num2.

Step 4: Add num1 and num2 and assign the result to sum.

sum←num1+num2

Step 5: Display sum

Step 6: **Stop**

Algorithm 2: Find the largest number among three numbers

Step 1: **Start**

Step 2: Declare variables a,b and c.

Step 3: Read variables a,b and c.

Step 4: If a > b

    If a > c

        Display a is the largest number.

    Else

        Display c is the largest number.

  Else

    If b > c

        Display b is the largest number.

    Else

        Display c is the greatest number.

Step 5: **Stop**

FLOW CHART

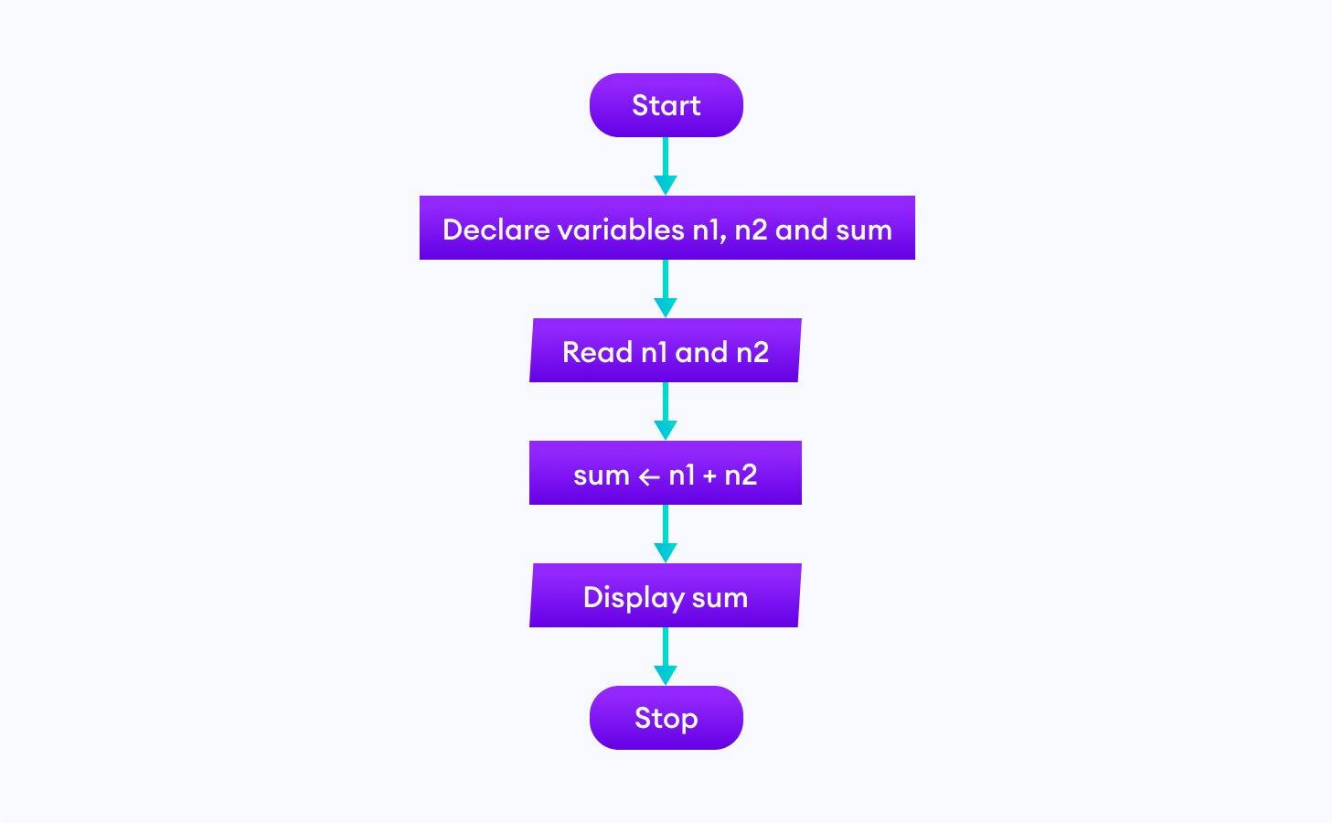
- Diagrammatic representation of an algorithm is called flow chart.
- A flowchart can be helpful for both writing programs and explaining the program to others.

Symbols Used In Flowchart

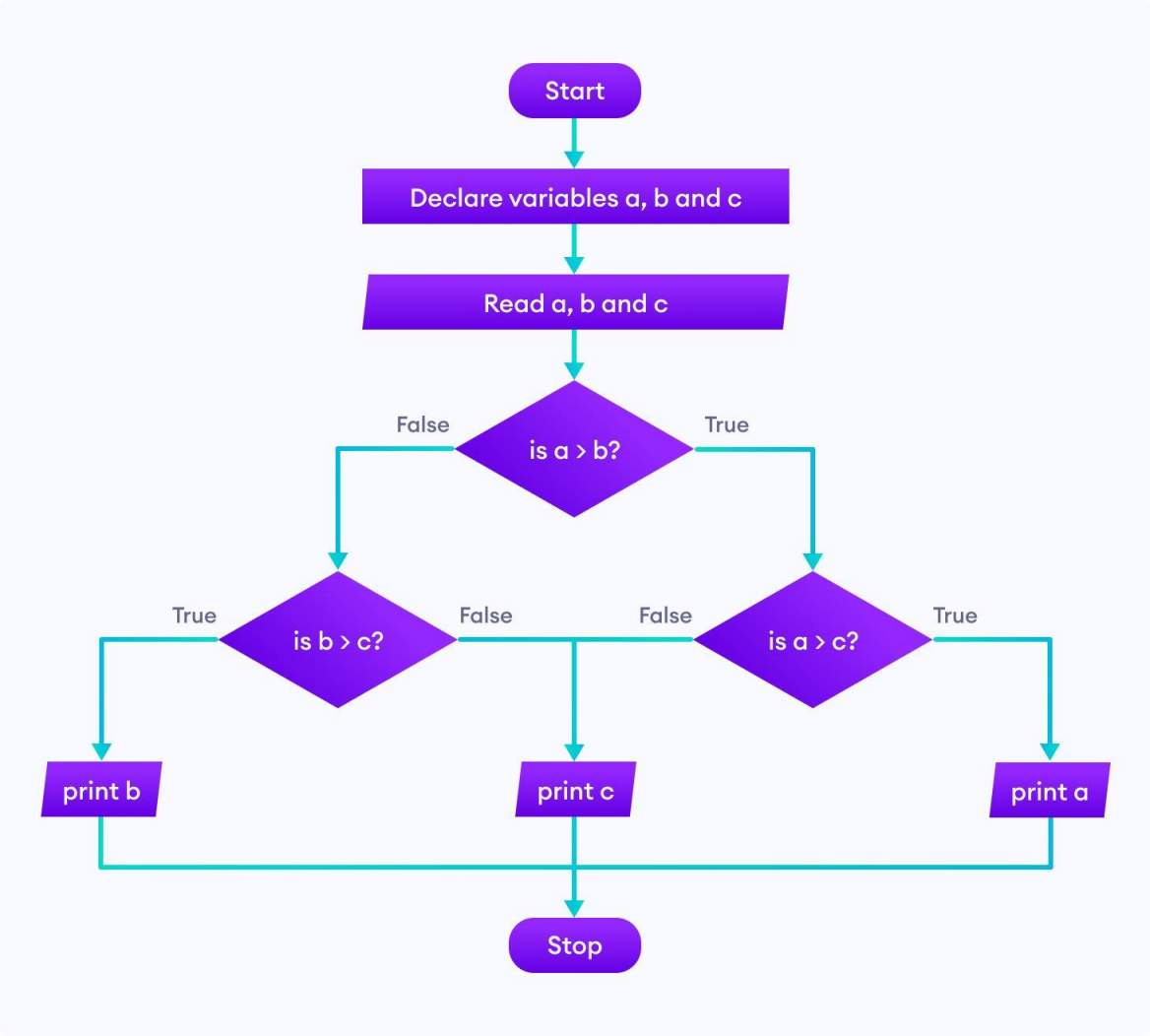
Symbol	Name	Function
	Start/end	An oval represents a start or end point
	Arrows	A line is a connector that shows relationships between the representative shapes
	Input/Output	A parallelogram represents input or output
	Process	A rectagle represents a process
	Decision	A diamond indicates a decision

Examples of flowcharts in programming

1. Add two numbers entered by the user.



2. Find the largest among three different numbers entered by the user.



Difference Between Algorithm & Flowchart

Algorithm

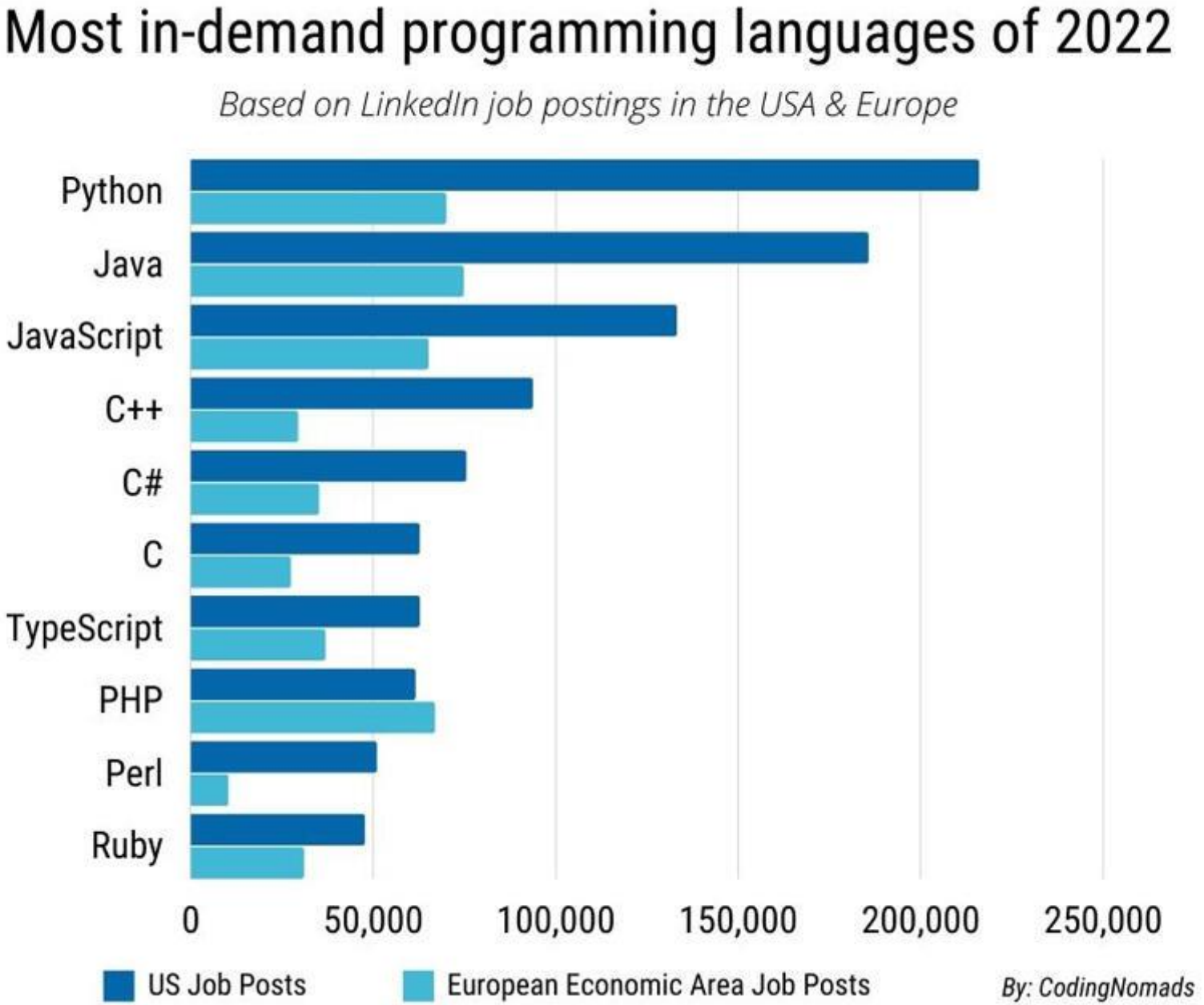
- It is defined as a sequence of well-defined steps.
- These steps provide a solution/ a way to solve a problem in hand.
- It gives the solution to a specific problem.
- This solution would be translated to machine code, which is then executed by the system to give the relevant output.
- It is difficult to understand.
- It is easy to debug.

Flowchart

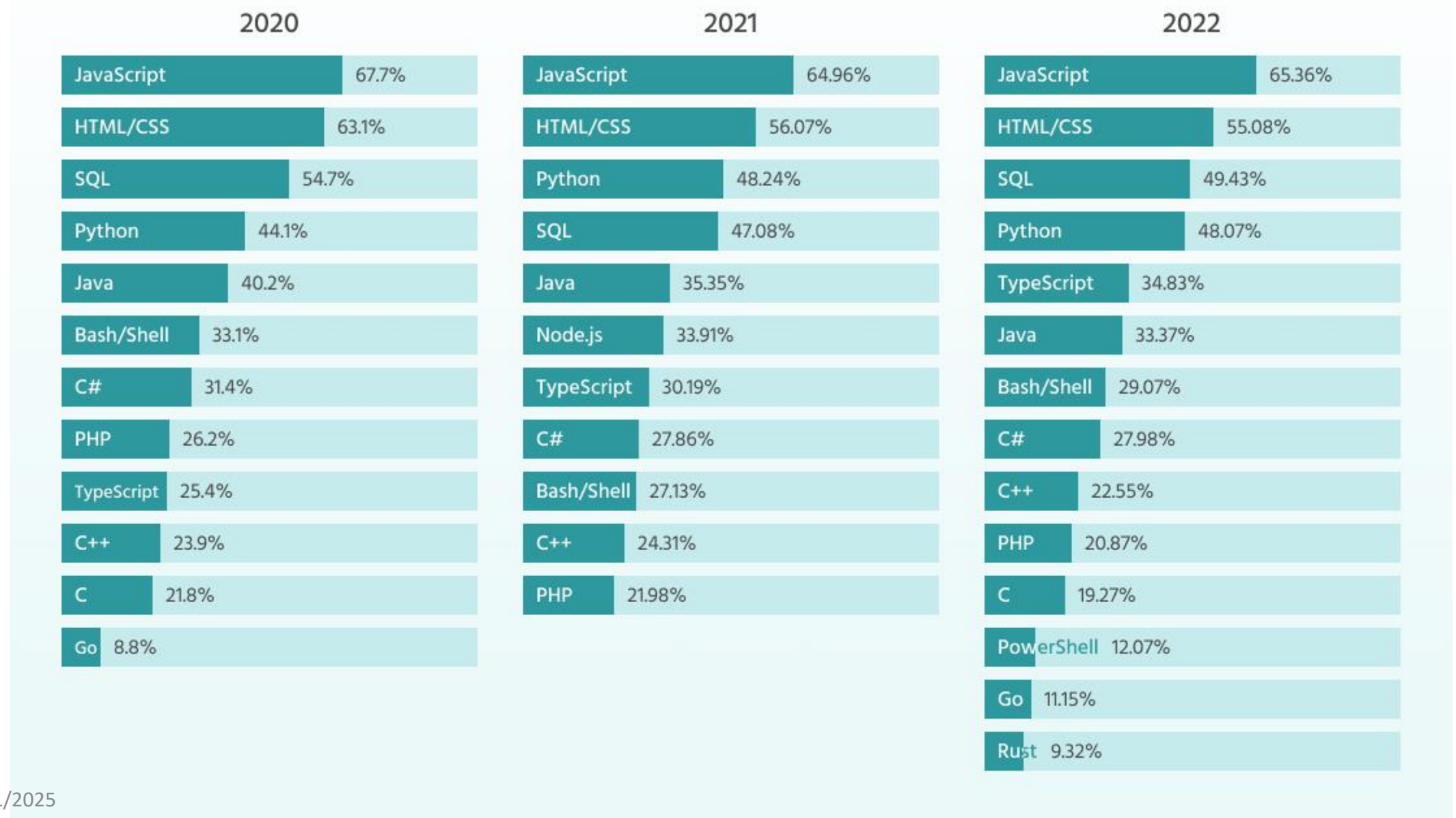
- It is a graphical representation of an algorithm.
- Programmers use it as a program-planning tool in order to solve a problem.
- This will help indicate the flow of control and information, and processing.
- The process of drawing a flowchart for an algorithm is known as "flowcharting".
- It is easy to understand.
- It is difficult to debug.

Programming Language

- A programming language is a vocabulary and set of grammatical rules for instructing a computer or computing device to perform specific tasks.
- The term *programming language* usually refers to high-level languages, such as BASIC, C, C++, COBOL, Java, FORTRAN, Ada, and Pascal.
- Each programming language has a unique set of keywords (words that it understands) and a special syntax for organizing program instructions.


Most in-demand programming languages of 2022
Based on LinkedIn job postings in the USA & Europe
Python
Java
JavaScript
C++
C#
C
TypeScript
PHP
Perl
Ruby
0
50,000
100,000
150,000
200,000
250,000
US Job Posts
European Economic Area Job Posts
By: CodingNomads

Top Programming, scripting, and markup languages



Thank you !!

C - Fundamentals

Pratik Shah
II – Sem CSE (DIV A)
02/01/25 to 21/04/25

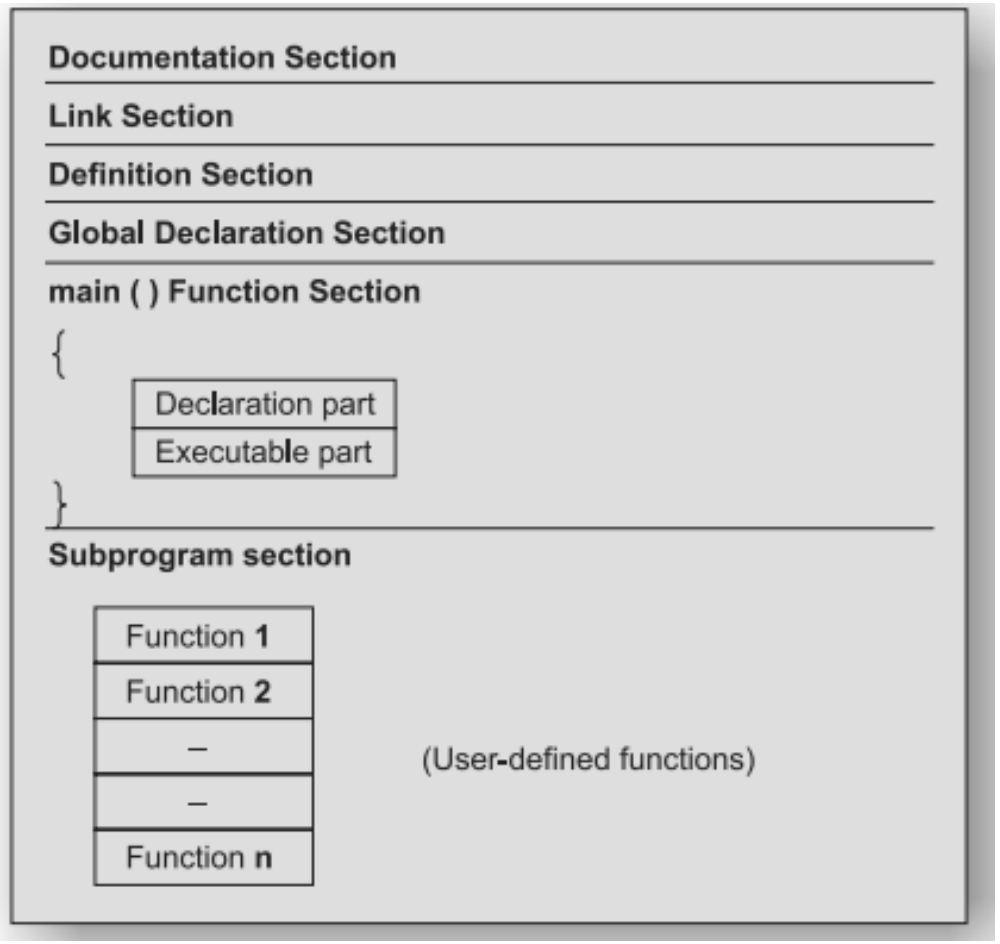
Student will be able to :

	Introduction to Programming and C fundamentals 1.1 Algorithms, Flowchart, 1.2 Programming Languages, Types of Languages
I	1.3 Basic Structure of C programming 1.4 Process of Executing C program 1.5 Character Sets, Keywords 1.6 Data types: int, char, float 1.7 Library I/O Functions 1.8 Identifiers, Constants, Declaration, Storage classes 1.9 Data input and output formatting

Importance of C

- It is a **robust language** whose rich set of built-in functions and operators can be used to write any complex program.
- Programs written in C are **efficient and fast**.
- There are only **32 keywords** in ANSI C and its strength lies in its built-in functions.
- C is highly **portable.**(This means that C programs written for one computer can be run on another with little or no modification. Portability is important if we plan to use a new computer with a different operating system)
- C language is well suited for **structured programming**
- Another important feature of C is its ability to **extend itself**. A C program is basically a collection of functions that are supported by the C library. We can continuously add our own functions to C library.
- With the availability of a large number of functions, the programming task becomes **simple.**

Basic structure of C programs



The **subprogram section** contains all the user-defined functions that are called in the main function. User-defined functions are generally placed immediately after the main function, although they may appear in any order.

The **documentation section** consists of a set of comment lines giving the name of the program, the author and other details, which the programmer would like to use later.

The **link section** provides instructions to the compiler to link functions from the system library.

The **definition section** defines all symbolic constants.

There are some variables that are used in more than one function. Such variables are called **global variables** and are declared in the global declaration section that is outside of all the functions.

