<u>B.C.A study</u>

unit-1: introduction to ‘c’ language

1. What is C?

- C is a programming language developed at AT & T’s Bell Laboratories of USA in 1972 by Dennis Ritchie.
- Any programming Language can be divided in to two categories.
    - Problem oriented (High level language)
    - Machine oriented (Low level language)

But C is considered as a Middle level Language.

- C is modular, portable, reusable.

2. Feature of C Program

- **Structured language**
    - It has the ability to divide and hide all the information and instruction.
    - Code can be partitioned in C using functions or code block.
    - C is a well structured language compare to other.
- **General purpose language**
    - Make it ideal language for system programming.
    - It can also be used for business and scientific application.
    - ANSI established a standard for c in 1983.
    - The ability of c is to manipulate bits,byte and addresses.
    - It is adopted in later 1990.
- **Portability**
    - Portability is the ability to port or use the software written .
    - One computer C program can be reused.
    - By modification or no modification.
- **Code Re-usability & Ability to customize and extend**
    - A programmer can easily create his own function

    - It can can be used repeatedly in different application
    - C program basically collection of function
    - The function are supported by ‘c’ library
    - Function can be added to ‘c’ library continuously
- **Limited Number of Key Word**
    - There are only 32 keywords in ‘C’
    - 27 keywords are given by ritchie
    - 5 is added by ANSI
    - The strength of ‘C’ is lies in its in-built function
    - Unix system provides as large number of C function
    - Some function are used in operation .
    - Other are for specialized in their application

3. C program structure

```
pre-processor directives

global declarations
main()
{
   local variable deceleration
   statement sequences
   function invoking
}
```

4. C Keywords

Keywords are the words whose meaning has already been explained to the C compiler. There are only 32 keywords available in C. The keywords are also called ‘Reserved words’.

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned

continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

5. C Character Set

A character denotes any alphabet, digit or special symbol used to represent information. Following are the valid alphabets, numbers and special symbols allowed in C.

- Alphabets – A, B, ….., Y, Z a, b, ……, y, z
- Digits – 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
- Special symbols – ~ ‘ ! @ # % ^ & * ( ) _ – + = | \ { }
  [ ] : ; ” ‘ < > , . ? /

6. Rules for Writing, Compiling and Executing the C program

- C is case sensitive means variable named “COUNTER” is different from a variable named “counter”.
- All keywords are lowercased.
- Keywords cannot be used for any other purpose (like variable names).
- Every C statement must end with a ;. Thus ;acts as a statement terminator.
- First character must be an alphabet or underscore, no special symbol other than an underscore, no commas or blank spaces are allowed with in a variable, constant or keyword.
- Blank spaces may be inserted between two words to improve the readability of the statement. However, no blank spaces are allowed within a variable, constant or keyword.
- Variable must be declared before it is used in the program.
- File should be have the extension .c
- Program need to be compiled before execution.

7. Data types & Placeholders

- C has 5 basic built-in data types.
- Data type defines a set of values that a variable can store along with a set of operations that can be performed on it.
- A variable takes different values at different times.
- General form for declaring a variable is:
  type name;
- An example for using variables comes below:

```
#include<stdio.h>

main()
```

```
{
   int sum;
   sum=12;
   sum=sum+5;
   printf("Sum is %d",sum);
}
```

printf function will print the following:
Sum is 17
In fact %d is the placeholder for integer variable value that its name comes after double quotes.

- Common data types are:
  - int – integer
  - char – character
  - long – long integer
  - float – float number
  - double – long float
- Other placeholders are:

Placeholders	Format
%c	Character
%d	Signed decimal integer
%i	Signed decimal integer
%e	Scientific notation[e]
%E	Scientific notation[E]
%f	Decimal floating point
%o	unsigned octal
%s	String of character
%u	unsigned decimal integer
%x	unsigned Hexadecimal (lower)
%X	unsigned Hexadecimal (upper)
%p	dispaly a pointer
%%	print a %

8. Control characters (Escape sequences)

Certain non printing characters as well as the backslash () and the apostrophe(‘), can be expressed in terms of escape sequence.

- \a – Bell
- \n – New line
- \r – Carriage return
- \b – Backspace
- \f – Formfeed
- \t – Horizontal tab
- \” – Quotation mark
- \v – Vertical tab
- \’ – Apostrophe
- \\ – Backslash
- \? – Question mark
- \0 – Null

9. Receiving input values from keyboard

scanf function used to receiving input from keyboard.
General form of scanf function is :

scanf(“Format string”,&variable,&variable,…);

Format string contains placeholders for variables that we intend to receive from keyboard.
A & sign comes before each variable name that comes in variable listing. Character strings are exceptions from this rule. They will not come with this sign before them.

Note: You are not allowed to insert any additional characters in format string other than placeholders and some special characters. Entering even a space or other undesired character will cause your program to work incorrectly and the results will be unexpected. So make sure you just insert placeholder characters in scanf format string. The following example receives multiple variables from keyboard.

float a;

int n;

scanf(“%d%f”,&n,&a);

B.C.A study

unit-2:expression and Operators Precedence – C Programming

DESCRIPTION	OPERATORS	ASSOCIATIVITY
Function Expression	()	Left to Right
Array Expression	[]	Left to Right
Structure Operator	->	Left to Right
Structure Operator	.	Left to Right
Unary minus	–	Right to Left
Increment/Decrement	++, —	Right to Left
One’s compliment	~	Right to Left
Negation	!	Right to Left
Address of	&	Right to Left
Value of address	`*`	Right to Left
Type cast	(type)	Right to Left
Size in bytes	sizeof	Right to Left
Multiplication	`*`	Left to Right
Division	/	Left to Right
Modulus	%	Left to Right
Addition	+	Left to Right
Subtraction	–	Left to Right
Left shift	<<	Left to Right
Right shift	>>	Left to Right
Less than	<	Left to Right
Less than or equal to	<=	Left to Right
Greater than	>	Left to Right

Greater than or equal to	>=	Left to Right
Equal to	==	Left to Right
Not equal to	!=	Left to Right
Bitwise AND	&	Left to Right
Bitwise exclusive OR	^	Left to Right
Bitwise inclusive OR	|	Left to Right
Logical AND	&&	Left to Right
Logical OR	||	Left to Right
Conditional	?:	Right to Left
Assignment	=, *=, /=, %=, +=, -=, &=, ^=, |=, <<=, >>=	Right to Left
Comma	,	Right to Left

Console based I/O and related I/-

Input means to provide the program with some data to be used in the program and **Output** means to display data on screen or write the data to a printer or a file.

C programming language provides many built-in functions to read any given input and to display data on screen when there is a need to output the result.

In this tutorial, we will learn about such functions, which can be used in our program to take input from user and to output the result on screen.

All these built-in functions are present in C header files, we will also specify the name of header files in which a particular function is defined while discussing about it.

`scanf()` and `printf()` functions

The standard input-output header file, named `stdio.h` contains the definition of the functions `printf()` and `scanf()`, which are used to display output on screen and to take input from user respectively.

```c
#include<stdio.h>

void main()

{

    // defining a variable

    int i;

      /*

        displaying message on the screen

        asking the user to input a value

    */

    printf("Please enter a value...");
      /*

        reading the value entered by the user

    */

    scanf("%d", &i);

      /*

        displaying the number as output

    */

    printf( "\nYou entered: %d", i);

}
```

When you will compile the above code, it will ask you to enter a value. When you will enter the value, it will display the value you have entered on screen.

You must be wondering what is the purpose of %d inside the `scanf()` or `printf()` functions. It is known as **format string** and this informs the `scanf()` function, what type of input to expect and in `printf()` it is used to give a heads up to the compiler, what type of output to expect.

Format String	**Meaning**
%d	Scan or print an integer as signed decimal number

`%f`	Scan or print a floating point number
`%c`	To scan or print a character
`%s`	To scan or print a character string. The scanning ends at whitespace.

We can also **limit the number of digits or characters** that can be input or output, by adding a number with the format string specifier, like `"%1d"` or `"%3s"`, the first one means a single numeric digit and the second one means 3 characters, hence if you try to input `42`, while `scanf()` has `"%1d"`, it will take only 4 as input. Same is the case for output.

In C Language, computer monitor, printer etc output devices are treated as files and the same process is followed to write output to these devices as would have been followed to write the output to a file.

NOTE : `printf()` function returns the number of characters printed by it, and `scanf()` returns the number of characters read by it.

```
int i = printf("studytonight");
```

In this program `printf("studytonight");` will return `12` as result, which will be stored in the variable `i`, because studytonight has 12 characters.

`getchar()` & `putchar()` functions

The `getchar()` function reads a character from the terminal and returns it as an integer. This function reads only single character at a time. You can use this method in a loop in case you want to read more than one character. The `putchar()` function displays the character passed to it on the screen and returns the same character. This function too displays only a single character at a time. In case you want to display more than one characters, use `putchar()` method in a loop.

```
#include < stdio . h >


void main ( )
```

```
{

    int c ;

     printf ( "Enter a character" );

     /*

         Take a character as input and

         store it in variable c

    */

    c  = getchar ();

     /*

         display the character stored
```

```
        in variable c

    */

     putchar ( c );

 }
```

When you will compile the above code, it will ask you to enter a value. When you will enter the value, it will display the value you have entered.

gets() & puts() functions

The `gets()` function reads a line from **stdin**(standard input) into the buffer pointed to by `str` **pointer (https://www.studytonight.com/pointers-in-c.php)**, until either a terminating newline or EOF (end of file) occurs. The `puts()` function writes the string `str` and a trailing newline to **stdout**.

`str` → This is the pointer to an array of chars where the C string is stored. (Ignore if you are not able to understand this now.)

```
#include < stdio . h >


void  main ()
```

```
{

     /* character array of length 100 */

    char str [ 100 ];

     printf ( "Enter a string" );

     gets (  str  );

     puts (  str  );

     getch ();

}
```

When you will compile the above code, it will ask you to enter a string. When you will enter the string, it will display the value you have entered.

Difference between `scanf()` and `gets()`

The main difference between these two functions is that `scanf()` stops reading characters when it encounters a space, but `gets()` reads space as character too.

If you enter name as **Study Tonight** using `scanf()` it will only read and store **Study** and will leave the part after space. But `gets()` function will read it completely.

Header Files in C

Header files contain definitions of functions and variables, which is imported or used into any C program by using the pre-processor #include statement. Header file have an extension “.h” which contains C function declaration and macro definition.

header files in c

Each header file contains information (or declarations) for a particular group of functions. Like **stdio.h** header file contains declarations of standard input and output functions available in C which is used for get the input and print the output. Similarly, the header file **math.h** contains declarations of mathematical functions available in C.

Types of Header Files in C

- **System Header Files:** It is comes with compiler.
- **User header files:** It is written by programmer.