Every C program must have one **main() function section**. This section contains two parts, declaration part and executable part.
The declaration part declares all the variables used in the executable part. There is at least one statement in the executable part. These two parts must appear between the opening and the closing braces. The program execution begins at the opening brace and ends at the closing brace. The closing brace of the main function section is the logical end of the program. All statements in the declaration and executable parts end with a semicolon(;).

Executing a ‘C’ program

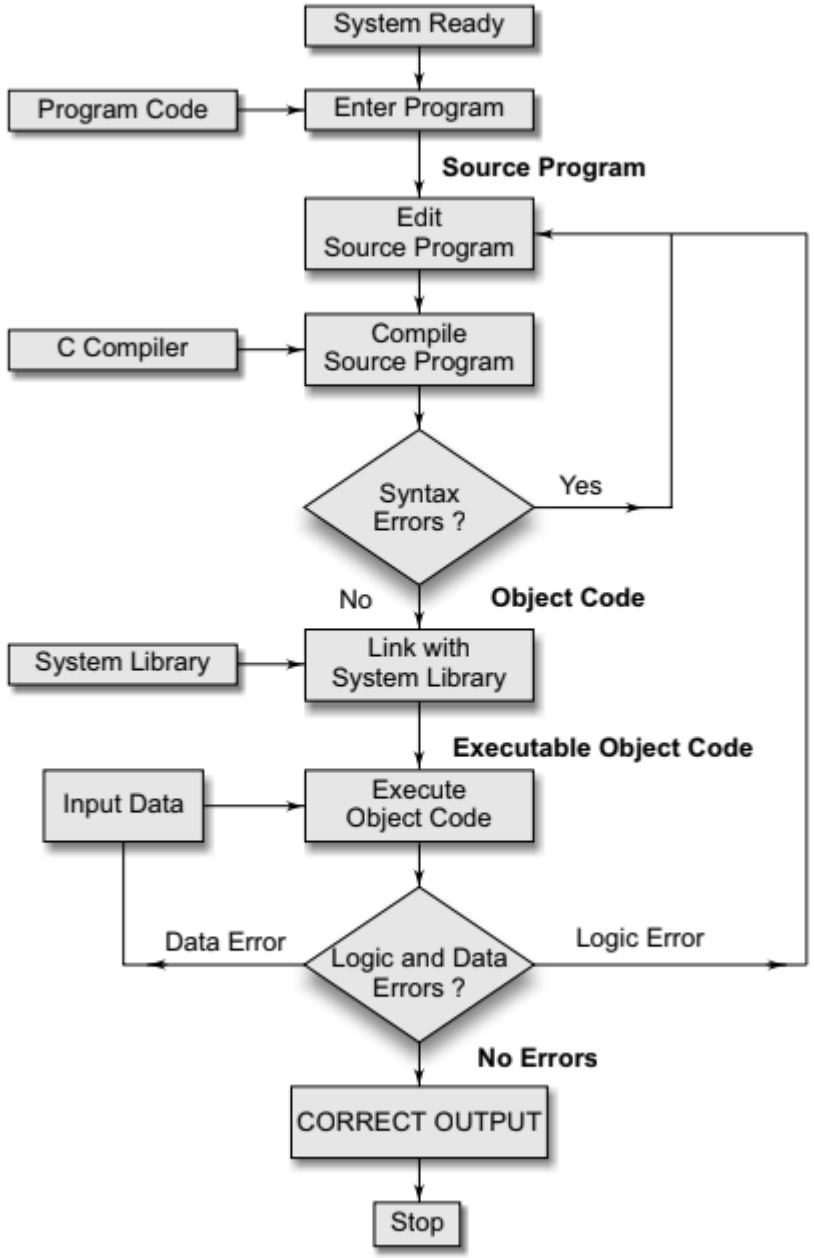
Executing a program written in C involves a series of steps.

1. Creating the program

2. Compiling the program

3. Linking the program with functions that are needed from the C library

4. Executing the program



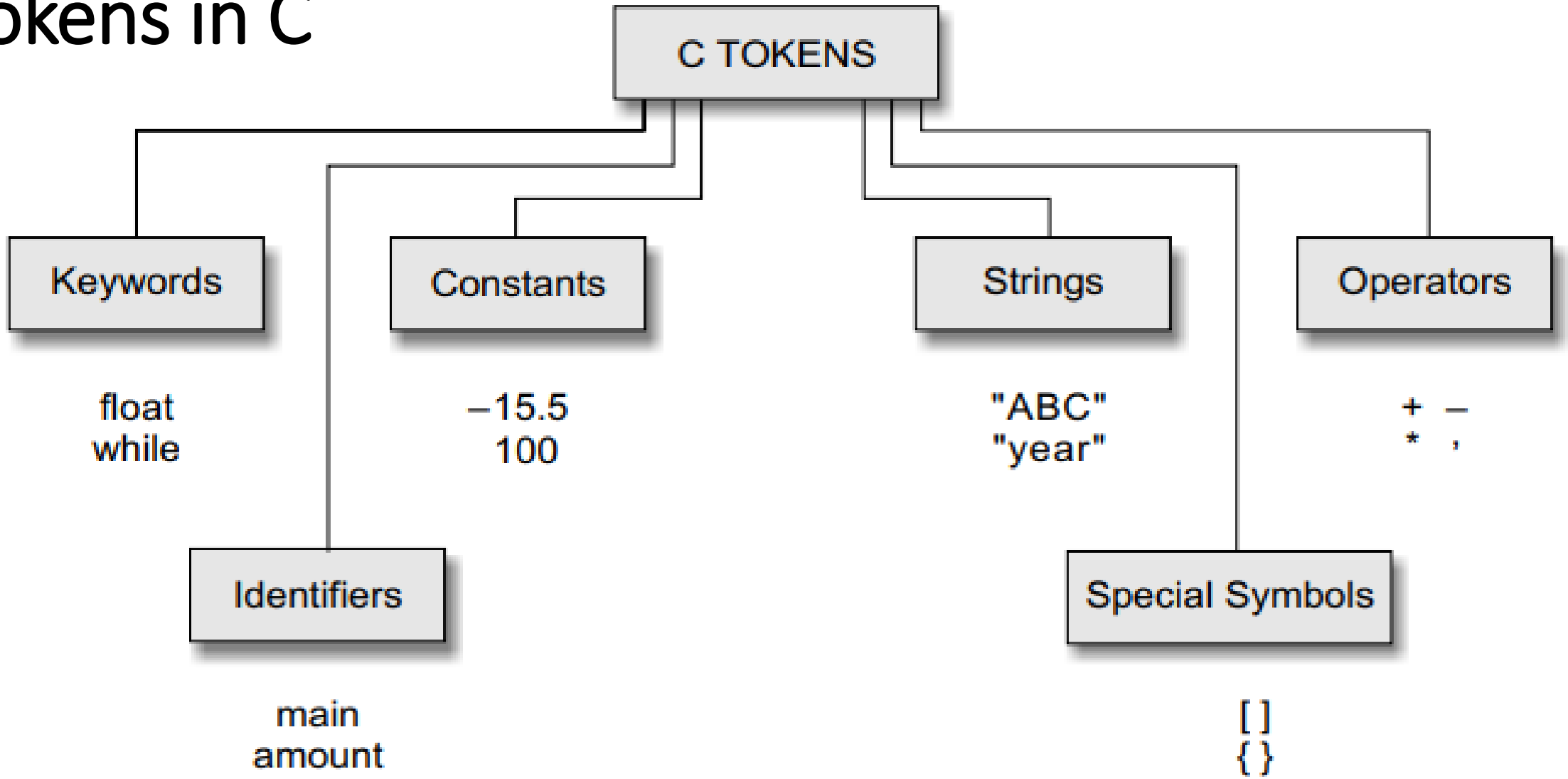
Process of compiling and runnig a C program

Character Set

- The characters that can be used to form words, numbers and expressions depend upon the computer on which the program is run.
- The characters in C are grouped into the following categories:
  - Letters
  - Digits
  - Special characters
  - White spaces

Letters		*Digits*
Uppercase A.....Z		All decimal digits 0 9
Lowercase a.....z		
	Special Characters	
, comma		& ampersand
. period		^ caret
; semicolon		* asterisk
: colon		– minus sign
? question mark		+ plus sign
‘ apostrophe		< opening angle bracket
“ quotation mark		(or less than sign)
! exclamation mark		> closing angle bracket
| vertical bar		(or greater than sign)
/ slash		(left parenthesis
\ backslash		) right parenthesis
~ tilde		[left bracket
_ under score		] right bracket
$ dollar sign		{ left brace
% percent sign		} right brace
		# number sign
	White Spaces	
	Blank space	
	Horizontal tab	
	Carriage return	
	New line	
	Form feed	

Tokens in C



C tokens and examples

- In a passage of text, individual words and punctuation marks are called **tokens.**
- A C program consists of various tokens and a token is either a keyword, an identifier, a constant, a string literal, or a symbol. For example, the following C statement consists of five tokens −
  - printf("Hello, World! \n");

The individual tokens are −

printf

(

"Hello, World! \n"

)

;

Semicolons

- In a C program, the semicolon is a statement terminator. That is, each individual statement must be ended with a semicolon. It indicates the end of one logical entity.
- Given below are two different statements −
  - printf("Hello, World! \n");
  - c=a+b;

Comments

- Comments are like helping text in your C program and they are ignored by the compiler. They start with /* and terminate with the characters */ as shown below –

```
/* my first program in C */
```

- Single line comment : they start with //
  - //my first program

Keywords

- The following list shows the reserved words in C. These reserved words may not be used as constants or variables or any other identifier names.

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

Identifiers

- A C identifier is a name used to identify a variable, function, or any other user-defined item. An identifier starts with a letter A to Z, a to z, or an underscore '_' followed by zero or more letters, underscores, and digits (0 to 9).
- C does not allow punctuation characters such as @, $, and % within identifiers. C is a **case-sensitive** programming language.
  Thus, *Manpower* and *manpower* are two different identifiers in C.
  Here are some examples of acceptable identifiers –

  mohd zara abc move_name a_123

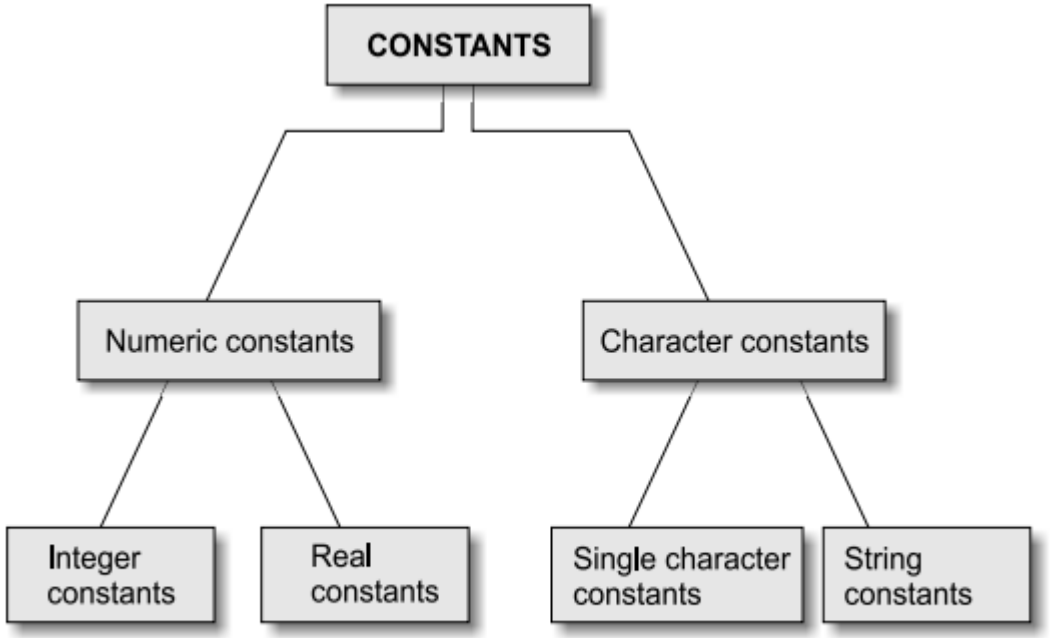
  myname50 _temp j a23b9 retVal

Rules for Identifiers

- First character must be an alphabet (or underscore).
- Must consist of only letters, digits or underscore.
- Only first 31 characters are significant.
- Cannot use a keyword.
- Must not contain white space

Constants

- Constants in C refer to fixed values that do not change during the execution of a program.
- C supports several types of constants as illustrated in diagram



Basic types of C constants

Backslash character constants

- C supports some special backslash character constants that are used in output functions.

Backslash Character Constants

Constant	*Meaning*
'\a'	audible alert (bell)
'\b'	back space
'\f'	form feed
'\n'	new line
'\r'	carriage return
'\t'	horizontal tab
'\v'	vertical tab
'\''	single quote
'\"'	double quote
'\?'	question mark
'\\'	backslash
'10'	null

C - Variables

- A variable is a data name that may be used to store a data value.
- A variable name can be chosen by the programmer in a meaningful way so as to reflect its function or nature in the program.
- Some examples of such names are:
  - Average
  - height
  - Total
  - Counter_1
  - class_strength

- variable names may consist of **letters, digits, and the underscore(_) character**, subject to the following conditions:
  - They must **begin** with a **letter**.
  - Some systems permit underscore as the first character.
  - ANSI standard recognizes a length of 31 characters. However, **length** should **not** be normally more than **eight characters**, since only the first eight characters are treated as significant by many compilers. (In C99, at least 63 characters are significant.)
  - Uppercase and lowercase are **significant.** That is, the variable Total is not the same as total or TOTAL.
  - It should **not** be a **keyword**.
  - **White space** is **not** allowed.

Some valid declarations are shown here

- int i, j, k;
- char c, ch;
- float f, salary;
- double d;

- The line **int i, j, k;** declares and defines the variables i, j, and k; which instruct the compiler to create variables named i, j and k of type int.

- Variables can be initialized (assigned an initial value) in their declaration. The initializer consists of an equal sign followed by a constant expression as follows −
  - type variable_name = value;
  - Some examples are −
  - int d = 3, f = 5; // definition and initializing d and f.
  - char x = 'x'; // the variable x has the value 'x'.

Whitespace in C

- A line containing only whitespace, possibly with a comment, is known as a blank line, and a C compiler totally ignores it.
- Whitespace is the term used in C to describe blanks, tabs, newline characters and comments.
- Whitespace separates one part of a statement from another and enables the compiler to identify where one element in a statement, such as int, ends and the next element begins.

- Therefore, in the following statement −
  - **int age;**

  there must be at least one whitespace character (usually a space) between int and age for the compiler to be able to distinguish them.

  On the other hand, in the following statement −
  - **fruit = apples + oranges;** // get the total fruit

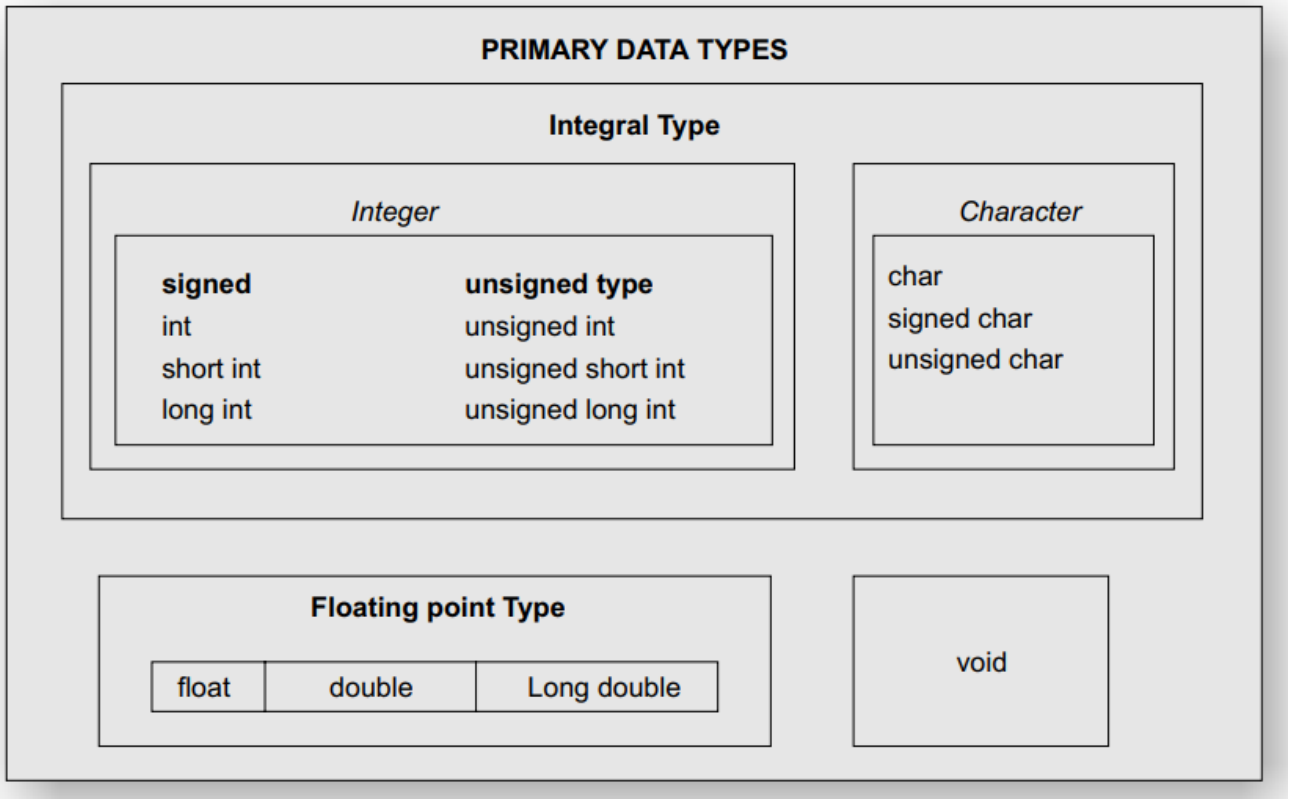
  no whitespace characters are necessary between fruit and =, or between = and apples, although you are free to include some if you wish to increase readability.

C - Data Types

- Data types in c refer to an extensive system used for declaring variables or functions of different types.
- The type of a variable determines how much space it occupies in storage and how the bit pattern stored is interpreted.
- ANSI C supports **three** classes of **data types**:
  - **Primary (or fundamental) data types**
  - **Derived data types**
  - **User-defined data types**

Primary Datatypes

- All C compilers support **five** fundamental data types, namely integer (**int**), character (**char**), floating point (**float**), double-precision floating point (**double**) and **void**.



Primary data types in C

Size and Range of Data Types on a 16-bit Machine

Type	*Size (bits)*	*Range*
char or signed char	8	–128 to 127
unsigned char	8	0 to 255
int or signed int	16	–32,768 to 32,767
unsigned int	16	0 to 65535
short int or signed short int	8	–128 to 127
unsigned short int	8	0 to 255
long int or signed long int	32	–2,147,483,648 to 2,147,483,647
unsigned long int	32	0 to 4,294,967,295
float	32	3.4E – 38 to 3.4E + 38
double	64	1.7E – 308 to 1.7E + 308
long double	80	3.4E – 4932 to 1.1E + 4932

Derived Datatypes

- **Derived Datatypes** are composed of fundamental datatypes;
- They are derived from the fundamental data types.
- Therefore, they have some additional characteristics and properties other than that of fundamental data types.
- Programmers can modify or redefine the derived datatypes.
- Some common examples of derived datatypes include **Arrays, Functions, Pointers,** etc.

User-defined datatypes

- User-defined data types are created by the user using a combination of fundamental and derived data types in the C programming language.
- Some common examples of User-defined datatypes include **Structure, Union, Typedef, enum**

Declaration of Storage Class

- Variables in C can have not only data type but also storage class that provides information about their location and visibility.
- The storage class decides the portion of the program within which the variables are recognized.

- Consider the following example:

```
/* Example of storage classes */
int   m;
main()
{
   int i;
   float balance;
   ....
   ....
  function1();
}
function1()
{
   int i;
   float sum;
   ....
   ....
   }
```

The **variable m** which has been declared before the main is called **global variable**

It can be used in all the functions in the program.

It need not be declared in other functions.

A global variable is also known as an external variable.

The variables **i, balance and sum** are called **local variables** because they are declared inside a function.

Local variables are visible and meaningful only inside the functions in which they are declared. They are not known to other functions.

Note that the **variable i** has been declared in both the functions. Any change in the value of i in one function does not affect its value in the other.

- C provides a variety of storage class specifiers that can be used to declare explicitly the scope and lifetime of variables.
- The concepts of scope and lifetime are important only in multifunction and multiple file programs and therefore the storage classes are considered in detail later when functions are discussed.
- For now, remember that there are four storage class specifiers

  (**auto, register, static, and extern**)

Storage Classes and Their Meaning

Storage class	*Meaning*
auto	Local variable known only to the function in which it is declared. *Default is auto.*
static	Local variable which exists and retains its value even after the control is transferred to the calling function.
extern	Global variable known to all functions in the file.
register	Local variable which is stored in the register.

Automatic

- Automatic variables are allocated memory automatically at runtime.
- The visibility of the automatic variables is limited to the block in which they are defined.
- The scope of the automatic variables is limited to the block in which they are defined. The automatic variables are initialized to garbage by default.
- The memory assigned to automatic variables gets freed upon exiting from the block.
- The keyword used for defining automatic variables is auto.
- Every local variable is automatic in C by default.

```c
#include <stdio.h>
int main()
{
int a; //auto
char b;
float c;
printf("%d %c %f",a,b,c); // printing initial default value of automatic variables a, b, and c.
return 0;
}
```

Output:

```
garbage garbage garbage
```

Static

- The variables defined as static specifier can hold their value between the multiple function calls.
- Static local variables are visible only to the function or the block in which they are defined.
- A same static variable can be declared many times but can be assigned at only one time.
- Default initial value of the static integral variable is 0 otherwise null.
- The visibility of the static global variable is limited to the file in which it has declared.
- The keyword used to define static variable is static.