Why need of header files

When we want to use any function in our c program then first we need to import their definition from c library, for importing their declaration and definition we need to include header file in program by using #include. Header file include at the top of any C program.

For example if we use printf() in C program, then we need to include, stdio.h header file, because in stdio.h header file definition of printf() (for print message on screen) is written in stdio.h header file.

Syntax Header Files in C

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

How to use Header File in Program

Both user and system header files are include using the pre-processing directive #include. It has following two forms:

Syntax

```
#include<file>
```

This form is used for system header files. It searches for a file named file in a standard list of system directives.

Preprocessor Directives –

The C preprocessor is a macro processor that is used automatically by the C compiler to transform your program before actual compilation (Proprocessor direcives are executed before compilation.). It is called a macro processor because it allows you to define macros, which are brief abbreviations for longer constructs. A macro is a segment of code which is replaced by the value of macro. Macro is defined by `#define` directive.

Preprocessing directives are lines in your program that start with #. The # is followed by an identifier that is the directive name. For example, `#define` is the directive that defines a macro. Whitespace is also allowed before and after the #.

The # and the directive name cannot come from a macro expansion. For example, if `foo` is defined as a macro expanding to `define`, that does not make `#foo` a valid preprocessing directive.

All preprocessor directives starts with hash # symbol.

List of preprocessor directives :

1. `#include`
2. `#define`
3. `#undef`
4. `#ifdef`
5. `#ifndef`
6. `#if`
7. `#else`
8. `#elif`
9. `#endif`
10. `#error`
11. `#pragma`

1. #include

The #include preprocessor directive is used to paste code of given file into current file. It is used include system-defined and user-defined header files. If included file is not found, compiler renders error. It has three variants:

`#include <file>`

This variant is used for system header files. It searches for a file named file in a list of directories specified by you, then in a standard list of system directories.

`#include "file"`

This variant is used for header files of your own program. It searches for a file named file first in the current directory, then in the same directories used for system header files. The current directory is the directory of the current input file.

```
#include anything else
```

This variant is called a computed `#include`. Any `#include` directive whose argument does not fit the above two forms is a computed include

2. Macro's (#define)

Let's start with macro, as we discuss, a macro is a segment of code which is replaced by the value of macro. Macro is defined by `#define` directive.

Syntax

```
#define token value
```

There are two types of macros:

1. Object-like Macros
2. Function-like Macros

1. Object-like Macros

The object-like macro is an identifier that is replaced by value. It is widely used to represent numeric constants. For example:

```
#define PI 3.1415
```

Here, PI is the macro name which will be replaced by the value 3.14. Let's see an example of Object-like Macros :

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

```
#define PI 3.1415

main()

{

   printf("%f",PI);

}
```

Output:

```
3.14000
```

2. Function-like Macros

The function-like macro looks like function call. For example:

```
#define MIN(a,b) ((a)<(b)?(a):(b))
```

Here, MIN is the macro name. Let’s see an example of Function-like Macros :

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

```
#define MIN(a,b) ((a)<(b)?(a):(b))

void main() {

   printf("Minimum between 10 and 20 is: %d\n", MIN(10,20));

}
```

Output:

```
Minimum between 10 and 20 is: 10
```

Unit-3: Control structures

Control structures in C programming allow you to control the flow of execution of your program based on conditions or repetition. There are three main control structures in C:

1. **Conditional (if-else) statements**: These are used to execute a block of code only if a certain condition is met.

Example:

```
int x = 10;
if (x > 5) {
printf(“x is greater than 5”);
}
```

2. **Loops**: These are used to repeatedly execute a block of code until a certain condition is met.

Example:

```
for (int i = 0; i < 10; i++) {
printf(“%d\n”, i);
}
```

3. **Switch statements:** These are used to select one block of code from many to be executed.

Example:

```
char grade = ‘B’;
switch (grade) {
case ‘A’:
printf(“Excellent!”);
break;
case ‘B’:
printf(“Good!”);
break;
case ‘C’:
printf(“Average”);
break;
default:
printf("Invalid grade");
}
```

Decision making structures

Decision making structures in C are used to make decisions in a program based on certain conditions. There are two main decision making structures in C:

The `if` statement in C programming is used to execute a block of code only if a certain condition is met. The general syntax for an `if` statement is as follows:

```
if (condition) {
// code to be executed if condition is true
}
```

The `if` statement in C programming is used to execute a block of code only if a certain condition is met. The general syntax for an `if` statement is as follows:

```
if (condition) {
  // code to be executed if condition is true
}
```

Here, `condition` is a Boolean expression that can be either true or false. If the `condition` is true, the code inside the braces will be executed, and if the `condition` is false, the code inside the braces will be skipped.

Example:

```
int x = 10;
if (x > 5) {
printf("x is greater than 5");
}
```

In this example, the `if` statement checks if `x` is greater than 5. If it is, the message "x is greater than 5" is printed.

1. **if-else statemen**t: The if-else statement allows you to execute a block of code if a certain condition is met and another block of code if the condition is not met.

Example:

```
int x = 10;
if (x > 5) {
printf(“x is greater than 5”);
} else {
printf(“x is not greater than 5”);
}
```

2. **switch statement:** The switch statement allows you to choose one block of code from many to be executed based on the value of an expression.