Example 1

```
#include<stdio.h>
static char c;
static int i;
static float f;
static char s[100];
void main ()
{
printf("%d %d %f %s",c,i,f); // the initial default value of c, i, and f will be printed.
}
```

Output:

0 0 0.000000 (null)

Register

- The variables defined as the register is allocated the memory into the CPU registers depending upon the size of the memory remaining in the CPU.
- We can not dereference the register variables, i.e., we can not use &operator for the register variable.
- The access time of the register variables is faster than the automatic variables.
- The initial default value of the register local variables is 0.
- The register keyword is used for the variable which should be stored in the CPU register. However, it is compilers choice whether or not; the variables can be stored in the register.
- We can store pointers into the register, i.e., a register can store the address of a variable.
- Static variables can not be stored into the register since we can not use more than one storage specifier for the same variable.

Example 1

```
#include <stdio.h>
int main()
{
register int a; // variable a is allocated memory in the CPU register. The initial default value of a is 0.
printf("%d",a);
}
```

Output:

```
0
```

External

- The external storage class is used to tell the compiler that the variable defined as extern is declared with an external linkage elsewhere in the program.
- The variables declared as extern are not allocated any memory. It is only declaration and intended to specify that the variable is declared elsewhere in the program.
- The default initial value of external integral type is 0 otherwise null.
- We can only initialize the extern variable globally, i.e., we can not initialize the external variable within any block or method.
- An external variable can be declared many times but can be initialized at only once.
- If a variable is declared as external then the compiler searches for that variable to be initialized somewhere in the program which may be extern or static. If it is not, then the compiler will show an error.

Example 1

```c
#include <stdio.h>
int main()
{
extern int a;
printf("%d",a);
}
```

Output

```
main.c:(.text+0x6): undefined reference to `a'
collect2: error: ld returned 1 exit status
```

C library functions

- Library functions are built-in functions that are grouped together and placed in a common location called library.
- Each function here performs a specific operation. We can use this library functions to get the pre-defined output.
- All C standard library functions are declared by using many header files.
- These library functions are created at the time of designing the compilers.
- We include the header files in our C program by using **#include<filename.h>**. Whenever the program is run and executed, the related files are included in the C program.

Header File Functions

- Some of the header file functions are as follows –
- **stdio.h** – It is a standard i/o header file in which Input/output functions are declared
- **conio.h** – This is a console input/output header file.
- **string.h** – All string related functions are in this header file.
- **stdlib.h** – This file contains common functions which are used in the C programs.
- **math.h** – All functions related to mathematics are in this header file.
- **time.h** – This file contains time and clock related functions.Built functions in stdio.h

Built functions in stdio.h

1	**printf()** This function is used to print the all char, int, float, string etc., values onto the output screen.
2	**scanf()** This function is used to read data from keyboard.
3	**getc()** It reads character from file.
4	**gets()** It reads line from keyboard.
5	**getchar()** It reads character from keyboard.
6	**puts()** It writes line to o/p screen.
7	**putchar()** It writes a character to screen.
8	**fopen()** All file handling functions are defined in stdio.h header file.
9	**fclose()** Closes an opened file.
10	**getw()** Reads an integer from file.
11	**putw()** Writes an integer to file.
12	**fgetc()** Reads a character from file.
13	**putc()** Writes a character to file.
14	**fputc()** Writes a character to file.
15	**fgets()** Reads string from a file, one line at a time.
16	**f puts()** Writes string to a file.
17	**feof()** Finds end of file.

18	**fgetchar** Reads a character from keyboard.
19	**fgetc()** Reads a character from file.
20	**fprintf()** Writes formatted data to a file.
21	**fscanf()** Reads formatted data from a file.
22	**fputchar** Writes a character from keyboard.
23	**fseek()** Moves file pointer to given location.
24	**SEEK_SET** Moves file pointer at the beginning of the file.
25	**SEEK_CUR** Moves file pointer at given location.
26	**SEEK_END** Moves file pointer at the end of file.
27	**ftell()** Gives current position of file pointer.

28	**rewind()** Moves file pointer to the beginning of the file.
29	**putc()** Writes a character to file.
30	**sprint()** Writes formatted output to string.
31	**sscanf()** Reads formatted input from a string.
32	**remove()** Deletes a file.
33	**flush()** Flushes a file.

https://www.tutorialspoint.com/what-are-the-c-library-functions

Data Output Formatting

- To display any message or value on output screen, then we use **printf()**
- printf and PRINTF are not the same.
- In C, everything is written in lowercase letters.
- **Syntax :** **printf(“Your message here”);**
- **Example :** **printf(“Hello World”);**

- **To print the value of variable :** **printf(“Control string”, name of variable);**
- **Example :** **printf(“%d”,a);**

Data Input Formatting

- Another way of giving values to variables is to input data through keyboard using the **scanf function.**
- The general format of scanf is as follows:

```
scanf("control string", &variable1,&variable2,....);
```

- The control string contains the format of data being received.
- The ampersand symbol & before each variable name is an operator that specifies the variable name's address.
- We must always use this operator, otherwise unexpected results may occur.

Example

scanf("%d", &number);

- When this statement is encountered by the computer, the execution stops and waits for the value of the variable number to be typed in. Since the control string "%d" specifies that an integer value is to be read from the terminal, we have to type in the value in integer form.
- Once the number is typed in and the 'Enter' Key is pressed, the computer then proceeds to the next statement.
- Thus, the use of scanf provides an interactive feature and makes the program 'user friendly'.
- The value is assigned to the variable number.

References

- Programming in ANSI C – Balaguruswamy
- https://www.tutorialspoint.com/what-are-the-c-library-functions

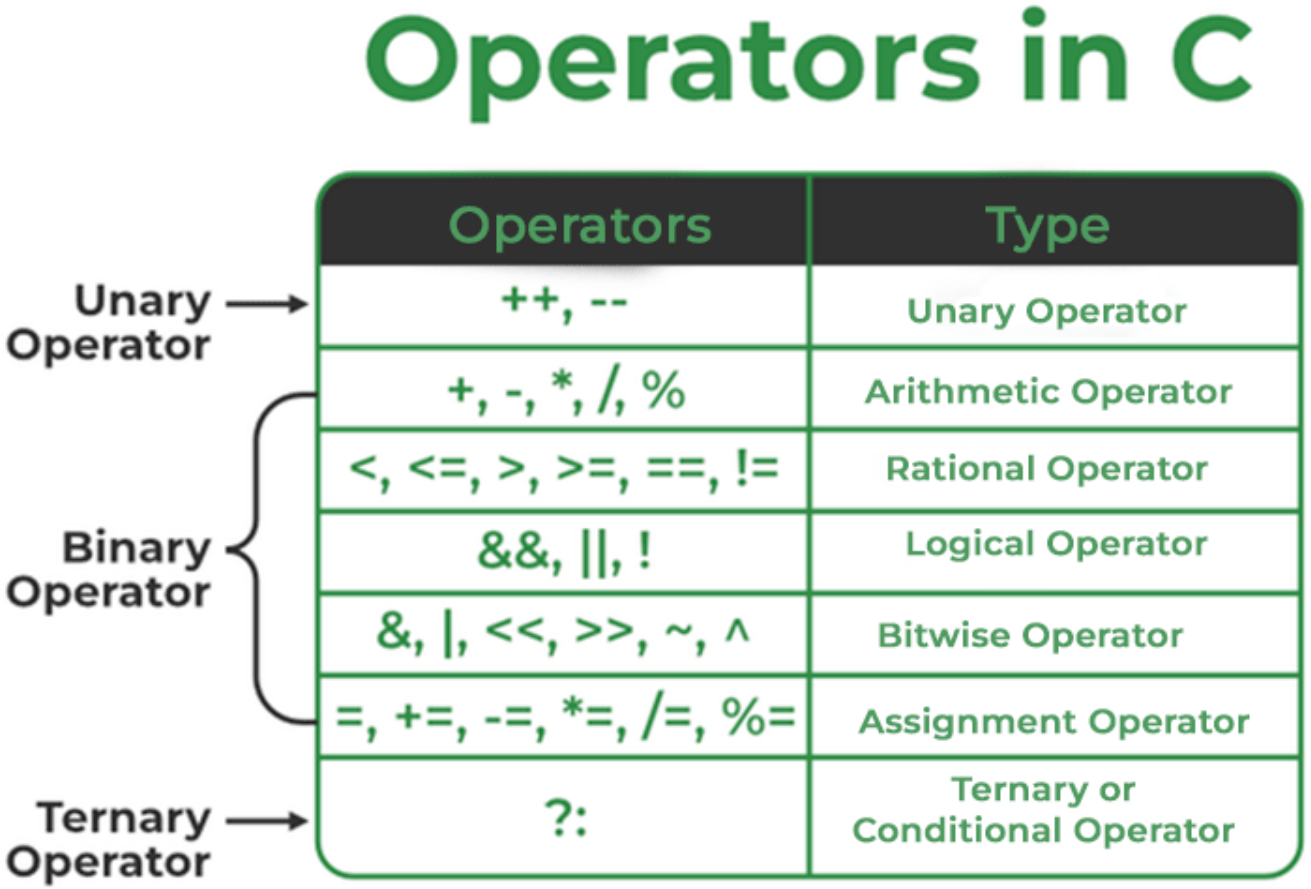
Thank You !!!!

C - Operators

Pratik Shah
II – Sem CSE (DIV A)
02/01/25 to 21/04/25

What is a C Operator?

An operator in C can be defined as the symbol that helps us to perform some specific **mathematical, relational, bitwise, conditional, or logical computations on values and variables. The values and variables used with operators are called operands.** So we can say that the operators are the symbols that perform operations on operands.



Types of Operators in C

C language provides a wide range of operators that can be classified into **6 types** based on their functionality:

1. **Arithmetic Operators**
2. **Relational Operators**
3. **Logical Operators**
4. **Bitwise Operators**
5. **Assignment Operators**
6. **Other Operators**

1. Arithmetic Operations in C

The arithmetic operators are used to perform **arithmetic/mathematical operations** on operands.
There are 9 arithmetic operators in C language

S. No.	Symbol	Operator	Description	Syntax
1	+	Plus	Adds two numeric values.	a + b
2	–	Minus	Subtracts right operand from left operand.	a – b
3	*	Multiply	Multiply two numeric values.	a * b
4	/	Divide	Divide two numeric values.	a / b
5	%	Modulus	Returns the remainder after diving the left operand with the right operand.	a % b
6	+	Unary Plus	Used to specify the positive values.	+a
7	–	Unary Minus	Flips the sign of the value.	-a
8	++	Increment	Increases the value of the operand by 1.	a++
9	—	Decrement	Decreases the value of the operand by 1.	a–

Example of C Arithmetic Operators

```
// C program to illustrate the arithmetic operators
#include <stdio.h>
void main()
{

    int a = 25, b = 5;
    printf("a + b = %d\n", a + b);
    printf("a - b = %d\n", a - b);
    printf("a * b = %d\n", a * b);
    printf("a / b = %d\n", a / b);
    printf("a % b = %d\n", a % b);
    printf("+a = %d\n", +a);
    printf("-a = %d\n", -a);
    printf("a++ = %d\n", a++);
    printf("a-- = %d\n", a--);
}
```

2. Relational Operators in C

The relational operators in C are used for the comparison of the two operands. All these operators are binary operators that **return true or false** values as the result of comparison.
These are a total of **6 relational operators** in C

S. No.	Symbol	Operator	Description	Syntax
1	**<**	**Less than**	Returns true if the left operand is less than the right operand. Else false	**a < b**
2	**>**	**Greater than**	Returns true if the left operand is greater than the right operand. Else false	**a > b**
3	**<=**	**Less than or equal to**	Returns true if the left operand is less than or equal to the right operand. Else false	**a <= b**
4	**>=**	**Greater than or equal to**	Returns true if the left operand is greater than or equal to right operand. Else false	**a >= b**
5	**==**	**Equal to**	Returns true if both the operands are equal.	**a == b**
6	**!=**	**Not equal to**	Returns true if both the operands are NOT equal.	**a != b**

Example of C Relational Operators

```
// C program to illustrate the relational operators
#include <stdio.h>
void main()
{

    int a = 25, b = 5;
    printf("a < b  : %d\n", a < b);
    printf("a > b  : %d\n", a > b);
    printf("a <= b: %d\n", a <= b);
    printf("a >= b: %d\n", a >= b);
    printf("a == b: %d\n", a == b);
    printf("a != b : %d\n", a != b);


}
```

3. Logical Operator in C

Logical Operators are used to **combine two or more conditions/constraints or to complement the evaluation of the original condition in consideration.** The result of the operation of a logical operator is a Boolean value either **true** or **false**.

S. No.	Symbol	Operator	Description	Syntax
1	**&&**	**Logical AND**	Returns true if both the operands are true.	**a && b**
2	**||**	**Logical OR**	Returns true if both or any of the operand is true.	**a || b**
3	**!**	**Logical NOT**	Returns true if the operand is false.	**!a**

Example of C Logical Operators

```
// C program to illustrate the logical operators
#include <stdio.h>
void main()
{

    int a = 25, b = 5;
    printf("a && b : %d\n", a>20 && b>2);
    printf("a || b : %d\n", a>20 || b>10);
    printf("!a: %d\n", !(a>15));
}
```

4. Bitwise Operators in C

The Bitwise operators are used to perform **bit-level operations on the operands**. **The operators are first converted to bit-level and then the calculation is performed on the operands. Mathematical operations such as addition, subtraction, multiplication, etc. can be performed at the bit level for faster processing.**

There are **6 bitwise operators** in C:

S. No.	Symbol	Operator	Description	Syntax
1	&	Bitwise AND	Performs bit-by-bit AND operation and returns the result.	a & b
2	|	Bitwise OR	Performs bit-by-bit OR operation and returns the result.	a | b
3	^	Bitwise XOR	Performs bit-by-bit XOR operation and returns the result.	a ^ b
4	~	Bitwise First Complement	Flips all the set and unset bits on the number.	~a
5	<<	Bitwise Leftshift	Shifts the number in binary form by one place in the operation and returns the result.	a << b
6	>>	Bitwise Rightshilft	Shifts the number in binary form by one place in the operation and returns the result.	a >> b

Example of C Bitwise Operators

```
// C program to illustrate the Bitwise operators
#include <stdio.h>
void main()
{

    int a = 25, b = 5;
    printf("a & b: %d\n", a & b);
    printf("a | b: %d\n", a | b);
    printf("a ^ b: %d\n", a ^ b);
    printf("~a: %d\n", ~a);
    printf("a >> b: %d\n", a >> b);
    printf("a << b: %d\n", a << b);
}
```

5. Assignment Operators in C

Assignment operators are used to assign value to a variable.
In C, there are **11 assignment operators** :

S. No.	Symbol	Operator	Description	Syntax
1	=	Simple Assignment	Assign the value of the right operand to the left operand.	a = b
2	+=	Plus and assign	Add the right operand and left operand and assign this value to the left operand.	a += b
3	-=	Minus and assign	Subtract the right operand and left operand and assign this value to the left operand.	a -= b
4	*=	Multiply and assign	Multiply the right operand and left operand and assign this value to the left operand.	a *= b
5	/=	Divide and assign	Divide the left operand with the right operand and assign this value to the left operand.	a /= b
6	%=	Modulus and assign	Assign the remainder in the division of left operand with the right operand to the left operand.	a %= b
7	&=	AND and assign	Performs bitwise AND and assigns this value to the left operand.	a &= b
8	|=	OR and assign	Performs bitwise OR and assigns this value to the left operand.	a |= b
9	^=	XOR and assign	Performs bitwise XOR and assigns this value to the left operand.	a ^= b
10	>>=	Rightshift and assign	Performs bitwise Rightshift and assign this value to the left operand.	a >>= b
11	<<=	Leftshift and assign	Performs bitwise Leftshift and assign this value to the left operand.	a <<= b

Example of C Assignment Operators

```
// C program to illustrate the Assignment operators
#include <stdio.h>
void main()
{
    int a = 25, b = 5;
    printf("a = b: %d\n", a = b);
    printf("a += b: %d\n", a += b);
    printf("a -= b: %d\n", a -= b);
    printf("a *= b: %d\n", a *= b);
    printf("a /= b: %d\n", a /= b);
    printf("a %%= b: %d\n", a %= b);
    printf("a &= b: %d\n", a &= b);
    printf("a |= b: %d\n", a |= b);
    printf("a >>= b: %d\n", a >>= b);
    printf("a <<= b: %d\n", a <<= b);
}
```

6. Other Operators

Apart from the above operators, there are some other operators available in C used to perform some specific tasks.

sizeof Operator

It is a compile-time unary operator which can be used to compute the size of its operand.
The result of sizeof is of the unsigned integral type which is usually denoted by size_t.
Basically, the sizeof the operator is used to compute the size of the variable or datatype.

Syntax

sizeof (operand)

Cast Operator

- Casting operators convert one data type to another. **For example, int(2.2000) would return 2.**
- A cast is a special operator that forces one data type to be converted into another.
- **[(type) expression]**.

Syntax

(new_type) operand;

Conditional Operator (? :)

The conditional operator is the only ternary operator in C++.

Here, Expression1 is the condition to be evaluated. If the condition(Expression1) is True then we will execute and return the result of Expression2 otherwise if the condition(Expression1) is false then we will execute and return the result of Expression3.

We may replace the use of if..else statements with conditional operators.

Syntax

operand1 ? operand2 : operand3;

addressof (&) and Dereference (*) Operators

Pointer operator & returns the address of a variable. For example &a; will give the actual address of the variable.

The **pointer operator * is a pointer to a variable.**

For example ***var; will pointer to a variable var.**

Example of Other C Operators

```
// C Program to demonstrate the use of Misc operators
#include <stdio.h>
void main()
{
    int num = 10;
    int* add_of_num = &num;

   printf("sizeof(num) = %d bytes\n", sizeof(num));
    printf("&num = %p\n", &num);
    printf("*add_of_num = %d\n", *add_of_num);
    printf("(10 < 5) ? 10 : 20 = %d\n", (10 < 5) ? 10 : 20);
    printf("(float)num = %f\n", (float)num);
}
```

Operator Precedence and Associativity in C

In C, it is very common for an expression or statement to have multiple operators and in these expression, there should be a fixed order or priority of operator evaluation to avoid ambiguity.

Operator Precedence and Associativity *is the concept that decides which operator will be evaluated first in the case when there are multiple operators present in an expression.*

Precedence	Operator	Description	Associativity
1	()	Parentheses (function call)	left-to-right
	[]	Brackets (array subscript)	left-to-right
	.	Member selection via object name	left-to-right
	->	Member selection via a pointer	left-to-right
	a++ , a–	Postfix increment/decrement (a is a variable)	left-to-right
2	++a , –a	Prefix increment/decrement (a is a variable)	right-to-left
	+ , –	Unary plus/minus	right-to-left
	! , ~	Logical negation/bitwise complement	right-to-left
	(type)	Cast (convert value to temporary value of type)	right-to-left
	*	Dereference	right-to-left
	&	Address (of operand)	right-to-left
	sizeof	Determine size in bytes on this implementation	right-to-left
3	* , / , %	Multiplication/division/modulus	left-to-right
4	+ , –	Addition/subtraction	left-to-right
5	<< , >>	Bitwise shift left, Bitwise shift right	left-to-right
6	< , <=	Relational less than/less than or equal to	left-to-right
	> , >=	Relational greater than/greater than or equal to	left-to-right
7	== , !=	Relational is equal to/is not equal to	left-to-right
8	&	Bitwise AND	left-to-right

9	**^**	Bitwise XOR	left-to-right
10	**|**	Bitwise OR	left-to-right
11	**&&**	Logical AND	left-to-right
12	**||**	Logical OR	left-to-right
13	**?:**	Ternary conditional	right-to-left
14	**=**	Assignment	right-to-left
	+= , -=	Addition/subtraction assignment	right-to-left
	***= , /=**	Multiplication/division assignment	right-to-left
	%= , &=	Modulus/bitwise AND assignment	right-to-left
	^= , |=	Bitwise exclusive/inclusive OR assignment	right-to-left
	<<=, >>=	Bitwise shift left/right assignment	right-to-left
15	**,**	expression separator	left-to-right

References

- https://www.geeksforgeeks.org/operators-in-c/

Thank You !!!!