Example:

```
char grade = ‘B’;
switch (grade) {
case ‘A’:
printf(“Excellent!”);
break;
case ‘B’:
printf(“Good!”);
break;
case ‘C’:
printf(“Average”);
break;
default:
printf(“Invalid grade”);
}
```

Nested If-else

Nested if-else is a construct in C programming where an if-else statement is contained inside another if-else statement. This allows you to test multiple conditions and execute different blocks of code based on those conditions.

Example:

```
int x = 10, y = 20;
if (x > 5) {
if (y > 15) {
printf(“x is greater than 5 and y is greater than 15”);
} else {
printf(“x is greater than 5 and y is not greater than 15”);
}
} else {
printf(“x is not greater than 5”);
}
```

In this example, the outer if statement tests if `x` is greater than 5. If it is, then the inner if statement is executed, which tests if `y` is greater than 15. Depending on the results of these two conditions, different messages are printed.

loop control structures

Loop control structures in C programming are used to repeatedly execute a block of code until a certain condition is met. There are three main loop control structures in C:

1. **`for` loop**: The `for` loop is used to execute a block of code a specific number of times. The general syntax for a `for` loop is:

```
for (initialization; condition; increment) {
// code to be executed
}
```

Here, `initialization` is the initial value for the loop variable, `condition` is the condition that must be met for the loop to continue executing, and `increment` is the step by which the loop variable is incremented on each iteration.

Example:

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

Loop control structures in C programming are used to repeatedly execute a block of code until a certain condition is met. There are three main loop control structures in C:

1. **`for` loop:** The `for` loop is used to execute a block of code a specific number of times. The general syntax for a `for` loop is:

```
cssCopy codefor (initialization; condition; increment) {
  // code to be executed
}
```

Here, `initialization` is the initial value for the loop variable, `condition` is the condition that must be met for the loop to continue executing, and `increment` is the step by which the loop variable is incremented on each iteration.

Example:

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

In this example, the `for` loop will execute 10 times, starting from 0 and ending at 9, printing each value of `i` on a separate line.

2. **`while` loop:** The `while` loop is used to execute a block of code as long as a certain condition is true. The general syntax for a `while` loop is:

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

Here, `condition` is the condition that must be true for the loop to continue executing.

Example:

int i = 0;
while (i < 10) {
printf(“%d\n”, i);
i++;
}

In this example, the `while` loop will execute 10 times, starting from 0 and ending at 9, printing each value of `i` on a separate line.

3. **`do-while` loop:** The `do-while` loop is similar to the `while` loop, but with one important difference: the code inside the loop will always be executed at least once, regardless of the condition. The general syntax for a `do-while` loop is:

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

Here, `condition` is the condition that must be true for the loop to continue executing.

Example:

```
  int i = 0;
do {
  printf("%d\n", i);
  i++;
} while (i < 10);
```

In this example, the `do-while` loop will execute 10 times, starting from 0 and ending at 9, printing each value of `i` on a separate line.

`do-while` loop: The `do-while` loop is similar to the `while` loop, but with one important difference: the code inside the loop will always be executed at least once, regardless of the condition. The general syntax for a `do-while` loop is:

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

Here, `condition` is the condition that must be true for the loop to continue executing.

Example:

```
int i = 0;
do {
printf("%d\n", i);
i++;
} while (i < 10);
```

In this example, the `do-while` loop will execute 10 times, starting from 0 and ending at 9, printing each value of `i` on a separate line

Nested for loop

A nested for loop is a loop inside another loop. It allows you to run multiple iterations of the inner loop for each iteration of the outer loop. Here is an example in C:

```
include<studio.h>

int main() {
int i, j;

for (i = 1; i <= 3; i++) {
for (j = 1; j <= 2; j++) {
printf("i = %d, j = %d\n", i, j);
```

}
}

return 0;
}

This code will output:

```
i = 1, j = 1
i = 1, j = 2
i = 2, j = 1
i = 2, j = 2
i = 3, j = 1
i = 3, j = 2
```

break, continue, goto, exit

Here’s a brief explanation of `break`, `continue`, `goto`, and `exit` in C programming language with examples:

1. **`break`:** The `break` statement is used to exit a loop prematurely. It can be used in `for`, `while`, `do-while` loops. Here’s an example:

```
#include <stdio.h>

int main() {
  int i;

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

  return 0;
}
```

This code will output:

```
Copy code0
1
2
3
4
```

2. **`continue`**: The `continue` statement is used to skip the current iteration of a loop and move on to the next iteration. It can be used in `for`, `while`, `do-while` loops. Here's an example:

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

int main() {
  int i;

  for (i = 0; i < 10; i++) {
    if (i % 2 == 0) {
      continue;
    }
    printf("%d\n", i);
  }

  return 0;
}
```

This code will output:

```
Copy code1
3
5
7
9
```

3. **`goto`:** The `goto` statement is used to jump to a labeled statement in the program. It can be used to jump out of a deeply nested loop or to jump to a specific section of code. Here's an example:

```
#include <stdio.h>

int main() {
  int i;

  for (i = 0; i < 10; i++) {
    if (i == 5) {
      goto end;
    }
    printf("%d\n", i);
  }

  end:
    printf("Reached end of program.\n");

  return 0;
}
```

This code will output:

```
0
1
2
3
4
Reached end of program.
```

4. **`exit`:** The `exit` function is used to immediately exit a program and return a status value. Here's an example:

```
#include<studio.h>

int main() {
int i;

for (i = 0; i < 10; i++) {
if (i == 5) {
exit(0);
}
printf("%d\n", i);
}

return 0;
}
```

This code will output:

```
0
1
2
3
4
```

•••

Unit-4:Introduction to problem solving Concept

Problem solving is a critical skill in computer science and programming. It refers to the process of finding solutions to problems or challenges by applying logic and critical thinking.

Here are some key concepts in problem solving:

1. Understanding the problem: This involves carefully reading and comprehending the problem statement and defining the problem in your own words.
2. Analyzing the problem: This involves breaking down the problem into smaller, more manageable parts and identifying the information and data required to solve the problem.
3. Formulating a plan: This involves creating a step-by-step plan for solving the problem, including the methods and algorithms that will be used.
4. Implementing the plan: This involves coding the solution and testing it to ensure that it works as expected.
5. Evaluating the solution: This involves analyzing the solution to ensure that it's correct, efficient, and meets the requirements of the problem.

It's important to note that problem solving is an iterative process and may require multiple iterations of the above steps. The goal is to find a solution that works and meets the requirements of the problem. Effective problem solving skills require patience, persistence, and a willingness to try different approaches until the right solution is found

Problem solving in programming requires critical thinking, creativity, and a deep understanding of the programming concepts and algorithms. It's important to be able to identify patterns and use abstraction and decomposition to break down complex problems into simpler parts. Effective problem solving skills also require patience, persistence, and a willingness to try different approaches until the right solution is found.

Problem solving techniques

There are several techniques that can be used to solve problems in programming. Here are some of the most common problem solving techniques along with examples:

1. Brute Force: This involves trying all possible combinations or solutions to find the correct answer. For example, you can use brute force to solve a problem by trying every possible password combination until the correct one is found.
2. Divide and Conquer: This involves breaking down a complex problem into smaller, more manageable sub-problems and solving each sub-problem individually. For example, you can use divide and conquer to solve a problem by breaking down a large data set into smaller chunks, sorting each chunk individually, and then merging the sorted chunks back together.
3. Greedy Algorithm: This involves making the best choice at each step, with the hope of finding an optimal solution. For example, you can use a greedy algorithm to solve a problem by selecting the highest-value item at each step until you have a complete solution.
4. Backtracking: This involves trying out different solutions and undoing the steps if they lead to an incorrect solution. For example, you can use backtracking to solve a problem by trying out different combinations of numbers and undoing the steps if they don't lead to the correct solution.
5. Dynamic Programming: This involves breaking down a problem into sub-problems and storing the solutions to those sub-problems in a table for later reuse. For example, you can use dynamic programming to solve a problem by breaking down a large data set into smaller chunks and storing the solutions to each chunk in a table for later reuse.
6. Recursion: This involves breaking down a problem into smaller sub-problems and solving each sub-problem recursively. For example, you can use recursion to solve a problem by breaking down a large data set into smaller chunks and solving each chunk recursively until you have the final solution.

It's important to note that different problems may require different problem solving techniques, and that a single problem may have multiple solutions using different techniques. Effective problem solving skills require being able to identify the right technique for the problem at hand and using it to find the optimal solution.

trial & error

Trial and error is a problem solving technique in programming where you test various solutions to a problem and evaluate their results to determine the best solution. This technique involves trying out different solutions and evaluating their results until you find the correct solution.

Here's an example of trial and error in programming:

Suppose you have to write a program to find the square root of a number. You can start by trying out different solutions using the trial and error technique. For example, you can try dividing the number by 2, then by 3, then by 4, and so on, until you find a number that, when multiplied by itself, is close to the original number. You can then use this number as an approximation of the square root.

With trial and error, you can iteratively test different solutions until you find the correct one. This technique is often used when there is no clear or known solution to a problem. It's important to note that trial and error can be time-consuming and may not always produce the most efficient solution, but it can still be an effective technique for finding a solution to a complex problem

brainstorming

Brainstorming is a problem solving technique in programming where a group of people generate a large number of ideas or solutions to a problem. This technique involves generating as many ideas as possible in a short period of time without evaluating them. The goal is to generate a large number of ideas that can later be evaluated and refined.

Here’s an example of brainstorming in programming:

Suppose you have to write a program to manage a library’s inventory. You and your team decide to use brainstorming to generate as many ideas as possible for the program’s features. During the brainstorming session, you and your team generate a large number of ideas, such as:

- A search function to find books by author, title, or ISBN
- A check-in/check-out function to keep track of which books are checked out
- A notification system to remind users when a book they have checked out is due
- A rating system to allow users to rate books they have read
- A recommendation system to suggest books to users based on their reading history

After the brainstorming session, you and your team evaluate the ideas generated and refine them to come up with the final solution for the program. Brainstorming can be an effective technique for generating a large number of ideas and can be especially useful in a team setting where multiple perspectives can be brought to bear on a problem

Divide and Conquer