Shri Vile Parle Kelavani Mandal's

SHRI BHAGUBHAI MAFATLAL POLYTECHNIC

Computer Engineering Department

Control Structure Part - I

Course: Programming in C

Course Code: PRC238912

Name of Staff: Mr. Pratik H. Shah

SEMESTER : II

DIVISION : A

Reference: https://www.geeksforgeeks.org/decision-making-c-cpp/

Control Structures in Programming Languages

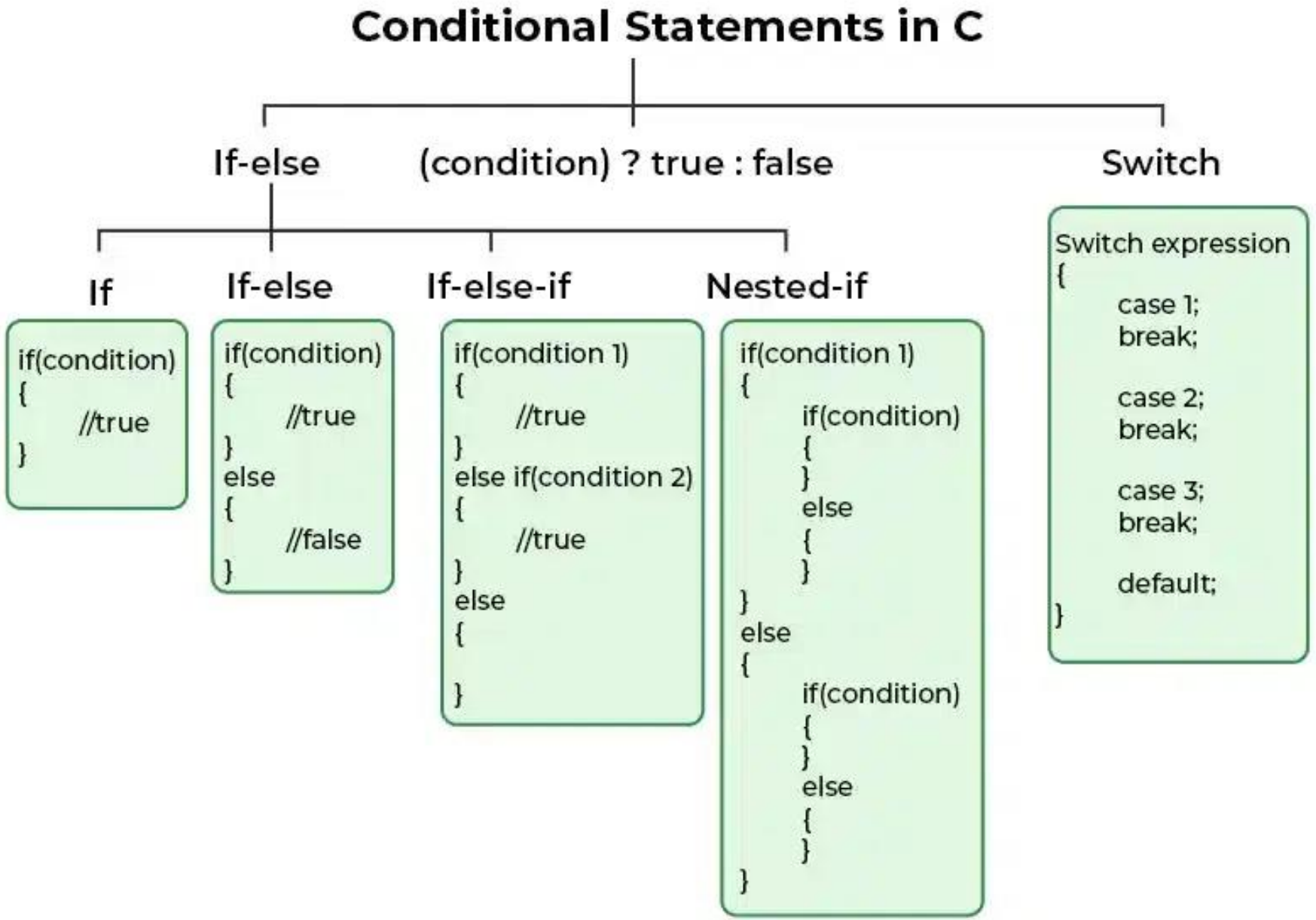
Control Structures are just a way to specify flow of control in programs. Any algorithm or program can be more clear and understood if they use self-contained modules called as logic or control structures. It basically analyzes and chooses in which direction a program flows based on certain parameters or conditions. There are three basic types of logic, or flow of control, known as:

1. Sequence logic, or sequential flow
2. Selection logic, or conditional flow
3. Iteration logic, or repetitive flow

Sequential Logic (Sequential Flow) Sequential logic as the name suggests follows a serial or sequential flow in which the flow depends on the series of instructions given to the computer. Unless new instructions are given, the modules are executed in the obvious sequence.

Selection Logic (Conditional Flow) Selection Logic simply involves a number of conditions or parameters which decides one out of several written modules. The structures which use these type of logic are known as **Conditional Structures**.

Iteration Logic (Repetitive Flow)
The Iteration logic employs a loop which involves a repeat statement followed by a module known as the body of a loop.


Conditional Statements in C
If-else
(condition) ? true : false
Switch
If
If-else
If-else-if
Nested-if
if(condition)
{
//true
}
if(condition)
{
//true
}
else
{
//false
}
if(condition 1)
{
//true
}
else if(condition 2)
{
//true
}
else
{
}
if(condition 1)
{
if(condition)
{
}
else
{
}
}
else
{
if(condition)
{
}
else
{
}
}
Switch expression
{
case 1;
break;
case 2;
break;
case 3;
break;
default;
}

1. if in C

The if statement is the most simple decision-making statement. It is used to decide whether a certain statement or block of statements will be executed or not i.e if a certain condition is true then a block of statements is executed otherwise not.

Syntax of if Statement

```
if(condition)
{
  // Statements to execute if
  // condition is true
}
```

Here, the **condition** after evaluation will be either true or false. C if statement accepts boolean values – if the value is true then it will execute the block of statements below it otherwise not. If we do not provide the curly braces '{' and '}' after if(condition) then by default if statement will consider the first immediately below statement to be inside its block.

```
// C program to illustrate If statement
#include <stdio.h>
void main()
{
    int i = 10;
    if (i > 15)
   {
        printf("10 is greater than 15");
    }
    printf("I am Not in if");
}
```

2. if-else in C

The *if* statement alone tells us that if a condition is true it will execute a block of statements and if the condition is false it won't. But what if we want to do something else when the condition is false? Here comes the C *else* statement. We can use the *else* statement with the *if* statement to execute a block of code when the condition is false. The if-else statement consists of two blocks, one for false expression and one for true expression.

Syntax of if else in C

```
if (condition)
{
   // Executes this block if
   // condition is true
}
else
{
   // Executes this block if
   // condition is false
}
```

```
// C program to illustrate If-else statement
#include <stdio.h>
void main()
{
    int i = 20;
    if (i < 15)
    {
        printf("i is smaller than 15");
    }
    else
   {
        printf("i is greater than 15");
    }
}
```

3. Nested if-else in C

A nested if in C is an if statement that is the target of another if statement. Nested if statements mean an if statement inside another if statement. Yes, C allow us to nested if statements within if statements, i.e, we can place an if statement inside another if statement.

Syntax of Nested if-else

```
if (condition1)
{
  // Executes when condition1 is true
  if (condition_2)
  {
    // statement 1
  }
  else
  {
      // Statement 2
   }
}
else {
    if (condition_3)
  {
    // statement 3
  }
  else
  {
      // Statement 4
   }
}
```

```
// C program to illustrate nested-if statement
#include <stdio.h>
void main()
{
    int i = 10;
    if (i == 10) {
            if (i < 15)
                printf("i is smaller than 15\n");
            else
                printf("i is greater than 15");
    }
    else {
        if (i == 20) {
                if (i < 22)
                    printf("i is smaller than 22 too\n");
                else
                 printf("i is greater than 25");
        }
    }
}
```

4. if-else-if Ladder in C

The if else if statements are used when the user has to decide among multiple options. The C if statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the C else-if ladder is bypassed. If none of the conditions is true, then the final else statement will be executed. if-else-if ladder is similar to the switch statement.

Syntax of if-else-if Ladder

```
if (condition)
    statement;
else if (condition)
    statement;
.
.
else
    statement;
```

```c
// C program to illustrate nested-if statement
#include <stdio.h>
void main()
{
    int i = 20;
    if (i == 10)
        printf("i is 10");
    else if (i == 15)
        printf("i is 15");
    else if (i == 20)
        printf("i is 20");
    else
        printf("i is not present");
}
```

Goto Statement

The goto statement in C also referred to as the unconditional jump statement can be used to jump from one point to another within a function.

Syntax of goto

```
Syntax1       |   Syntax2
--------------------------
goto label;   |    label:
.             |    .
.             |    .
.             |    .
label:        |    goto label;
```

In the above syntax, the first line tells the compiler to go to or jump to the statement marked as a label. Here, a label is a user-defined identifier that indicates the target statement. The statement immediately followed after ‘label:’ is the destination statement. The ‘label:’ can also appear before the ‘goto label;’ statement in the above syntax.

```
// C program to print numbers from 1 to 10 using goto statement
#include <stdio.h>
void main()
{
    int n = 1;
label:
    printf("%d ", n);
    n++;
    if (n <= 10)
        goto label;
}
```

THANK YOU !!!!

Shri Vile Parle Kelavani Mandal's

SHRI BHAGUBHAI MAFATLAL POLYTECHNIC

Computer Engineering Department

Control Structure Part - II

Course: Programming in C

Course Code: PRC238912

Name of Staff: Mr. Pratik H. Shah

SEMESTER : II

DIVISION : A

Course Outcome – 2

Conceptualize loops and control structure

Lab Outcome:

Student will be able to

- Conceptualize the concept of for loops
- Use for loop in various real life problem statement

For loop

1. for is a Keyword
2. It is used to iterate the statements or a part of the program several times.
3. A for loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

Syntax

```
for ( initialization; condition; Updation )
{
        statement 1;
        statement 2;
        statement 3;
        ……..
}
```

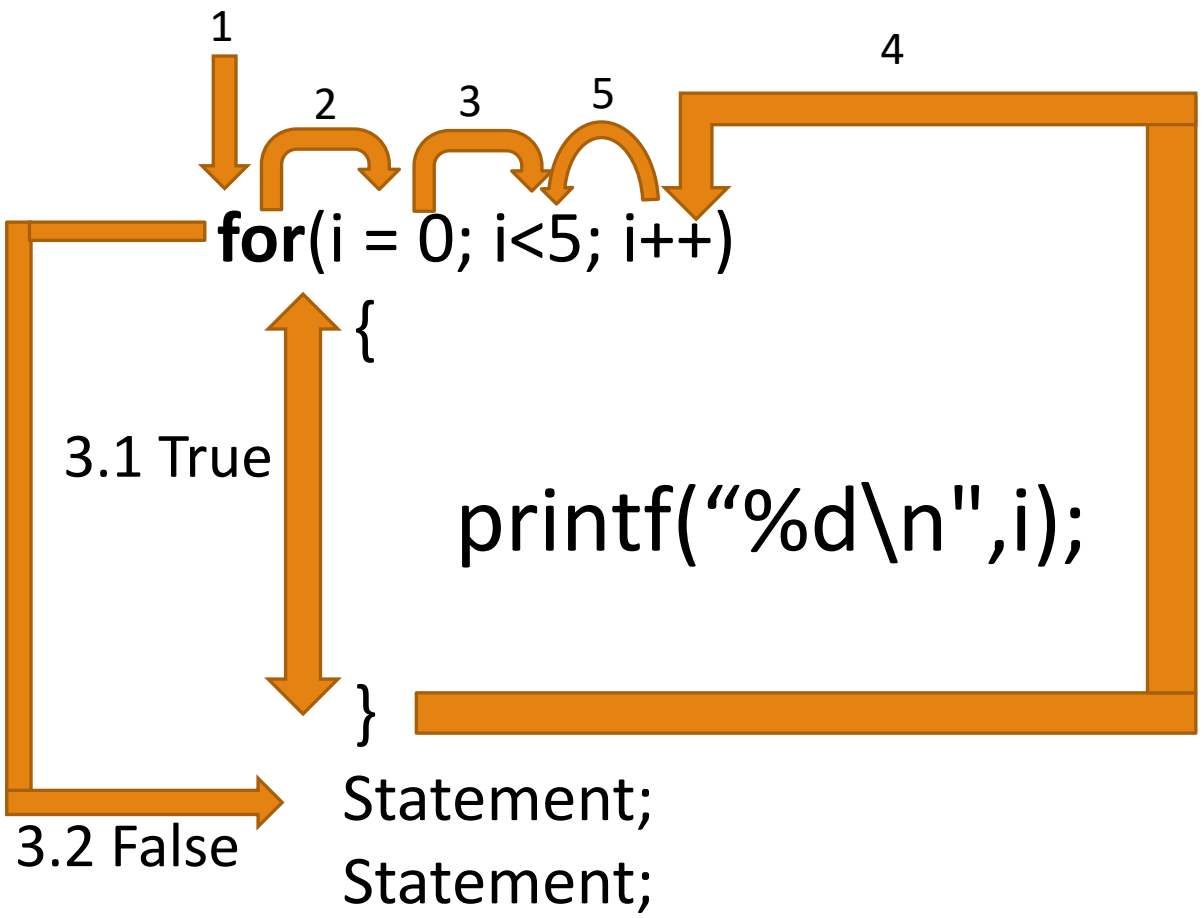
Simple Example using for loop

```
#include<stdio.h>
void main ()
{
        int i;
        for(i = 0; i<5; i++)
        {
                printf(“%d\n",i);
        }
    printf("came outside of loop i = %d",i);
}
```

Output

```
0
1
2
3
4
came outside of loop i=5
```

Explanation: How it Works



Output

0
1
2
3
4

Write a C program to compute the sum of the first 10 natural numbers

```
#include <stdio.h>
void main()
{
    int  j, sum = 0;
    printf("The first 10 natural number is :\n");
    for (j = 1; j <= 10; j++)
    {
        sum = sum + j;
        printf("%d ",j);
    }
    printf("\nThe Sum is : %d\n", sum);
}
```

Output

```
The first 10 natural number is :
1 2 3 4 5 6 7 8 9 10
The Sum is : 55
```

Write a program in C to display the multiplication table for a given integer

```
#include <stdio.h>
void main()
{
    int j,n;
    printf("Input the number (Table to be calculated) : ");
    scanf("%d",&n);
    printf("\n");
    for(j=1;j<=10;j++)
    {
        printf("%d X %d = %d \n",n,j,n*j);
    }
}
```

Output

```
Input the number (Table to be calculated) : 15

15 X 1 = 15
15 X 2 = 30
15 X 3 = 45
15 X 4 = 60
15 X 5 = 75
15 X 6 = 90
15 X 7 = 105
15 X 8 = 120
15 X 9 = 135
15 X 10 = 150
```

Points to Remember

1. For loop is used to evaluate the initialization part first, checking the condition for true or false.
2. If the condition is true, it executes the statements of for loop.
3. After that, it evaluates the increment or decrement condition until the condition becomes false it repeats the same steps.
4. It will exit the loop when the condition is false.

Exercise

1. Write ‘C’ program to find the factorial of a number

Sample Output :

Input a number: 5
Expected Output :
The Factorial of 5 is 120.

Correct Answer

```
#include <stdio.h>
void main()
{
    int i,f=1,num;
    printf("Input a number : ");
    scanf("%d",&num);
    for(i=1;i<=num;i++)
    {
        f=f*i;
     }
     printf("The Factorial of %d is: %d\n",num,f);
}
```

Output

First time Execution:

Input a number: 5
The Factorial of 5 is 120

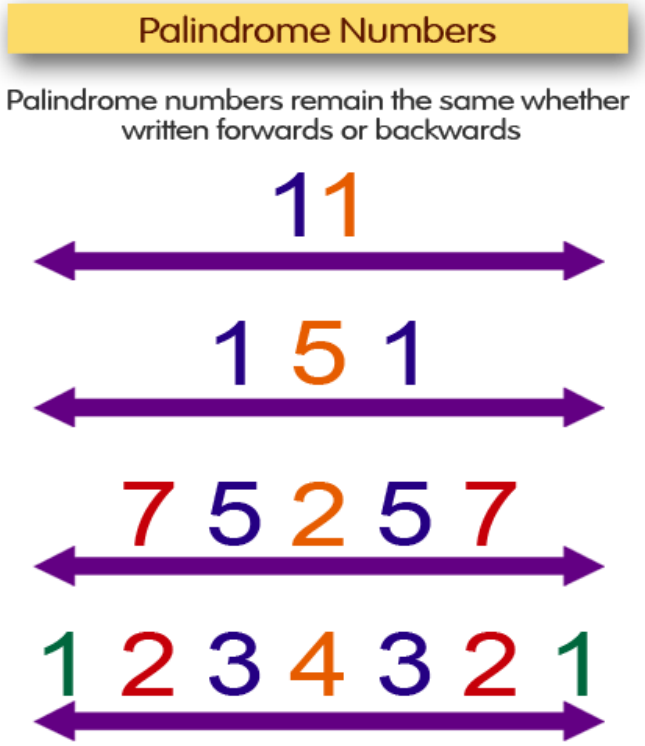
Second time Execution:

Input a number: 4
The Factorial of 4 is 24

Exercise

2. Write ‘C’ program to check whether a entered number is a Palindrome number or not.

Sample Output : Input a number: 151
Expected Output :
151 is a Palindrome number.



Correct Answer

```
#include <stdio.h>
void main()
{
   int num,r,sum=0,t;
   printf("Input a number: ");
   scanf("%d",&num);
   for(t=num;num!=0;num=num/10)
   {
       r=num % 10;
       sum=sum*10+r;
   }
   if(t==sum)
       printf("%d is a palindrome number.\n",t);
   else
       printf("%d is not a palindrome number.\n",t);
}
```

Output

First time Execution:

Input a number: 121
121 is a palindrome number.

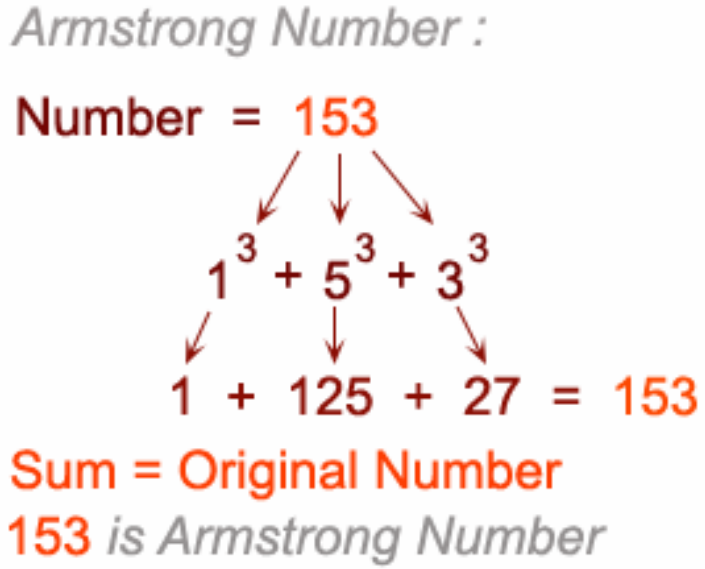
Second time Execution:

Input a number: 134
134 is not a palindrome number.

Exercise

3. Write ‘C’ program to check whether a entered number is an Armstrong number or not.

Sample Output : Input a number: 153
Expected Output :
153 is an Armstrong number.



Correct Answer

```
#include <stdio.h>
void main()
{
   int num,r,sum=0,temp;
   printf("Input  a number: ");
   scanf("%d",&num);
   for(temp=num;num!=0;num=num/10)
  {
       r=num % 10;
       sum=sum+(r*r*r);
   }
   if(sum==temp)
       printf("%d is an Armstrong number.\n",temp);
   else
       printf("%d is not an Armstrong number.\n",temp);
}
```

Output

First time Execution:

Input a number: 153
153 is an Armstrong number.

Second time Execution:

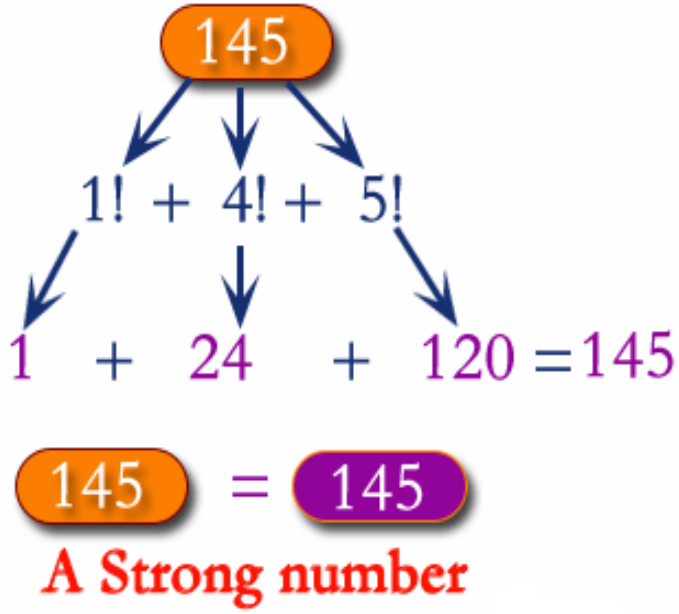
Input a number: 122
122 is not an Armstrong number.

Exercise

4. Write ‘C’ program to check whether a entered number is an strong number or not.

Sample Output :

Input a number: 145
Expected Output :
145 is a Strong number.



Correct Answer

```
#include <stdio.h>
void main()
{
   int i, n, n1, s1=0,j;
   long fact;
   printf("Input a number");
   scanf("%d", &n);
   n1 = n;
   for(j=n;j>0;j=j/10)
   {
      fact = 1;
      for(i=1; i<=j % 10; i++)
      {
         fact = fact * i;
      }
       s1 = s1 + fact;
  }
  if(s1==n1)
      printf("\n%d is Strong number.", n1);
 else
      printf("\n%d is not Strong number.", n1);
}
```

Output

First time Execution:

Input a number: 145
145 is strong number.

Second time Execution:

Input a number: 166
166 is not strong number.

Conclusion

- Can you Conceptualize the concept of for loops ?
- Will you be able to Use for loop in various real life problem statement?

THANK YOU !!!!