Divide and Conquer is a problem solving technique that involves breaking down a complex problem into smaller, more manageable sub-problems, and then solving each sub-problem individually. This technique is often used to solve problems that are too complex to be solved in one step, or to find solutions to problems where a brute force approach is too time-consuming or infeasible.

Here’s an example of divide and conquer in programming:

Suppose you have to sort a large array of numbers. One way to sort the array is to use a divide and conquer approach by dividing the array into smaller sub-arrays, sorting each sub-array individually, and then merging the sorted sub-arrays back together to form the final sorted array.

The divide and conquer approach can be implemented using a sorting algorithm such as merge sort, where the array is recursively divided into smaller sub-arrays until each sub-array contains only one element, at which point the sub-arrays are merged back together in sorted order. This approach can be more efficient than a brute force approach, as it allows you to solve the problem in smaller, more manageable steps.

Divide and conquer can be a useful technique for solving complex problems, as it allows you to break down the problem into smaller, more manageable sub-problems that can be solved individually. By solving each sub-problem separately, you can find solutions to the larger problem that would otherwise be difficult or impossible to find

Steps in problem solving

1. Define the problem: Clearly identify and understand the problem that needs to be solved.
2. Gather information: Collect data and information related to the problem.
3. Develop potential solutions: Generate multiple possible solutions to the problem.
4. Evaluate potential solutions: Assess each solution based on its potential effectiveness, feasibility, and impact.
5. Select a solution: Choose the best solution based on the evaluation.
6. Implement the solution: Put the chosen solution into action.
7. Monitor progress: Continuously monitor and evaluate the solution to ensure it is solving the problem effectively.
8. Refine the solution: Make necessary adjustments to the solution if it is not working as intended.

Algorithms and Flowcharts

The **algorithm and flowchart** are two types of tools to explain the process of a program. In this page, we discuss the differences between an algorithm and a flowchart and how to create a flowchart to illustrate the algorithm visually. **Algorithms and flowcharts** are two different tools that are helpful for creating new programs, especially in computer programming. An algorithm is a step-by-step analysis of the process, while a flowchart explains the steps of a program in a graphical way.

Definition of Algorithm

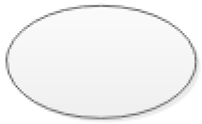
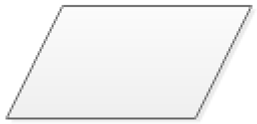
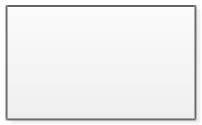
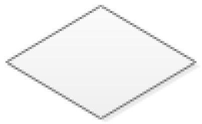
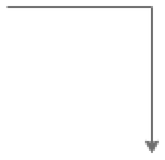
Writing a logical step-by-step method to solve the problem is called the **algorithm (https://www.edrawsoft.com/algorithm-definition.html)**. In other words, an algorithm is a procedure for solving problems. In order to solve a mathematical or computer problem, this is the first step in the process.

An algorithm includes calculations, reasoning, and data processing. Algorithms can be presented by natural languages, pseudocode, and flowcharts, etc.

Definition of Flowchart

A flowchart is the graphical or pictorial representation of an algorithm with the help of different symbols, shapes, and arrows to demonstrate a process or a program. With algorithms, we can easily understand a program. The main purpose of using a flowchart is to analyze different methods. Several standard symbols are applied in a flowchart:

Common Abbreviations Used in P&ID

Terminal Box – Start / End	
Input / Output	
Process / Instruction	
Decision	
Connector / Arrow	

The symbols above represent different parts of a flowchart. The process in a flowchart can be expressed through boxes and arrows with different sizes and colors. In a flowchart, we can easily highlight certain elements and the relationships between each part.

Difference between Algorithm and Flowchart

If you compare a flowchart to a movie, then an algorithm is the story of that movie. In other words, **an algorithm is the core of a flowchart**. Actually, in the field of computer programming, there are many differences between algorithm and flowchart regarding various aspects, such as the accuracy, the way they display, and the way people feel about them. Below is a table illustrating the differences between them in detail.Algorithm

ALGORITHM

- It is a procedure for solving problems.
- The process is shown in step-by-step instruction.
- It is complex and difficult to understand.
- It is convenient to debug errors.
- The solution is showcased in natural language.
- It is somewhat easier to solve complex problem.
- It costs more time to create an algorithm.

Flowchart

- It is a graphic representation of a process.
- The process is shown in block-by-block information diagram.
- It is intuitive and easy to understand.
- It is hard to debug errors.
- The solution is showcased in pictorial format.
- It is hard to solve complex problem.
- It costs less time to create a flowchart.

types of algorithm

#1 Recursive Algorithm

It refers to a way to solve problems by repeatedly breaking down the problem into sub-problems of the same kind. The classic example of using a recursive algorithm to solve problems is the Tower of Hanoi.

#2 Divide and Conquer Algorithm

Traditionally, the divide and conquer algorithm consists of two parts: 1. breaking down a problem into some smaller independent sub-problems of the same type; 2. finding the final solution of the original issues after solving these more minor problems separately. The key points of the divide and conquer algorithm are:

- If you can find the repeated sub-problems and the loop substructure of the original problem, you may quickly turn the original problem into a small, simple issue.
- Try to break down the whole solution into various steps (different steps need different solutions) to make the process easier.
- Are sub-problems easy to solve? If not, the original problem may cost lots of time.

#3 Dynamic Programming Algorithm

Developed by Richard Bellman in the 1950s, the dynamic programming algorithm is generally used for optimization problems. In this type of algorithm, past results are collected for future use. Like the divide and conquer algorithm, a dynamic programming algorithm simplifies a complex problem by breaking it down into some simple sub-problems. However, the most significant difference between them is that the latter requires overlapping sub-problems, while the former doesn't need to.