Shri Vile Parle Kelavani Mandal's

SHRI BHAGUBHAI MAFATLAL POLYTECHNIC

Computer Engineering Department

Control Structure Part - III

Course: Programming in C

Course Code: PRC238912

Name of Staff: Mr. Pratik H. Shah

SEMESTER : II

DIVISION : A

https://www.javatpoint.com/do-while-loop-in-c

https://www.javatpoint.com/while-loop-in-c

Course Outcome – 2

Conceptualize loops and control structure

Lab Outcome:

Student will be able to

- Conceptualize the concept of while and do-while loops
- Use while and do-while loop in various real life problem statement

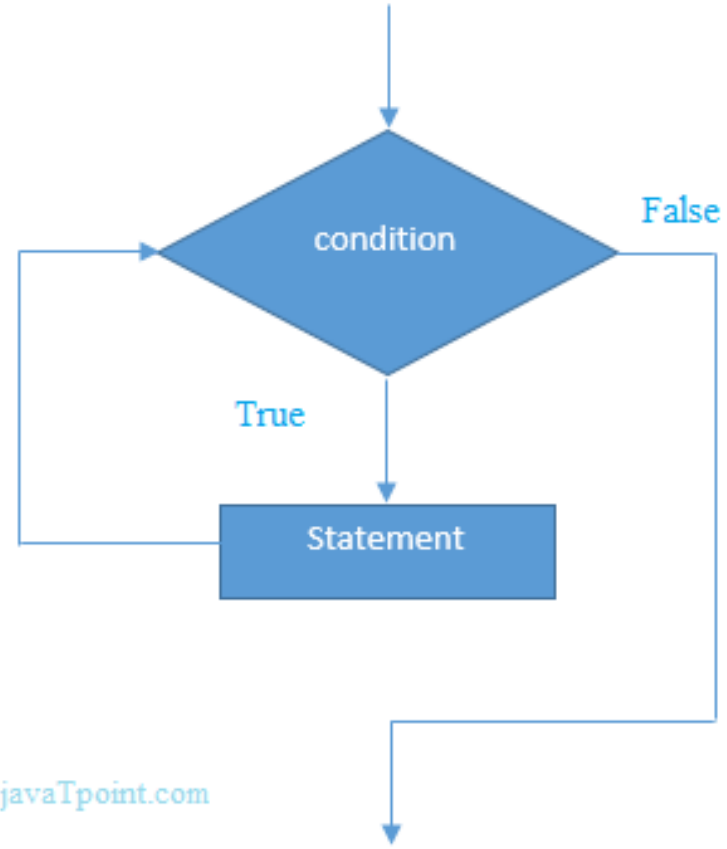
while loop

1. while is a Keyword
2. While loop is also known as a pre-tested loop.
3. In general, a while loop allows a part of the code to be executed multiple times depending upon a given boolean condition.
4. The while loop is mostly used in the case where the number of iterations is not known in advance.

Syntax of while loop in C language

```
while(condition)
{
//code to be executed
}
```

Flowchart of while loop in C



Example of the while loop in C language

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

Program to print table for the given number using while loop in C

```
#include<stdio.h>
void main()
{
int i=1,number=0,b=9;
printf("Enter a number: ");
scanf("%d",&number);
while(i<=10)
{
        printf("%d \n",(number*i));
        i++;
}
}
```

Infinitive while loop in C

If the expression passed in while loop results in any non-zero value then the loop will run the infinite number of times.

```
while(1)
{
//statement
}
```

do while loop in C

Loops are used to execute repeating activities and boost programming performance. **There are multiple loops in the C programming language, one of which is the "do-while" loop.**

A **"do-while" loop is a form of a loop in C that executes the code block first, followed by the condition. If the condition is true, the loop continues to run; else, it stops. However, whether the condition is originally true, it ensures that the code block is performed at least once.**

do while loop syntax

```
do
{
//code to be executed
}while(condition);
```

- The **do keyword marks the beginning of the Loop.**
- The **code block within curly braces {} is the body of the loop, which contains the code you want to repeat.**
- The **while keyword is followed by a condition enclosed in parentheses (). After the code block has been run, this condition is verified. If the condition is true, the loop continues else, the loop ends.**

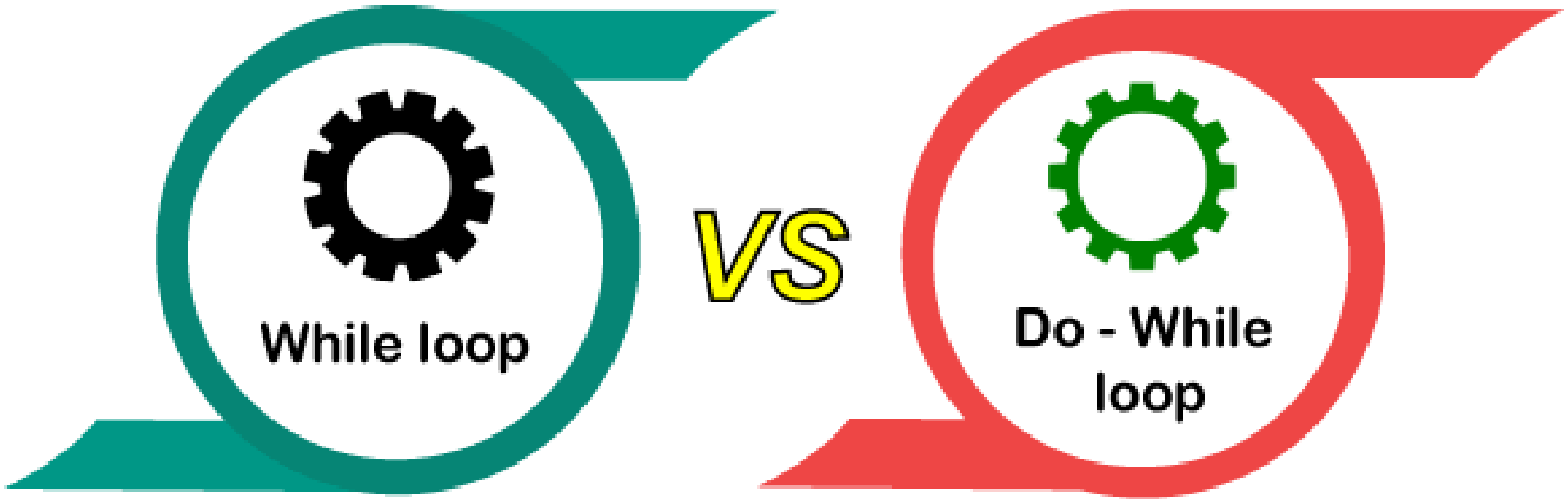
Example of do while loop in C:

```
#include <stdio.h>
void main()
{
inti = 1;
do
{
        printf("%d\n", i);
        i++;
} while (i<= 5);
}
```

Program to print table for the given number using do while Loop

```
#include<stdio.h>
void main()
{
inti=1,number=0;
printf("Enter a number: ");
scanf("%d",&number);
do
{
printf("%d \n",(number*i));
i++;
}while(i<=10);
}
```

Difference Between while and do-while Loop



SR. NO	while loop	do-while loop
1.	While the loop is an entry control loop because firstly, the condition is checked, then the loop's body is executed.	The do-while loop is an exit control loop because in this, first of all, the body of the loop is executed then the condition is checked true or false.
2.	The statement of while loop may not be executed at all.	The statement of the do-while loop must be executed at least once.
3.	The while loop terminates when the condition becomes false.	As long as the condition is true, the compiler keeps executing the loop in the do-while loop.
4.	In a while loop, the test condition variable must be initialized first to check the test condition in the loop.	In a do-while loop, the variable of test condition Initialized in the loop also.

5	**Syntax of while loop:**
	while (condition)
	{
	Block of statements;
	}
	Statement-x;
		Syntax of do-while loop:
		do
		{
		statements;
		} while (condition);
		Statement-x;

Practice Program Questions

- Write following program **using while and do-while loops.**
  - **Armstrong number or not**
  - **Strong number or not**
  - **Palindrome or not**
  - **Neon number or not**
  - **Factorial of a number**
  - **Prime number or not**

Conclusion

- Can you Conceptualize the concept of while and do-while loops ?
- Will you be able to Use while and do-while loop in various real life problem statement?

THANK YOU !!!!

Shri Vile Parle Kelavani Mandal's

SHRI BHAGUBHAI MAFATLAL POLYTECHNIC

Computer Engineering Department

Control Structure Part - IV

Course: Programming in C

Course Code: PRC238912

Name of Staff: Mr. Pratik H. Shah

SEMESTER : II

DIVISION : A

Course Outcome – 2

Conceptualize loops and control structure

Student will be able to

- Use break and continue statement with loops and switch case.
- Differentiate break and continue statement

Break Statement

1. Keyword
2. It is used to bring the program control out of the loop.
3. It is used inside loops or switch statement.
4. It breaks the loop one by one

Syntax: Break; // in case of loop or switch

Break Statement with for loop

```
#include<stdio.h>
void main ()
{
   int i;
   for(i = 0; i<10; i++)
   {
      printf(“%d\n",i);
      if(i == 5)
      break;
   }
   printf("came outside of loop i = %d",i);
}
```

Output

```
0
1
2
3
4
5
came outside of loop i=5
```

Break Statement with while loop

```
#include<stdio.h>
void main ()
{
   int i = 0;
   while(i<20)
   {
      printf("%d  ",i);
      i++;
      if(i == 10)
      break;
   }
   printf("came out of while loop");
}
```

Output

```
0  1  2  3  4  5  6  7  8  9  came out of while loop
```

Break Statement with do-while loop

```
#include <stdio.h>
void main ()
{
    int a = 10;
   do
   {
       if( a == 15)
      {
         a = a + 1;
         break;
      }
      printf("value of a: %d\n", a);
      a++;
   } while( a < 20 );
}
```

Output

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
```

Break Statement with Switch Case

```
#include<stdio.h>
void main()
{
int number;
printf("enter a number:");
scanf("%d",&number);
switch(number)
        {
        case 1:printf(“You have entered 1");
        break;
        case 2:printf(" You have entered 2");
        break;
        case 3:printf(" You have entered 3");
        break;
        default:printf("number is not equal to 1,2 and 3");
        }
}
```

Output

First time Execution:

```
enter a number:2
You have entered 2
```

Second time Execution:

```
enter a number:5
number is not equal to 1,2 and 3
```

Continue Statement

Continue Statement

1. Keyword
2. It is used to bring the program control to the beginning of the loop.
3. skips some lines of code inside the loop and continues with the next iteration.

Syntax: Continue; // in case of loop

Continue Statement with for loop

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

Output

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

Continue Statement with while loop

```
#include <stdio.h>
void main()
 {
    int i = 0;
    while (i < 10)
    {
        if (i == 4)
       {
            i++;
           continue;
       }
       printf("%d\n", i);
       i++;
    }
 }
```

Output

```
0
1
2
3
5
6
7
8
9
```

Continue Statement with do-while loop

```
#include <stdio.h>
void main ()
{
    int a = 10;
   do
   {
       if( a == 15)
      {
         a = a + 1;
         continue;
      }
      printf("value of a: %d\n", a);
      a++;
   } while( a < 20 );
}
```

Output

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

Break vs Continue

Break

1. This statement allows a user to exit a loop's overall structure.
2. It can be written as:
   break;
3. We can use break statement with switch and loops.
4. 'break' resumes the control of the program to the end of loop enclosing that 'break'.

Continue

1. It does not allow a user to exit an overall loop structure.
2. It can be written as:
   continue;
3. We can use continue statement with loops but not with switch.
4. 'continue' resumes the control of the program to the next iteration of that loop enclosing 'continue'.

Conclusion

Both the break and continue statements are jump statements that move control to another section of the programme.

Whereas the break statement allowed the control to exit the loop, the continue statement allowed the control to enter the loop's next iteration.

Exercise

1. What will be the output of the following code:

```
#include <stdio.h>
void main()
{
  int outside,inside;
  for(outside=0;outside<=3;outside++)
  {
      for(inside=0;inside<=3;inside++)
      {
          printf("outside=%d & inside = %d \n",outside,inside);
         if(outside==2 && inside==1)
         {
             break;
         }
      }
   printf("\n");
  }
}
```

Correct Answer

```
outside=0 & inside = 0
outside=0 & inside = 1
outside=0 & inside = 2
outside=0 & inside = 3

outside=1 & inside = 0
outside=1 & inside = 1
outside=1 & inside = 2
outside=1 & inside = 3

outside=2 & inside = 0
outside=2 & inside = 1

outside=3 & inside = 0
outside=3 & inside = 1
outside=3 & inside = 2
outside=3 & inside = 3
```

2. What will be the output of the following code:

```
#include <stdio.h>
void main()
{
    int i;
    for(i=0;i<20;i++)
   {
        if(i%2==0)
       {
            continue;
       }
       printf("%d ",i);
   }
}
```

Correct Answer

1 3 5 7 9 11 13 15 17 19

3. What will be the output of the following code:

```
#include <stdio.h>
void main()
{
    int a = 10;
    do
    {
         if( a == 15)
        {
            a = a + 5;
            continue;
        }
        printf("%d ", a);
        a++;
    } while( a < 30 );
}
```

Correct Answer

10 11 12 13 14 20 21 22 23 24 25 26 27 28 29

4. What ‘C’ program to print following patterns using nested for loop and break statement

1

1 2

1 2 3

1 2 3 4

Correct Answer

```
#include <stdio.h>
void main()
{
        for (int i = 1; i <= 6; i++)
       {
            for (int j = 1; j <= i; j++)
            {
                if (i <= 4)
                {
                    printf("%d ", j);
                }
               else
               {
                   break;
               }
            }
            printf("\n");
       }
}
```

5. Write ‘C’ program to calculate sum of 10 numbers (max 10 numbers) and if the users enters negative number then loop should be terminated

Correct Answer

```
#include <stdio.h>
void main()
{
      int i,number,sum=0;
     for (i = 1; i <= 10; i++)
    {
          printf("Enter n%d: ", i);
          scanf("%d", &number);
          if (number < 0)
         {
               break;
          }
          sum += number;
    }
    printf("Sum = %d", sum);
}
```

Conclusion

Will you be able to now use break and continue statement with loops and switch case?

Can you identify any 2 points for difference between break and continue?

THANK YOU !!!!

Shri Vile Parle Kelavani Mandal's

SHRI BHAGUBHAI MAFATLAL POLYTECHNIC

Computer Engineering Department

Arrays and String

Course: Programming in C

Course Code: PRC238912

Name of Staff: Mr. Pratik H. Shah

SEMESTER : II

DIVISION : A

Course Outcome – 4

Use primary, derived and user defined data types in application programs

Student will be able to

- Use Array as derived data types in developing applications
- Implement various string handling function

Introduction to Arrays

1. An array is a collection of data that holds fixed number of values of same type.
2. An array is a variable that can store multiple values.

Example

if you want to store 100 integers, you can create an array for it.

Types of an Array

1. Single Dimensional Array
2. Multidimensional Array

How to declare an array?

1. To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows

datatype array_name[array_size];

Here, we declared an array, mark, of floating-point type and size 5. Meaning, it can hold 5 floating-point values.

float mark[5];

How to initialize an array?

1. It is possible to initialize an array during declaration.
2. You can initialize an array in C either one by one or using a single statement as follows −

   int mark[5] = {19, 10, 8, 17, 9};

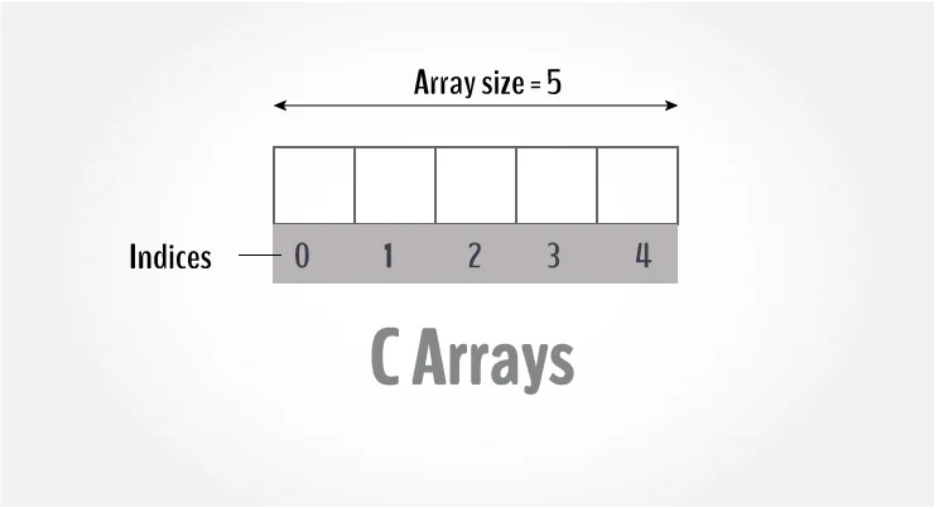
3. The number of values between braces { } cannot be larger than the number of elements that we declare for the array between square brackets [ ].

   int mark[] = {19, 10, 8, 17, 9};

   Here, we haven't specified the size. However, the compiler knows its size is 5 as we are initializing it with 5 elements.

Access Array Elements

1. You can access elements of an array by indices.
2. Suppose you declared an array mark as above.
3. The first element is mark[0], the second element is mark[1] and so on.



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

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

Few keynotes:

1. Arrays have **0 as the first index**, not 1. In this example, mark[0] is the first element.
2. If the size of an array is n, to access the **last element, the n-1** index is used. In this example, mark[4]
3. Suppose the starting **address of mark[0] is 2120d**. Then, the address of the mark[1] will be 2122d. Similarly, the address of mark[2] will be 2124d and so on.
4. This is because the **size of int is 2 bytes**.

Example 1: Array Input/output

```
#include <stdio.h>
void main()
{
  int values[5];
  int i;
  printf("Enter 5 integers: ");

     // taking input and storing it in an array
    for(i = 0; i < 5; i++)
    {
        scanf("%d", &values[i]);
    }

    printf("Displaying integers: ");

    // printing elements of an array
   for(i = 0; i < 5; i++)
  {
      printf("%d\n", values[i]);
   }
}
```

Output

```
Enter 5 integers: 1
-3
34
0
3

Displaying integers: 1
-3
34
0
3
```

Example 2: Calculate Average

```
#include <stdio.h>
void main()
{
   int marks[10], i, n, sum = 0;
   double average;
   printf("Enter number of elements: ");
   scanf("%d", &n);

  for(i=0; i < n; ++i)
  {
      printf("Enter number%d: ",i+1);
      scanf("%d", &marks[i]);
     sum += marks[i];
   }
   average = (double) sum / n;
   printf("Average = %.2lf", average);
}
```

Output

```
Enter number of elements: 5
Enter number1: 45
Enter number2: 35
Enter number3: 38
Enter number4: 31
Enter number5: 49
Average = 39.60
```

Access elements out of its bound!

Suppose you declared an array of 10 elements. Let's say,

int testArray[10];

You can **access** the array elements from **testArray[0] to testArray[9].**

Now let's say if you **try to access testArray[12].** The element is not available. This may cause unexpected output (undefined behavior). Sometimes you might get an error and some other time your program may run correctly.

Hence, you should never access elements of an array outside of its bound.

Multidimensional Arrays

1. In C programming, you can create an array of arrays. These arrays are known as multidimensional arrays

Example

float x[3][4];

- Here, **x is a two-dimensional (2d) array.**
- The array can hold 12 elements. Y
- You can think the array as a table with 3 rows and each row has 4 columns.

	Column 1	Column 2	Column 3	Column 4
Row 1	x[0][0]	x[0][1]	x[0][2]	x[0][3]
Row 2	x[1][0]	x[1][1]	x[1][2]	x[1][3]
Row 3	x[2][0]	x[2][1]	x[2][2]	x[2][3]

Initializing a multidimensional array

```
// Different ways to initialize two-dimensional array

int c[2][3] = {{1, 3, 0}, {-1, 5, 9}};

int c[][3] = {{1, 3, 0}, {-1, 5, 9}};

int c[2][3] = {1, 3, 0, -1, 5, 9};
```

Example 1: 2D Array Input/output

```
#include <stdio.h>
void main()
{
    int a[2][2], i,j;
   printf("Enter elements \n");
   for (i = 0; i < 2; i++)
   {
      for (j = 0; j < 2; j++)
      {
         printf("Enter a%d%d: ", i + 1, j + 1);
         scanf("%d", &a[i][j]);
      }
   }
   for (i = 0; i < 2; i++)
   {
      for (j = 0; j < 2; j++)
      {
         printf("\n a%d%d:%d ", i + 1, j + 1,a[i][j]);
      }
   }
}
```

Output

```
Enter elements
Enter a11: 2
Enter a12: 5
Enter a21: 1
Enter a22: 3

a11:2
a12:5
a21:1
a22:3
```

Pass arrays to a function in C

In C programming, you can pass an entire array to functions.

Example 1: Pass Individual Array Elements

```
#include <stdio.h>
void display(int age1, int age2)
 {
  printf("%d\n", age1);
  printf("%d\n", age2);
}

void main()
{
  int ageArray[] = {2, 8, 4, 12};
  display(ageArray[1], ageArray[2]);
}
```

Output

```
8
4
```

String

Introduction to String

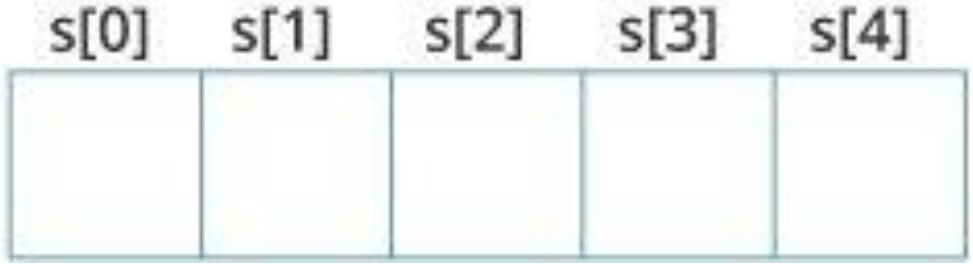
1. In C programming, a string is a sequence of characters terminated with a null character \0.
2. When the compiler encounters a sequence of characters enclosed in the double quotation marks, it appends a null character \0 at the end by default.

Example

```
char c[] = "c string";
```

How to declare a string?

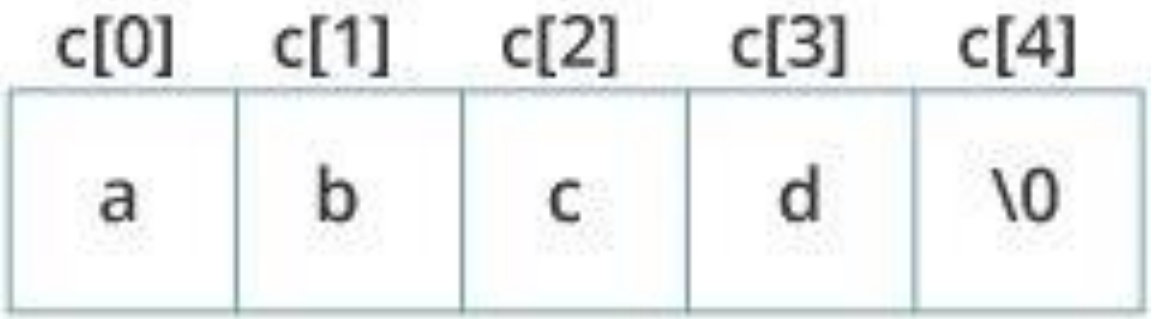
```
char s[5];
```



Here, we have declared a string of 5 characters.

How to initialize strings?

```
// Different ways to initialize String

char c[] = "abcd";

char c[50] = "abcd";

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

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



Assigning Values to Strings

Arrays and strings are second-class citizens in C; they do not support the assignment operator once it is declared

```
char c[100];
c = "C programming";  // Error! array type is not assignable.
```

Note: Use the **strcpy()** to copy the string instead.

Read String from the user

You can use the **scanf()** function to read a string.

The scanf() function reads the sequence of characters until it encounters whitespace (space, newline, tab, etc.).

Example 1: scanf() to read a string

```
#include <stdio.h>
void main()
{
   char name[20];
   printf("Enter name: ");
   scanf("%s", name);
   printf("Your name is %s.", name);
}
```

Output

```
Enter name: Dennis Ritchie
Your name is Dennis.
```

EXPLANATION of PREVIOUS SLIDE PROGRAM

Even though Dennis Ritchie was entered in the above program, **only "Dennis" was stored in the name string.** It's because there was a space after Dennis.

Also notice that we have used the code name instead of &name with scanf().

scanf("%s", name);

This is because name is a char array, and we know that array names decay to pointers in C.

Thus, the name in scanf() already points to the address of the first element in the string, which is why we don't need to use &.

How to read a line of text?

1. You can use the **fgets()** function to **read** a line of string.
2. You can use **puts()** to **display** the string.

Example 2: fgets() and puts()

```
#include <stdio.h>
void main()
{
   char name[30];
   printf("Enter name: ");
   fgets(name, sizeof(name), stdin);  // read string
   printf("Name: ");
   puts(name);          // display string
}
```

Output

```
Enter name: Tom Hanks
Name: Tom Hanks
```

EXPLANATION of PREVIOUS SLIDE PROGRAM

1. Here, we have used fgets() function to read a string from the user.

2. fgets(name, sizeof(name), stdlin); // read string

3. The sizeof(name) results to 30. Hence, we can take a maximum of 30 characters as input which is the size of the name string.

4. To print the string, we have used puts(name);.

5. **Note:** The gets() function can also be to take input from the user. However, it is removed from the C standard.

6. It's because gets() allows you to input any length of characters. Hence, there might be a buffer overflow.

String Handling Function

1. You need to often manipulate strings according to the need of a problem.
2. Most, if not all, of the time string manipulation can be done manually but, this makes programming complex and large.

3. To solve this, C supports a large number of string handling functions in the standard library **"string.h".**

Few commonly used string handling functions

Function	Work of Function
strlen()	computes string's length
strcpy()	copies a string to another
strcat()	concatenates(joins) two strings
strcmp()	compares two strings
strlwr()	converts string to lowercase
strupr()	converts string to uppercase

C strlen()

1. The strlen() function calculates the length of a given string.
2. The strlen() function takes a string as an argument and returns its length.
3. It is defined in the <string.h> header file.
4. **Note:** strlen() function doesn't count the null character \0 while calculating the length.

Example: C strlen() function

```
#include <stdio.h>
#include <string.h>
void main()
{
   char a[20]="Program";

   printf("Length of string a = %d \n",strlen(a));
}
```

Output

```
Length of string a = 7
```

C strcpy()

1. The strcpy() function copies the string pointed by source (including the null character) to the destination.
2. The strcpy() function also returns the copied string.
3. It is defined in the <string.h> header file.
4. **Note:** When you use strcpy(), the size of the destination string should be large enough to store the copied string. Otherwise, it may result in undefined behavior.

Example: C strcpy() function

```
#include <stdio.h>
#include <string.h>
void main()
{
  char str1[20] = "C programming";
  char str2[20];
  strcpy(str2, str1);
  puts(str2);                 // C programming
}
```

Output

C programming

C strcat()

1. the strcat() function contcatenates (joins) two strings.
2. The strcat() function concatenates the destination string and the source string, and the result is stored in the destination string..
3. It is defined in the <string.h> header file.
4. **Note:** When we use strcat(), the size of the destination string should be large enough to store the resultant string. If not, we will get the segmentation fault error.

Example: C strcat() function

```
#include <stdio.h>
#include <string.h>
void main()
{
  char str1[100] = “Hello ", str2[] = “World";
  strcat(str1, str2);
  puts(str1);
  puts(str2);
}
```

Output

Hello World
World

C strcmp()

1. The strcmp() compares two strings character by character.
   If the strings are equal, the function returns 0.

strcmp() Parameters

The function takes two parameters:

str1 - a string

str2 - a string

The strcmp() function is defined in the string.h header file.

Return Value from strcmp()

Return Value	Remarks
0	if strings are equal
> 0	if the first non-matching character in str1 is greater (in ASCII) than that of str2.
< 0	if the first non-matching character in str1 is lower (in ASCII) than that of str2.

Example: C strcmp() function

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

void main()
{
   char str1[] = "abcd", str2[] = "abCd", str3[] = "abcd";
   int result;

      // comparing strings str1 and str2
   result = strcmp(str1, str2);
   printf("strcmp(str1, str2) = %d\n", result);

     // comparing strings str1 and str3
   result = strcmp(str1, str3);
   printf("strcmp(str1, str3) = %d\n", result);

}
```

Output

```
strcmp(str1, str2) = 1
strcmp(str1, str3) = 0
```

Exercise

1. Write a program in C to sort the numbers in Ascending order using Arrays.
2. Write a program in C to scan and print 2 dimensional matrix using Arrays
3. Write a program in C to Add 2 matrices of 3*3 order using Arrays.
4. Write a program in C to search the elements using Arrays
5. Write a program in C to find the determinant of 2*2 order using Arrays
6. Write a program in C to Find the frequency of a character in a string.
7. Write a program in C to Find the number of vowels, consonants, digits and white spaces.
8. Write a program in C to Reverse a string using recursion.
9. Write a program in C to Remove all characters in a string except alphabets.
10. Write a program in C to Sort elements in the lexicographical order (dictionary order)

Theory Questions

1. Define Array
2. Write the syntax of two dimensional array
3. Explain Array to function with example program
4. Define String
5. Write the syntax of any two string function
6. Explain the use of following string function with example program:
   a)strcat() b)strlen() c)strcmp() d)strcpy
7. Enlist all the string handling function

THANK YOU !!!!

Shri Vile Parle Kelavani Mandal's

SHRI BHAGUBHAI MAFATLAL POLYTECHNIC

Computer Engineering Department

Unit 4 -Function

Course: Programming in C

Course Code: PRC238912

Name of Staff: Mr. Pratik H. Shah

SEMESTER : II

DIVISION : A

Course Outcome – 3

Implement modular approach in programming

Student will be able to

- Implement modular approach in programming
- Develop user defined function for real time application

Introduction to Function

1. A function is a block of code that performs a specific task.
2. we can divide a large program into the basic building blocks known as function.
3. A function can be called multiple times to provide reusability and modularity to the C program.

Advantages

1. we can avoid rewriting same logic/code again and again in a program.
2. We can call C functions any number of times in a program and from any place in a program.
3. We can track a large C program easily when it is divided into multiple functions.
4. Reusability is the main achievement of C functions.

Types of function

There are two types of function in C programming:

1. User-defined functions
2. Standard library functions

User-defined function

You can also create functions as per your need. Such functions created by the user are known as user-defined functions.

Standard library functions

1. Built-in functions in C programming.
2. These functions are defined in header files. The printf() is a standard library function to send formatted output to the screen (display output on the screen). This function is defined in the stdio.h header file.
3. To use the printf()function, we need to include the stdio.h header file using #include <stdio.h>.
4. The sqrt() function calculates the square root of a number. The function is defined in the math.h header file.

ELEMENTS OF USER-DEFINED FUNCTION

1. Function Declaration
2. Function Definition
3. Function Call

Function Declaration

1. It is also known as function prototype
2. It specifies function's name, parameters and return type.
3. It doesn't contain function body.
4. A function prototype gives information to the compiler that the function may later be used in the program.

Syntax of function prototype

```
returnType functionName(type1 parameter1, type2 parameter2, ...);
```

Function Definition

1. Function definition contains the block of code to perform a specific task.

Syntax of function definition

```
returnType functionName(type1 parameter1, type2 parameter2, ...)
{
   //body of the function
}
```

Function Call

1. Control of the program is transferred to the user-defined function by calling it.

Syntax of function call

```
functionName(argument1, argument2, ...);
```

Example: User-defined function

```
#include <stdio.h>
void addNumbers();                    // function declaration
void addNumbers()                     // function definition
{
    int result,a,b;
     a = 5;
     b = 10;
    result = a+b;
    printf("result=%d",result);
}

void main()
{
    addNumbers();                     // function call
}
```

What is Parameter ?

- In C Programming Function Passing Parameter is Optional.
- We can Call Function Without Passing Parameter .
- **Function With Parameter :**
  - add(a,b);
- Here Function add() is Called and 2 Parameters are Passed to Function.
- a,b are two Parameters.
- **Function Call Without Passing Parameter :**
  - Display();

1. Parameter : The names given in the function definition are called Parameters.
   - Formal Parameter :

     Parameter Written In Function Definition is Called “Formal Parameter”.
   - Actual Parameter :

   Parameter Written In Function Call is Called “Actual Parameter”.

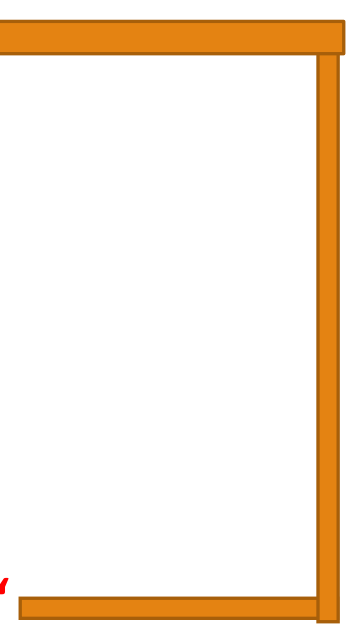
2. Argument : The values supplied in the function call are called Arguments.

```
void main()
{
int num1;
display(num1);
}

void display(int para1)
{
----------
----------
}
```

Para1 is “Formal Parameter“

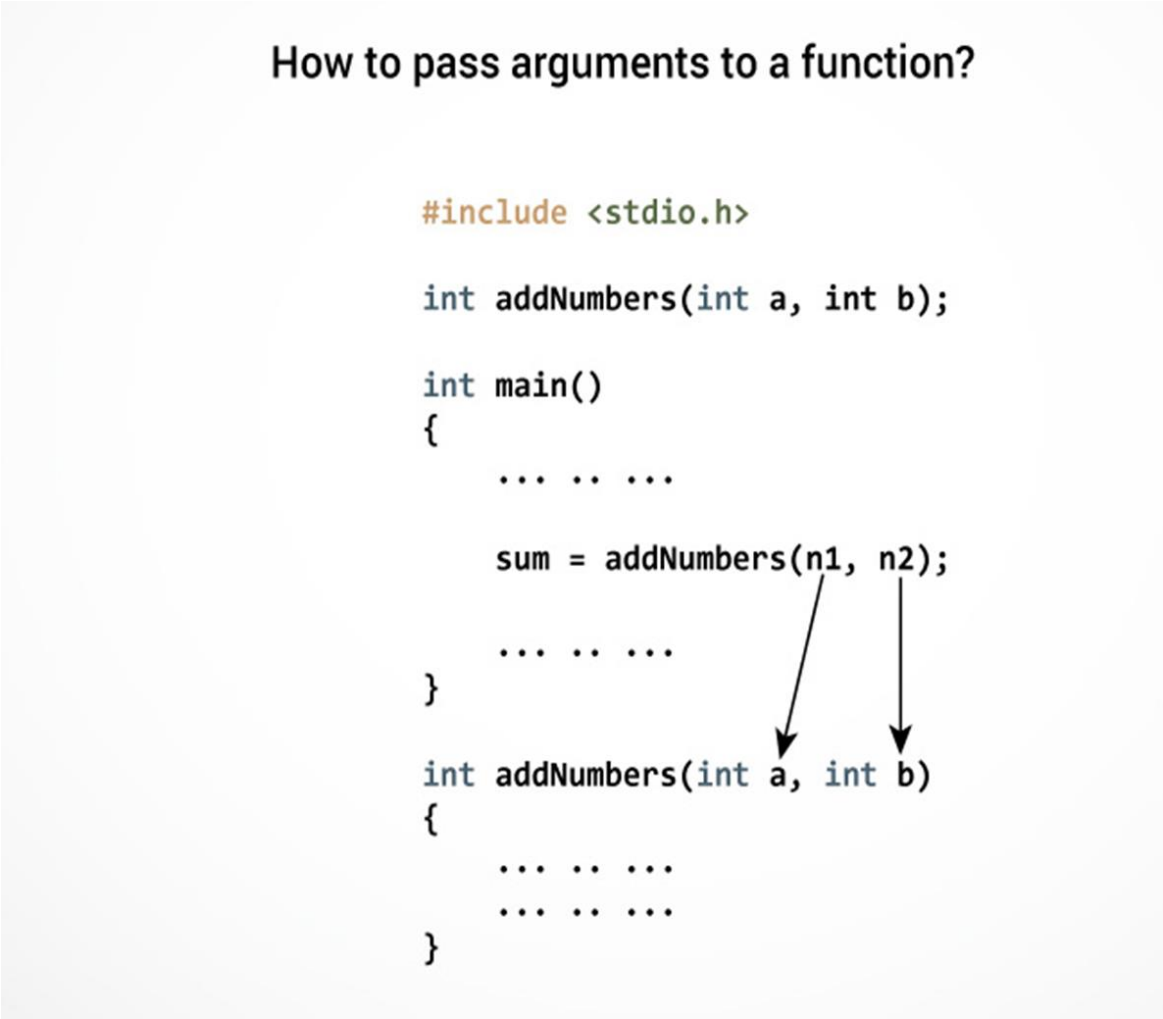
```
void main()
{
int num1;
display(num1);
}

void display(int para1)
{
----------
----------
}
```

num1 is “Actual Parameter“



Passing arguments to a function

In programming, argument refers to the variable passed to the function.



Return Statement

1. The return statement terminates the execution of a function and returns a value to the calling function.

2. The program control is transferred to the calling function after return statement.

Syntax of return statement

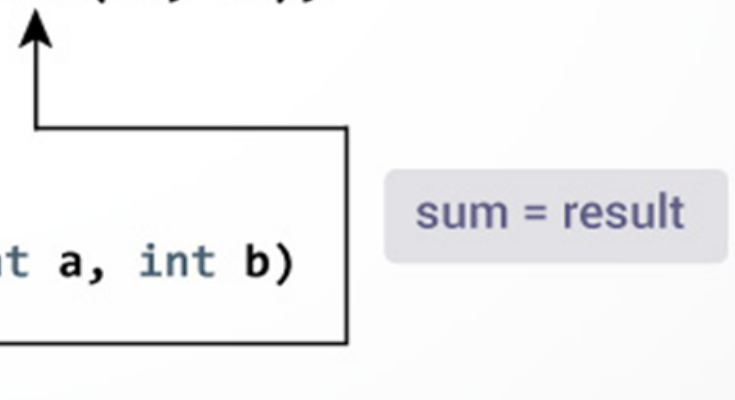
```
return (expression);
```

Example

```
return a;
return (a+b);
```

Return statement of a Function

```c
#include <stdio.h>

int addNumbers(int a, int b);

int main()
{
    ... .. ...

    sum = addNumbers(n1, n2);

    ... .. ...
}

int addNumbers(int a, int b)
{
    ... .. ...
    return result;
}
```



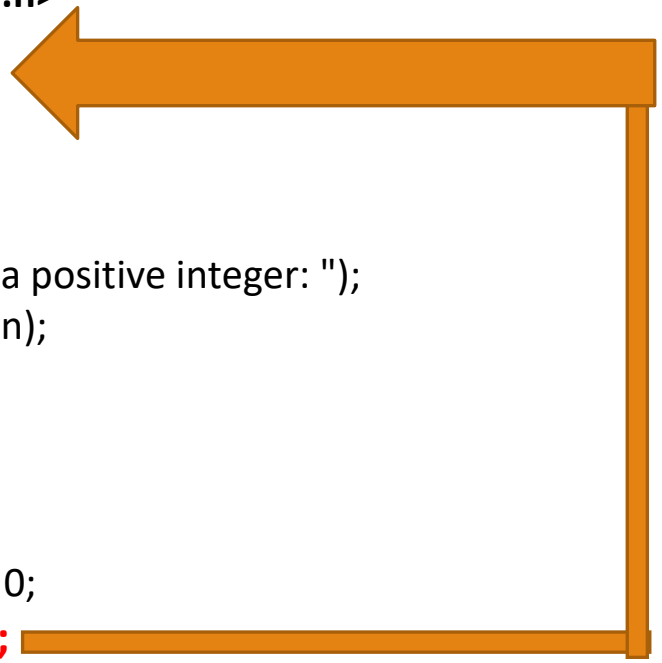
Example #1:
No arguments passed and no return Value

```
#include <stdio.h>
void prime();
void prime()
{
   int n, i, flag=0;
   printf("Enter a positive integer: ");
   scanf("%d",&n);
   for(i=2; i <= n/2; i++)
   {
      if(n%i == 0)
      {
         flag = 1;
      }
   }
   if (flag == 1)
      printf("%d is not a prime number.", n);
   else
      printf("%d is a prime number.", n);
}
void main()
{
   prime();
}
```

Example #2:

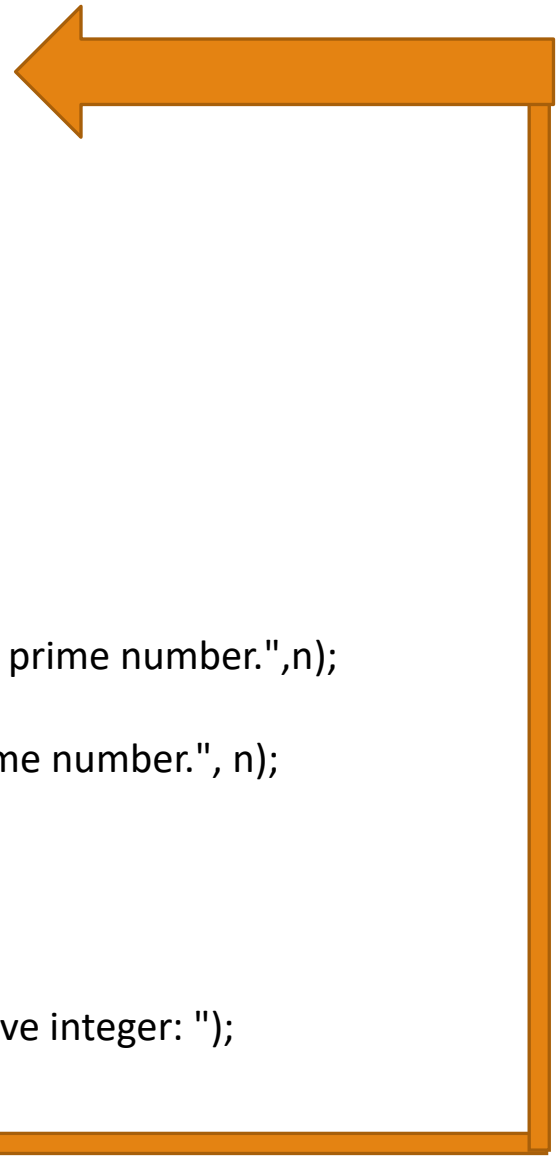
No arguments passed but a return value

```
#include <stdio.h>
int prime();
int prime()
{
   int n;
   printf("Enter a positive integer: ");
   scanf("%d",&n);
   return n;
}
void main()
{
   int n, i, flag = 0;
   n = prime();
   for(i=2; i<=n/2;i++)
   {
      if(n%i==0){
         flag = 1;
         break;
      }
   }
   if (flag == 1)
      printf("%d is not a prime number.", n);
   else
      printf("%d is a prime number.", n);
}
```



Example #3:
Argument passed but no return value

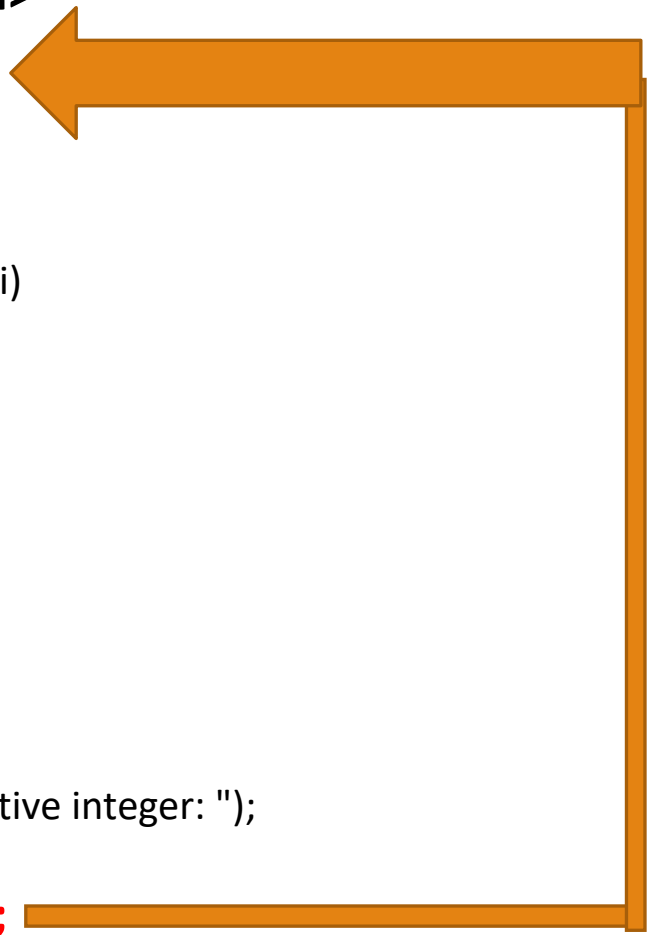
```
#include <stdio.h>
void prime(int n);
void prime(int n)
{
   int i, flag = 0;
   for(i=2; i <= n/2; ++i)
   {
      if(n%i == 0){
         flag = 1;
         break;
      }
   }
   if(flag == 1)
      printf("%d is not a prime number.",n);
   else
      printf("%d is a prime number.", n);
}
void main()
{
   int n;
   printf("Enter a positive integer: ");
   scanf("%d",&n);
   prime(n);
}
```



Example #4:

Argument passed and a return value

```
#include <stdio.h>
int prime(int n);
int prime(int n)
{
   int i;
   for(i=2; i <= n/2; ++i)
   {
      if(n%i == 0)
         return 1;
   }
   return 0;
}
void main()
{
   int n, flag;
   printf("Enter a positive integer: ");
   scanf("%d",&n);
   flag = prime(n);
   if(flag==1)
      printf("%d is not a prime number",n);
   else
      printf("%d is a prime number",n);
}
```



Which approach is better?

1. Well, it depends on the problem you are trying to solve.

2. In case of this problem, the last approach is better.

3. A function should perform a specific task.

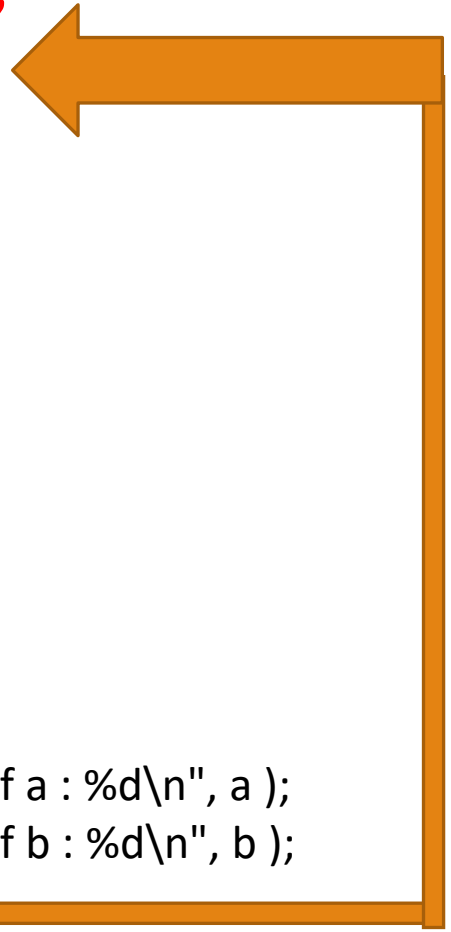
4. The prime() function doesn't take input from the user nor it displays the appropriate message.

5. It only checks whether a number is prime or not, which makes code modular, easy to understand and debug.

call by value

1. The **call by value** method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function.
2. In this case, changes made to the parameter inside the function have no effect on the argument.
3. By default, C programming uses *call by value* to pass arguments.
4. In general, it means the code within a function cannot alter the arguments used to call the function.

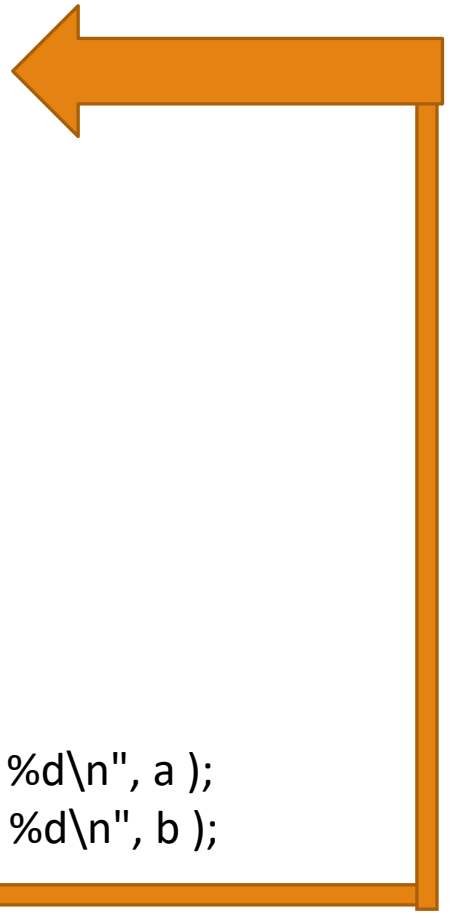
```
#include <stdio.h>
void swap(int x, int y);
void swap(int x, int y)
{
   int temp;
   temp = x;
   x = y;
   y = temp;

}
void 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 );
}
```



call by reference

1. The **call by reference** method of passing arguments to a function copies the address of an argument into the formal parameter.
2. Inside the function, the address is used to access the actual argument used in the call.
3. It means the changes made to the parameter affect the passed argument.
4. To pass a value by reference, argument pointers are passed to the functions just like any other value.
5. So accordingly you need to declare the function parameters as pointer types as in the following function **swap()**, which exchanges the values of the two integer variables pointed to, by their arguments.

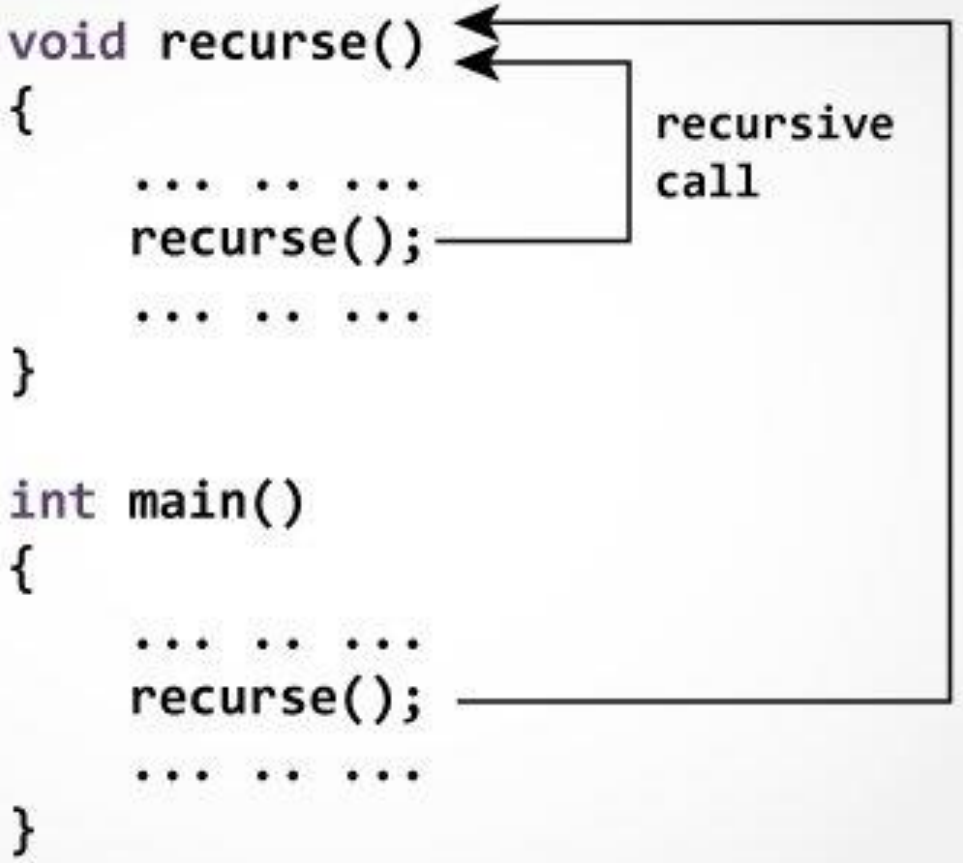
```
#include <stdio.h>
void swap(int *x, int *y);
void swap(int *x, int *y)
{
   int temp;
   temp = *x;
   *x = *y;
  *y = temp;
}
void main ()
{
int a = 100;
int b = 200;
   printf("Before swap, value of a : %d\n", a );
   printf("Before swap, value of b : %d\n", b );
    swap(&a, &b);
   printf("After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



Recursion Function

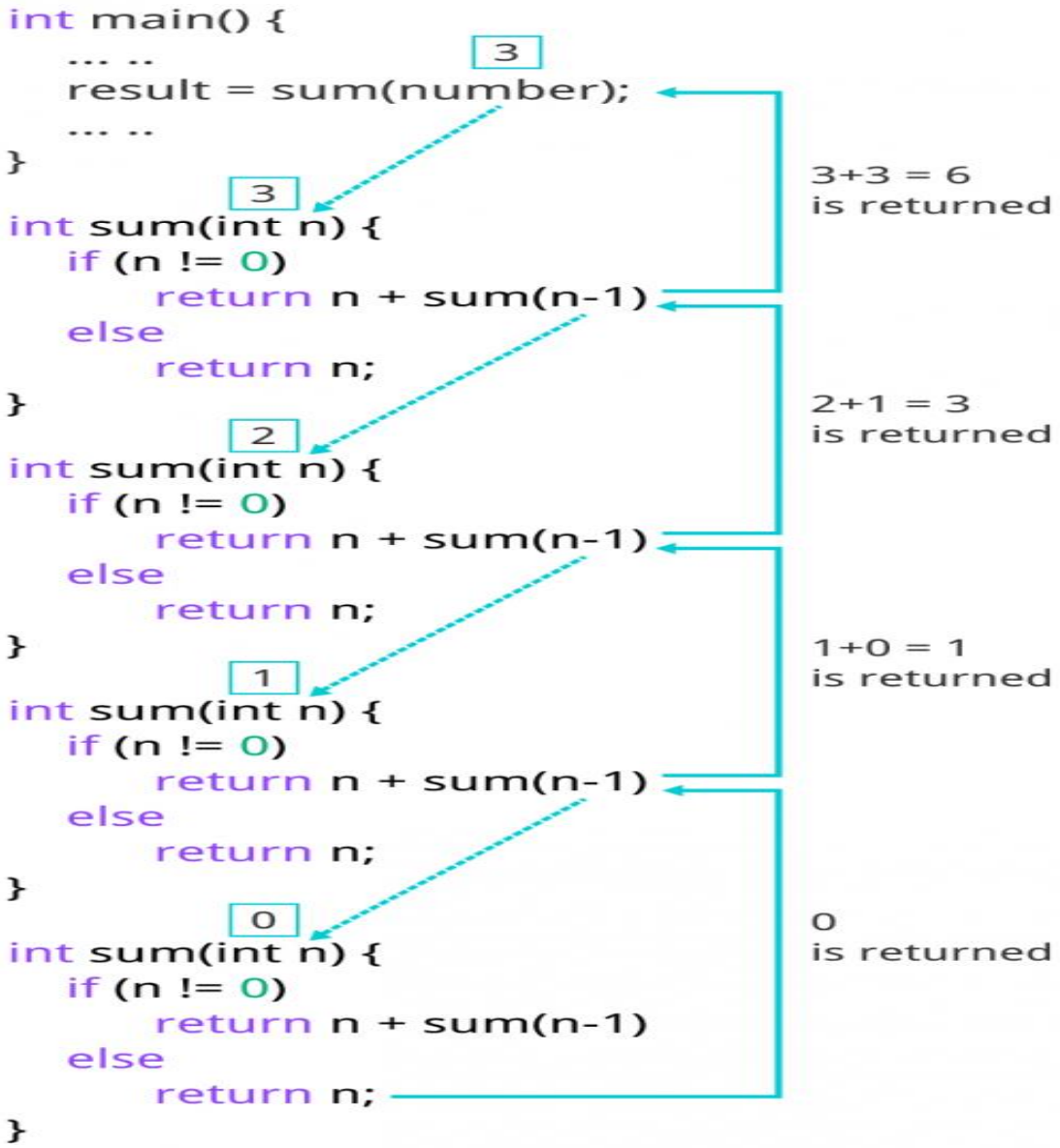
1. A function that calls itself is known as a recursive function. And, this technique is known as recursion.
2. Recursion is the process of repeating items in a self-similar way.
3. In programming languages, if a program allows you to call a function inside the same function, then it is called a recursive call of the function.
4. The C programming language supports recursion, i.e., a function to call itself.
5. But while using recursion, programmers need to be careful to define an exit condition from the function, otherwise it will go into an infinite loop.
6. Recursive functions are very useful to solve many mathematical problems, such as calculating the factorial of a number, generating Fibonacci series, etc.

How does recursion work?



Example: Sum of Natural Numbers Using Recursion

```
#include <stdio.h>
int sum(int num);
int sum(int num)
{
   if (num!=0)
      return num + sum(num-1); // sum() function calls itself
   else
      return num;
}
void main()
{
   int number, result;

   printf("Enter a positive integer: ");
   scanf("%d", &number);

   result = sum(number);

   printf("sum=%d", result);
}
```


int main() {
... ..
result = sum(number);
... ..
}
3
int sum(int n) {
if (n != 0)
return n + sum(n-1)
else
return n;
}
3
int sum(int n) {
if (n != 0)
return n + sum(n-1)
else
return n;
}
2
int sum(int n) {
if (n != 0)
return n + sum(n-1)
else
return n;
}
1
int sum(int n) {
if (n != 0)
return n + sum(n-1)
else
return n;
}
0
3+3 = 6
is returned
2+1 = 3
is returned
1+0 = 1
is returned
0
is returned

Advantages of Recursion

1. Recursion makes program elegant and cleaner.
2. All algorithms can be defined recursively which makes it easier to visualize and prove.

Disadvantages of Recursion

1. If the speed of the program is vital then, you should avoid using recursion.
2. Recursions use more memory and are generally slow. Instead of that, you can use loop

Exercise

1. Write a program in C to check if a given number is even or odd using the function.
2. Write a program in C to swap two numbers using a function**(Call by value and call by reference method**).
3. Write a program in C to check whether a number is a prime number or not using the function.
4. Write a program in C to calculate the sum of numbers from 1 to n using recursion.
5. Write a program in C to print the Fibonacci Series using recursion.
6. Write a program in C to find the Factorial of a number using recursion.
7. Write a program in C to print the first 50 natural numbers using recursion
8. Write a program in C to search a number from a list using the function.

Theory Questions

1. Explain Elements of User defined Function with example
2. Define Recursion
3. Explain different categories of function with example programs
4. Difference between call by value and call by reference
5. Define following terms: a) Parameter b) Argument
6. State Advantages of function
7. State advantages and disadvantages of Recursion function

THANK YOU !!!!

Shri Vile Parle Kelavani Mandal's

SHRI BHAGUBHAI MAFATLAL POLYTECHNIC

Computer Engineering Department

Pointers and Structure

Course: Programming in C

Course Code: PRC238912

Name of Staff: Mr. Pratik H. Shah

SEMESTER : II

DIVISION : A

Course Outcome – 4

Use primary, derived and user defined data types in application programs

Student will be able to

- Use Pointer as derived data types in developing applications
- Implement Structure as user defined datatype

Introduction to Pointers

1. Pointers are powerful features of C and (C++) programming that differentiates it from other popular programming languages like: Java and Python.
2. Pointers are used in C program to access the memory and manipulate the address.
3. **A pointer is a variable whose value is the address of another variable**, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before using it to store any variable address.

Pointer Syntax

In C, there is a special variable that stores just the address of another variable. It is **called Pointer variable** or, simply, a pointer.

Declaration of Pointer

```
data_type* pointer_variable_name;
int* p;
```

Above statement defines, p as pointer variable of type int.

Pointer Declaration

int* p;

Here, we have declared a pointer p of int type.
You can also declare pointers in these ways.

int *p1;
int * p2;

Let's take another example of declaring pointers.

int* p1, p2;

Here, we have declared a pointer p1 and a normal variable p2.

Reference operator (&) and Dereference operator (*)

1. As discussed, **& is called reference operator.** It gives you the address of a variable.
2. Likewise, there is another operator that gets you the value from the address, it is called a **dereference operator (*).**

3. Note: The * sign when declaring a pointer is not a dereference operator. It is just a similar notation that creates a pointer.

Memory Address

Before you get into the concept of pointers, let's first get familiar with address in C.

If you have a variable var in your program, &var will give you its address in the memory, where **&** is commonly called the **reference operator.**

You must have seen this notation while using scanf() function. It was used in the function to store the user inputted value in the address of var.

```
scanf("%d", &var);
```

/* Example to demonstrate use of reference operator in C programming. */

```
#include <stdio.h>
void main()
{
  int var = 5;
  printf("Value: %d\n", var);
  printf("Address: %u", &var);  //Notice, the ampersand(&) before var.
}
```

Output

```
Value: 5
Address: 2686778
```

Assigning addresses to Pointers

```
int* pc, c;
c = 5;
pc = &c;
```

Here, 5 is assigned to the c variable. And, the address of c is assigned to the pc pointer.

Get Value of Thing Pointed by Pointers

To get the value of the thing pointed by the pointers, we use the * operator.

```
int* pc, c;
c = 5;
pc = &c;
printf("%d", *pc);       // Output: 5
```

Here, the address of c is assigned to the pc pointer. To get the value stored in that address, we used *pc.

Changing Value Pointed by Pointers

```
int* pc, c;
c = 5;
pc = &c;
c = 1;
printf("%d", c);          // Output: 1
printf("%d", *pc);      // Ouptut: 1
```

Another Example

```
int* pc, c;
c = 5;
pc = &c;
*pc = 1;
printf("%d", *pc);       // Output: 1
printf("%d", c);         // Output: 1
```

We have assigned the address of c to the pc pointer.

Then, we changed *pc to 1 using *pc = 1;. Since pc and the address of c is the same, c will be equal to 1.

Another Example

```
int* pc, c, d;
c = 5;
d = -15;

pc = &c;
printf("%d", *pc);          // Output: 5
pc = &d;
printf("%d", *pc);         // Ouptut: -15
```

/* Source code to demonstrate, handling of pointers in C program */

```
#include <stdio.h>
void main()
{
  int* pc;
  int c;
  c=22;
  printf("Address of c:%u\n",&c);
  printf("Value of c:%d\n\n",c);
  pc=&c;
  printf("Address of pointer pc:%u\n",pc);
  printf("Content of pointer pc:%d\n\n",*pc);
  c=11;
  printf("Address of pointer pc:%u\n",pc);
  printf("Content of pointer pc:%d\n\n",*pc);
  *pc=2;
  printf("Address of c:%u\n",&c);
  printf("Value of c:%d\n\n",c);
}
```

Output

Address of c: 2686784
Value of c: 22

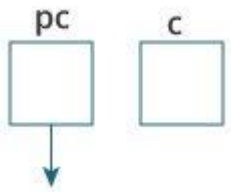
Address of pointer pc: 2686784
Content of pointer pc: 22

Address of pointer pc: 2686784
Content of pointer pc: 11

Address of c: 2686784
Value of c: 2

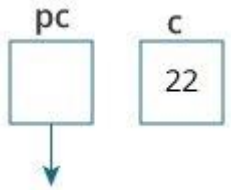
Explanation of the program

1. **int* pc**, c;



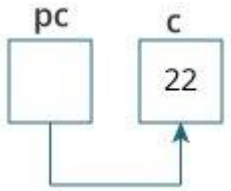
Here, a pointer pc and a normal variable c, both of type int, is created.
Since pc and c are not initialized at initially, pointer pc points to either no address or a random address. And, variable c has an address but contains random garbage value.

2. c = 22;



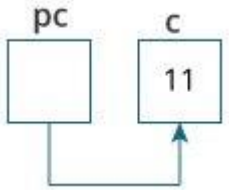
This assigns 22 to the variable c. That is, 22 is stored in the memory location of variable c.

3. pc = &c;



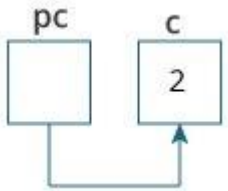
This assigns the address of variable c to the pointer pc.

4. c = 11;



This assigns 11 to variable c.

5. *pc = 2;



This change the value at the memory location pointed by the pointer pc to 2.

Common mistakes when working with pointers

Suppose, you want pointer pc to point to the address of c. Then,
int c, *pc;

```
// pc is address but c is not
pc = c;                    // ERROR

// &c is address but *pc is not
*pc = &c;                  // ERROR

// both &c and pc are addresses
pc = &c;                   // NOT AN ERROR

// both c and *pc are values
*pc = c;                   // NOT AN ERROR
```

Relationship Between Arrays and Pointers

An array is a block of sequential data. Let's write a program to print addresses of array elements.

```
#include <stdio.h>
void main()
{
      int x[4];
      int i;

      for(i = 0; i < 4; i++)
      {
           printf("&x[%d] = %p\n", i, &x[i]);
       }
       printf("Address of array x: %p", x);
}
```

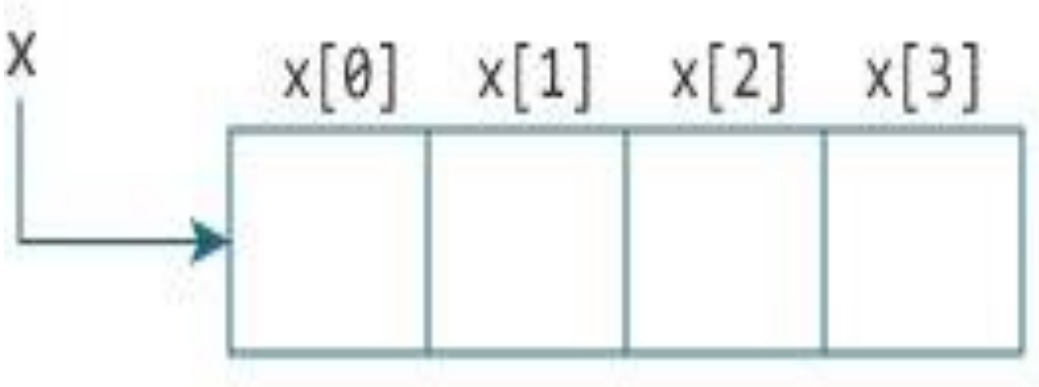
OUTPUT:

```
&x[0] = 1450734448
&x[1] = 1450734452
&x[2] = 1450734456
&x[3] = 1450734460
Address of array x: 1450734448
```

%p is a format specifier which is used for displaying the address in hex format.

There is a difference of **4 bytes** between two consecutive elements of **array x**. It is because the **size of int is 4 bytes** (on our compiler).

Notice that, the address of **&x[0]** and **x is the same**. It's because the variable name x points to the first element of the array.



Pointers and Arrays

```
#include <stdio.h>
void main()
{
        int i, x[6], sum = 0;
        printf("Enter 6 numbers: ");
        for(i = 0; i < 6; ++i)
       {
            scanf("%d", x+i);                  // Equivalent to scanf("%d", &x[i]);
            sum += *(x+i);                     // Equivalent to sum += x[i]
       }
       printf("Sum = %d", sum);
}
```

Arrays and Pointers

```
#include <stdio.h>
void main()
{
  int x[5] = {1, 2, 3, 4, 5};
  int* ptr;

  ptr = &x[2]; // ptr is assigned the address of the third element

  printf("*ptr = %d \n", *ptr);                  // 3
  printf("*(ptr+1) = %d \n", *(ptr+1));          // 4
  printf("*(ptr-1) = %d", *(ptr-1));             // 2
}
```

Array of Pointer

```
#include <stdio.h>
const int MAX = 3;
void main ()
{
  int  var[] = {10, 100, 200};
  int i, *ptr[MAX];
  for ( i = 0; i < MAX; i++)
  {
     ptr[i] = &var[i];
  }
  for ( i = 0; i < MAX; i++)
  {
    printf("Value of var[%d] = %d\n", i, *ptr[i]);
  }
}
```

Pointers and Functions

In C programming, it is also possible to **pass addresses as arguments** to functions.

To accept these addresses in the function definition, we can **use pointers**. It's because pointers are used to store addresses.

Example 1 : Pass Addresses to Functions

```
#include <stdio.h>
void swap(int *n1, int *n2);

void main()
{
   int num1 = 5, num2 = 10;
    swap( &num1, &num2); // address of num1 and num2 is passed

   printf("num1 = %d\n", num1);
   printf("num2 = %d", num2);
}
void swap(int* n1, int* n2)
{
   int temp;
    temp = *n1;
    *n1 = *n2;
    *n2 = temp;
}
```

Example 2: Passing Pointers to Functions

```
#include <stdio.h>
void addOne(int* ptr)
{
     (*ptr)++;          // adding 1 to *ptr
}
void main()
{
  int* p, i = 10;
  p = &i;
  addOne(p);

  printf("%d", *p);          // 11
}
```

C Dynamic Memory Allocation

As you know, an array is a collection of a fixed number of values. Once the size of an array is declared, you cannot change it.

Sometimes the size of the array you declared may be insufficient. To solve this issue, you can allocate memory manually during run-time. This is known as **dynamic memory allocation in C programming.**

To allocate memory dynamically, library functions are **malloc(), calloc(), realloc() and free()** are used. These functions are defined in the **<stdlib.h> header file.**

Function	Use of Function
malloc()	Allocates requested size of bytes and returns a pointer first byte of allocated space
calloc()	Allocates space for an array elements, initializes to zero and then returns a pointer to memory
free()	deallocate the previously allocated space
realloc()	Change the size of previously allocated space

C malloc()

1. The name malloc stands for "memory allocation".
2. The function malloc() reserves a block of memory of specified size and return a pointer of type void which can be casted into pointer of any form.

Syntax of malloc()

ptr = (cast-type*) malloc(byte-size)

1. Here, ptr is pointer of cast-type.
2. The malloc() function returns a pointer to an area of memory with size of byte size. If the space is insufficient, allocation fails and returns NULL pointer.

ptr = (int*) malloc(100 * sizeof(int));

C calloc()

The name calloc stands for "contiguous allocation".

The only difference between malloc() and calloc() is that, malloc() allocates single block of memory whereas calloc() allocates multiple blocks of memory each of same size and sets all bytes to zero.

Syntax of calloc()

ptr = (cast-type*)calloc(n, element-size);

This statement will allocate contiguous space in memory for an array of n elements.

For example:

ptr = (float*) calloc(25, sizeof(float));

This statement allocates contiguous space in memory for an array of 25 elements each of size of float, i.e, 4 bytes.

C free()

Dynamically allocated memory created with either calloc() or malloc() doesn't get freed on its own. You must explicitly use free() to release the space.

syntax of free()

```
free(ptr);
```

This statement frees the space allocated in the memory pointed by ptr.

Example #1: Using C malloc() and free()

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
   int num, i, *ptr, sum = 0;
   printf("Enter number of elements: ");
   scanf("%d", &num);
   ptr = (int*) malloc(num * sizeof(int));  //memory allocated using malloc
   if(ptr == NULL)
   {
      printf("Error! memory not allocated.");
      exit(0);
   }
   printf("Enter elements of array: ");
   for(i = 0; i < num; i++)
   {
      scanf("%d", ptr + i);
      sum += *(ptr + i);
   }
   printf("Sum = %d", sum);
   free(ptr);
}
```

Write a C program to find sum of n elements entered by user.
To perform this program, allocate memory dynamically using malloc() function.

Example #2: Using C calloc() and free()

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
   int num, i, *ptr, sum = 0;
   printf("Enter number of elements: ");
   scanf("%d", &num);
   ptr = (int*) calloc(num, sizeof(int));
   if(ptr == NULL)
   {
      printf("Error! memory not allocated.");
      exit(0);
   }
   printf("Enter elements of array: ");
   for(i = 0; i < num; i++)
   {
      scanf("%d", ptr + i);
      sum += *(ptr + i);
   }
   printf("Sum = %d", sum);
   free(ptr);
}
```

Write a C program to find sum of n elements entered by user. To perform this program, allocate memory dynamically using calloc() function.

C realloc()

If the previously allocated memory is insufficient or more than required, you can change the previously allocated memory size using realloc().

Syntax of realloc()

```
ptr = realloc(ptr, newsize);
```

Here, ptr is reallocated with size of newsize.

Example #3: Using realloc()

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
   int *ptr, i , n1, n2;
   printf("Enter size of array: ");
   scanf("%d", &n1);

   ptr = (int*) malloc(n1 * sizeof(int));
   printf("Address of previously allocated memory: ");
   for(i = 0; i < n1; i++)
       printf("%u\t",ptr + i);

   printf("\nEnter new size of array: ");
   scanf("%d", &n2);

   ptr = realloc(ptr, n2);

   for(i = 0; i < n2; i++)
       printf("%u\t", ptr + i);
}
```

Structure

Introduction

Structure is a collection of variables of different types under a single name.

For example:
You want to store some information about a person: his/her name, citizenship number and salary.

You can easily create different variables name, citNo, salary to store these information separately.

Define Structures

```
struct structureName
{
    dataType member1;
    dataType member2;
    ...
};
```

Example

```
struct Person
{
    char name[50];
    int citNo;
    float salary;
};
```

Create struct Variables

1. When a struct type is declared, no storage or memory is allocated.
2. To allocate memory of a given structure type and work with it, we need to create variables.

First Method

```
struct Person
{
    // code
};

void main()
{
  struct Person person1, person2, p[20];
  // remaining code
}
```

Second Method

```
struct Person
{
    // code
} person1, person2, p[20];
```

Access Members of a Structure

There are two types of operators used for accessing members of a structure.

- **. Member operator**
- **-> Structure pointer operator**

Suppose, you want to access the salary of person2. Here's how you can do it.

```
person2.salary
```

Example 1: C structs

```
#include <stdio.h>
#include <string.h>
struct Person
{
   char name[50];
   int citNo;
   float salary;
} person1;

void main()
{
      strcpy(person1.name, "George Orwell");          // assign value to name of person1
       person1.citNo = 1984;
       person1. salary = 2500;

     printf("Name: %s\n", person1.name);
     printf("Citizenship No.: %d\n", person1.citNo);
     printf("Salary: %.2f", person1.salary);
}
```

Nested Structures

The structure can be nested in the following ways.

1. **By Separate structure**
2. **By Embedded structure**

1) Separate structure

Here, we create two structures, but the dependent structure should be used inside the main structure as a member.

2) Embedded structure

The embedded structure enables us to declare the structure inside the structure. Hence, it requires less line of codes but it can not be used in multiple data structures. Consider the following example.

Example

1) Separate structure

```
struct Date
{
   int dd;
   int mm;
   int yyyy;
};
struct Employee
{
   int id;
   char name[20];
   struct Date doj;
}emp1;
```

2) Embedded structure

```
struct Employee
{
   int id;
   char name[20];
   struct Date
    {
      int dd;
      int mm;
      int yyyy;
    }doj;
}emp1;
```

Nested Structures

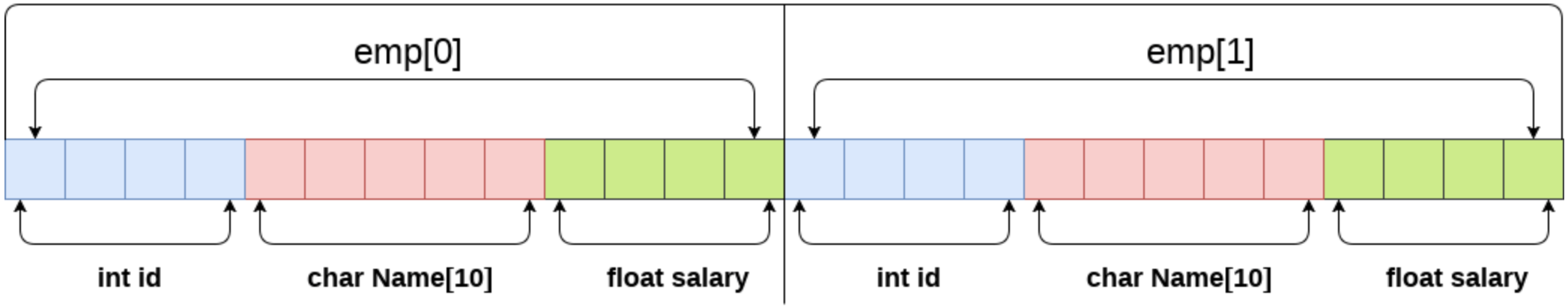
```
#include <stdio.h>
struct complex
{
        int imag;
        float real;
};
struct number
{
         struct complex comp;
         int integer;
} num1;
void main()
 {
     num1.comp.imag = 11;          // initialize complex variables
     num1.comp.real = 5.25;
     num1.integer = 6;                  // initialize number variable

    printf("Imaginary Part: %d\n", num1.comp.imag);
    printf("Real Part: %.2f\n", num1.comp.real);
    printf("Integer: %d", num1.integer);
}
```

Array of Structures in C

1. An array of structures in C can be defined as the collection of multiple structures variables where each variable contains information about different entities.

2. The array of structures in C are used to store information about multiple entities of different data types. The array of structures is also known as the collection of structures.

Array of structures




```
struct employee
{
    int id;
    char name[5];
    float salary;
};
struct employee emp[2];
```

sizeof (emp) = 4 + 5 + 4 = 13 bytes

sizeof (emp[2]) = 26 bytes

Example: Array of Structure

```
#include<stdio.h>
#include <string.h>
struct student
{
int rollno;
char name[10];
};
void main()
{
    int i;
    struct student st[5];
    for(i=0;i<5;i++)
    {
      printf("\nEnter Rollno:");
      scanf("%d",&st[i].rollno);
      printf("\nEnter Name:");
      scanf("%s",&st[i].name);
    }
    for(i=0;i<5;i++)
    {
      printf("\nRollno:%d, Name:%s",st[i].rollno,st[i].name);
    }
}
```

Why structs in C?

1. Suppose you want to store information about a person: his/her name, citizenship number, and salary. You can create different variables name, citNo and salary to store this information.

2. What if you need to store information of more than one person? Now, you need to create different variables for each information per person: name1, citNo1, salary1, name2, citNo2, salary2, etc.

3. A better approach would be to have a collection of all related information under a single name Person structure and use it for every person.

Example: Access members using Pointer

```
#include <stdio.h>
struct person
{
   int age;
   float weight;
};
void main()
{
    struct person *personPtr, person1;
    personPtr = &person1;

    printf("Enter age: ");
    scanf("%d", &personPtr->age);
    printf("Enter weight: ");
    scanf("%f", &personPtr->weight);

    printf("Age: %d\n", personPtr->age);
    printf("weight: %f", personPtr->weight);
}
```

Passing structs to functions

```
#include <stdio.h>
struct student
{
   int age;
};
void display(struct student s);
void main()
{
   struct student s1;
   printf("Enter age: ");
   scanf("%d", &s1.age);
   display(s1); // passing struct as an argument
}
void display(struct student s)
 {
     printf("\nAge: %d", s.age);
}
```

C Unions

A union is a user-defined type similar to structs in C except for one key difference.

Structures allocate enough space to store all their members, whereas **unions can only hold one member value at a time**.

How to define a union?

```
union car
{
  char name[50];
  int price;
};
```

Create union variables

```
union car
{
  char name[50];
  int price;
};

void main()
{
  union car car1, car2, *car3;
}
```

Example: Accessing Union Members

```
#include <stdio.h>
union Job
{
   float salary;
   int workerNo;
} j;

void main()
{
   j.salary = 12.3;          // when j.workerNo is assigned a value,
   j.workerNo = 100;     // j.salary will no longer hold 12.3

   printf("Salary = %.1f\n", j.salary);
   printf("Number of workers = %d", j.workerNo);
}
```

Output

Salary = 0.0
Number of workers = 100

Difference between unions and structures

```
#include <stdio.h>
union unionJob
{
   char name[32];
   float salary;
   int workerNo;
} uJob;

struct structJob
{
   char name[32];
   float salary;
   int workerNo;
} sJob;

void main()
{
   printf("size of union = %d bytes", sizeof(uJob));
   printf("\nsize of structure = %d bytes", sizeof(sJob));
}
```

Output

size of union = 32
size of structure = 40

Why this difference in the size of union and structure variables?

Here, the size of sJob is 40 bytes because

1. the size of name[32] is 32 bytes
2. the size of salary is 4 bytes
3. the size of workerNo is 4 bytes

However, the size of uJob is 32 bytes.

1. It's because the size of a union variable will always be the size of its largest element.
2. the size of its largest element, (name[32]), is 32 bytes.
3. With a union, all members share the same memory.

Exercise

1. Write a program in C to find the maximum number between two numbers using a pointer.
2. Write a program in C swap elements using call by reference.
3. Write a program in C to sort an array using a pointer.
4. Write a program in C to find the factorial of a given number using pointers.
5. Programs given in the presentation
6. Write a program in C to Add two distances (in inch-feet).
7. Write a program in C to Store information of a student using structure
8. Write a program in C to Store information of 10 Employees using structures
9. Programs given in the presentation

Theory Questions

1. Define pointer with example
2. Define the following terms: a) Reference operator b)Dereference operator
3. Explain Arrays of pointer with example program
4. Explain pointer to function with example program
5. State the use of following dynamic memory allocation function:
   a) malloc() b)calloc() c)free() d)realloc()
6. Define Structure and union
7. Write the syntax of Structure declaration and Union Declaration
8. Explain nested structure with example program
9. Explain Arrays of Structure with example program
10. Explain structure to function with example program
11. Differentiate between Structure and Union

THANK YOU !!!!

SHRI VILE PARLE KELAVANI MANDAL'S

SHRI BHAGUBHAI MAFATLAL POLYTECHNIC

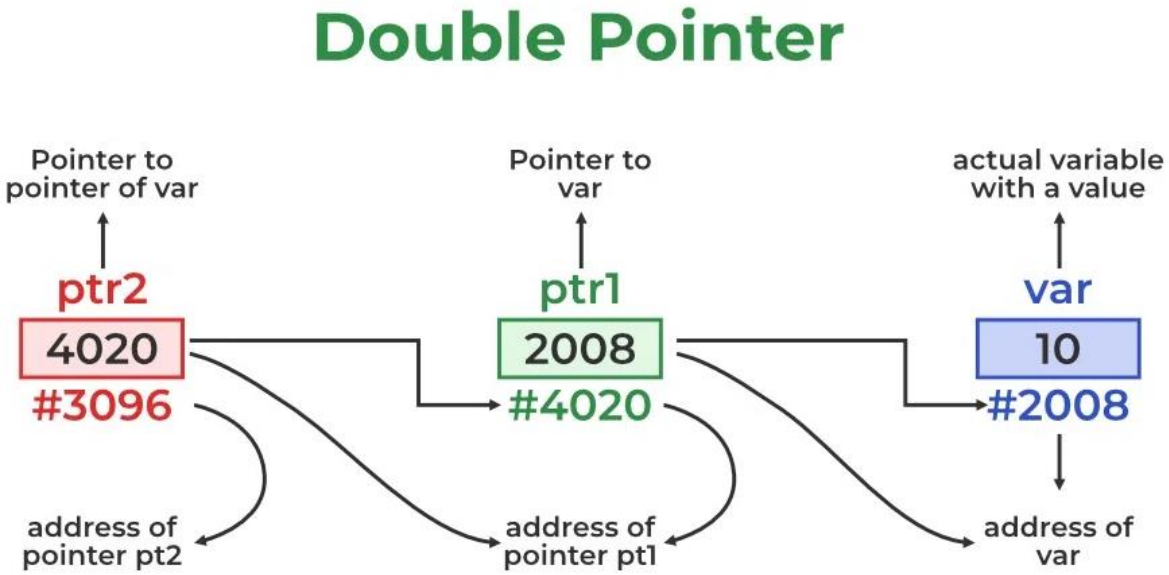
Programme: Computer Engineering **Semester: II**

Course: Programming in C **Course code: PRC238912**

Pointer to Pointer (Double Pointer)

The pointer to a pointer in C is used when we want to store the address of another pointer. The first pointer is used to store the address of the variable. And the second pointer is used to store the address of the first pointer. That is why they are also known as **double-pointers**.

We can use a pointer to a pointer to change the values of normal pointers or create a variable-sized 2-D array. A double pointer occupies the same amount of space in the memory stack as a normal pointer.



Declaration of Pointer to a Pointer in C

Declaring Pointer to Pointer is similar to declaring a pointer in C. The difference is we have to place an additional '*' before the name of the pointer.

data_type_of_pointer **name_of_variable = & normal_pointer_variable;

```
int val = 5;
int *ptr = &val;
int **d_ptr = &ptr;
```

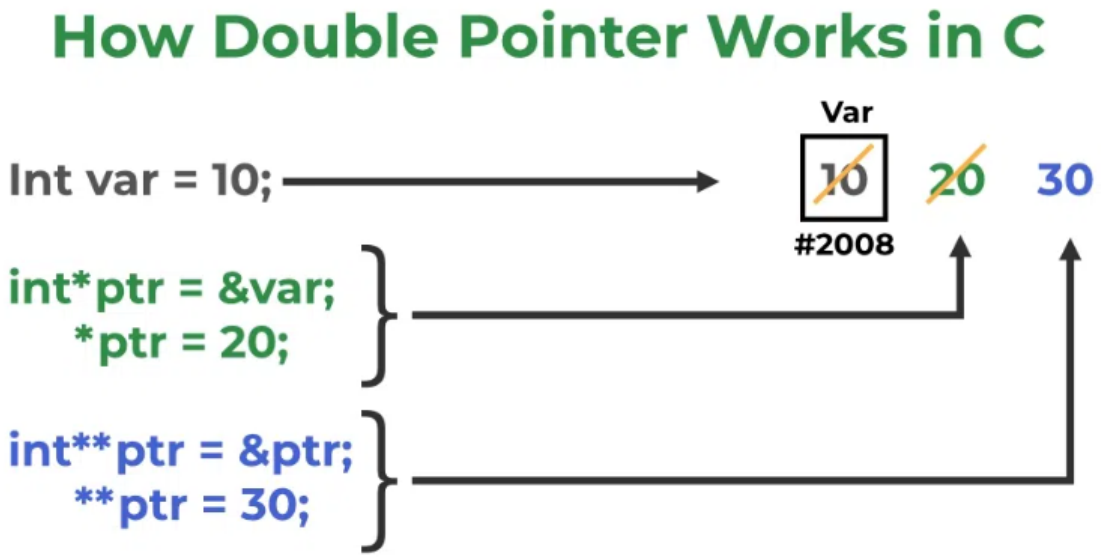
Example of Double Pointer in C

```
#include <stdio.h>
 void main()
{
    int var = 789;

    int* ptr2;

    int** ptr1;

    ptr2 = &var;

    ptr1 = &ptr2;

    printf("Value of var = %d\n", var);
    printf("Value of var using single pointer = %d\n", *ptr2);
    printf("Value of var using double pointer = %d\n", **ptr1);


}
```

Output

Value of var = 789
Value of var using single pointer = 789
Value of var using double pointer = 789

How Double Pointer Works?



The working of the double-pointer can be explained using the above image:

- The double pointer is declared using the syntax shown above.
- After that, we store the address of another pointer as the value of this new double pointer.
- Now, if we want to manipulate or dereference to any of its levels, we have to use Asterisk (*) operator the number of times down the level we want to go.

Application of Double Pointers in C

Following are the main uses of pointer to pointers in C:

- They are used in the dynamic memory allocation of multidimensional arrays.
- They can be used to store multilevel data such as the text document paragraph, sentences, and word semantics.
- They are used in data structures to directly manipulate the address of the nodes without copying.
- They can be used as function arguments to manipulate the address stored in the local pointer.