#4 Greedy Algorithm

This is another way of solving optimization problems – greedy algorithm. It refers to always finding the best solution in every step instead of considering the overall optimality. That is to say, what he has done is just at a local optimum. Due to the limitations of the greedy algorithm, it has to be noted that the key to choosing a greedy algorithm is whether to consider any consequences in the future.

#5 Brute Force Algorithm

The brute force algorithm is a simple and straightforward solution to the problem, generally based on the description of the problem and the definition of the concept involved. You can also use "just do it!" to describe the strategy of brute force. In short, a brute force algorithm is considered as one of the simplest algorithms, which iterates all possibilities and ends up with a satisfactory solution.

#6 Backtracking Algorithm

Based on a depth-first recursive search, the backtracking algorithm focusing on finding the solution to the problem during the enumeration-like searching process. When it cannot satisfy the condition, it will return "backtracking" and tries another path. It is suitable for solving large and complicated problems, which gains the reputation of the "general solution method". One of the most famous backtracking algorithm example it the eight queens puzzle.

Use Flowcharts to Represent Algorithms

#1 Recursive Algorithm

It refers to a way to solve problems by repeatedly breaking down the problem into sub-problems of the same kind. The classic example of using a recursive algorithm to solve problems is the Tower of Hanoi.

#2 Divide and Conquer Algorithm

Traditionally, the divide and conquer algorithm consists of two parts: 1. breaking down a problem into some smaller independent sub-problems of the same type; 2. finding the final solution of the original issues after solving these more minor problems separately. The key points of the divide and conquer algorithm are:

- If you can find the repeated sub-problems and the loop substructure of the original problem, you may quickly turn the original problem into a small, simple issue.
- Try to break down the whole solution into various steps (different steps need different solutions) to make the process easier.
- Are sub-problems easy to solve? If not, the original problem may cost lots of time.

#3 Dynamic Programming Algorithm

Developed by Richard Bellman in the 1950s, the dynamic programming algorithm is generally used for optimization problems. In this type of algorithm, past results are collected for future use. Like the divide and conquer algorithm, a dynamic programming algorithm simplifies a complex problem by breaking it down into some simple sub-problems. However, the most significant difference between them is that the latter requires overlapping sub-problems, while the former doesn’t need to.

#4 Greedy Algorithm

This is another way of solving optimization problems – greedy algorithm. It refers to always finding the best solution in every step instead of considering the overall optimality. That is to say, what he has done is just at a local optimum. Due to the limitations of the greedy algorithm, it has to be noted that the key to choosing a greedy algorithm is whether to consider any consequences in the future.

#5 Brute Force Algorithm

The brute force algorithm is a simple and straightforward solution to the problem, generally based on the description of the problem and the definition of the concept involved. You can also use “just do it!” to describe the strategy of brute force. In short, a brute force algorithm is considered as one of the simplest algorithms, which iterates all possibilities and ends up with a satisfactory solution.

#6 Backtracking Algorithm

Based on a depth-first recursive search, the backtracking algorithm focusing on finding the solution to the problem during the enumeration-like searching process. When it cannot satisfy the condition, it will return “backtracking” and tries another path. It is suitable for solving large and complicated problems, which gains the reputation of the “general solution method”. One of the most famous backtracking algorithm example it the eight queens puzzle.

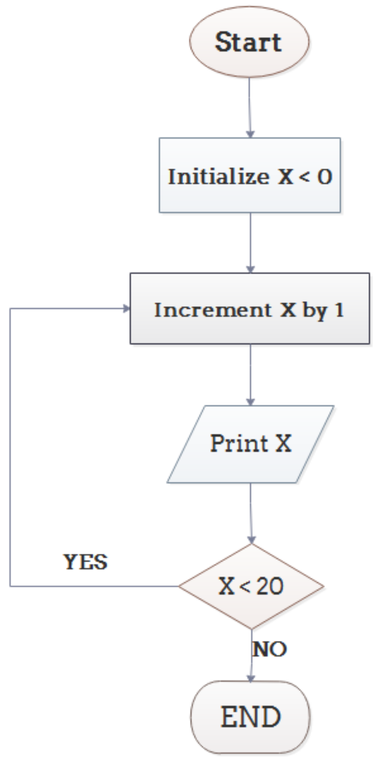
Use Flowcharts to Represent Algorithms

Example 1: Print 1 to 20:

Algorithm:

- Step 1: Initialize X as 0,
- Step 2: Increment X by 1,
- Step 3: Print X,
- Step 4: If X is less than 20 then go back to step 2.

Flowchart:

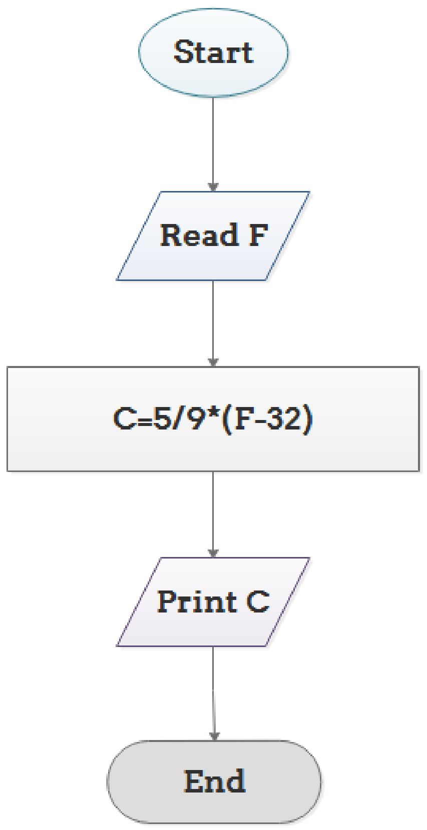


Example 2: Convert Temperature from Fahrenheit (℉) to Celsius (℃)

Algorithm:

- Step 1: Read temperature in Fahrenheit,
- Step 2: Calculate temperature with formula C=5/9*(F-32),
- Step 3: Print C.

Flowchart:



Characteristics of an algorithm

1. Input: An algorithm must have zero or more inputs that define the problem to be solved or the data to be processed. The input can be in the form of values, variables, or any other data structures.
2. Output: An algorithm must have a well-defined output, which can be a single value, multiple values, or a set of values. The output must be related to the problem that the algorithm is trying to solve.
3. Definiteness: An algorithm must have a well-defined set of steps or instructions that are executed in a specific order. These steps must be clear and unambiguous.
4. Finiteness: The algorithm must terminate after a finite number of steps. It must not run indefinitely or get stuck in an infinite loop.
5. Feasibility: An algorithm must be implementable, meaning that it can be translated into a program that can be executed on a computer.
6. Effectiveness: The algorithm must be efficient and solve the problem in a reasonable amount of time and with a reasonable amount of resources, such as memory and computational power.
7. Generality: The algorithm must be able to solve a wide range of problems or process a wide range of data, not just specific cases.

8. Optimality: An algorithm can be optimal if it produces the best possible solution for a given problem, or if it produces the solution in the minimum amount of time or with the minimum amount of resources.

In summary, an algorithm is a set of well-defined, finite, and effective steps or instructions for solving a problem or processing data, which must have inputs and outputs, be feasible and implementable, and have a level of generality and optimality.

Conditionals in pseudo-code

Conditional statements allow the execution of code to be dependent on certain conditions being met. In pseudo-code, conditionals are typically expressed using the keywords “if” and “else”.

The basic syntax of an “if” statement in pseudo-code is as follows:

```
if (condition)
then (action to be taken if condition is true)
```

For example, consider a program that checks whether a number is positive or negative:

```
input x
if (x > 0)
then print “x is positive”
```

If the condition is true, the code inside the “if” statement will be executed. If the condition is false, the code inside the “if” statement will be skipped.

An “if-else” statement allows for two different actions to be taken, depending on whether the condition is true or false. The syntax of an “if-else” statement in pseudo-code is as follows:

```
if (condition)
then (action to be taken if condition is true)
else
then (action to be taken if condition is false)
```

For example, consider a program that checks whether a number is positive, negative, or zero:

```
input x
if (x > 0)
then print “x is positive”
else if (x < 0)
then print “x is negative”
else
then print “x is zero”
```

In this example, if the condition “x > 0” is true, the code inside the first “if” statement will be executed. If the condition “x > 0” is false, the program will move on to the next condition, “x < 0”. If this condition is true, the code inside the second “if” statement will be executed. If both conditions are false, the code inside the “else” statement will be executed.

These are the basic concepts of conditionals in pseudo-code. The use of conditionals is essential in programming for controlling the flow of a program and making decisions based on the input data.

loops in pseudo code

Loops are a powerful programming construct that allow the repetition of a set of instructions multiple times, until a certain condition is met. In pseudo-code, loops are typically expressed using the keywords “for” or “while”.

A “for” loop is used to repeat a set of instructions a specific number of times. The basic syntax of a “for” loop in pseudo-code is as follows:

```
for i = 1 to n
do (action to be repeated n times)
```

For example, consider a program that prints the first 10 positive integers:

```
for i = 1 to 10
do print i
```

In this example, the “for” loop will repeat the instruction “print i” 10 times, and each time the value of “i” will be incremented by 1.

A “while” loop is used to repeat a set of instructions as long as a certain condition is true. The basic syntax of a “while” loop in pseudo-code is as follows:

```
while (condition)
do (action to be repeated while condition is true)
```

For example, consider a program that prints the positive integers until a certain number is reached:

```
input max
i = 1
while (i <= max)
do
print i
i = i + 1
```

In this example, the “while” loop will repeat the instructions “print i” and “i = i + 1” as long as the condition “i <= max” is true. The loop will terminate when “i” is no longer less than or equal to “max”.

These are the basic concepts of loops in pseudo-code. The use of loops is essential in programming for repeating actions, processing large amounts of data, and automating tasks.

Time complexity

Time complexity is a measure of the amount of time an algorithm takes to complete, as a function of the size of the input data. It provides a way to compare the performance of different algorithms and to evaluate the efficiency of a particular algorithm.

The time complexity of an algorithm is typically expressed using big O notation, which provides an upper bound on the number of operations performed by the algorithm as a function of the size of the input. For example, an algorithm with a time complexity of O(n) is said to have a linear time complexity, meaning that the number of operations performed is proportional to the size of the input data.

A common example of a linear time complexity algorithm is a linear search. In a linear search, an algorithm checks each element in an array one by one until it finds the target element. The time complexity of this algorithm is O(n), because the number of operations required to find the target element increases linearly with the size of the array.

Another example is a binary search, which is an algorithm for finding an element in a sorted array. The time complexity of binary search is O(log n), because the number of operations required to find the target element decreases logarithmically with the size of the array. This makes binary search a much faster algorithm than linear search for large arrays.

In summary, time complexity is a critical aspect of algorithm design and analysis. Understanding the time complexity of an algorithm is important for evaluating its performance, comparing it to other algorithms, and optimizing its efficienc

Big-Oh notation

Big-Oh notation, also known as big O notation, is a mathematical notation used to describe the upper bound on the growth rate of the time complexity of an algorithm. It provides a way to compare the performance of different algorithms and to evaluate the efficiency of a particular algorithm.

Big-Oh notation is expressed as O(f(n)), where “f(n)” is a function of the size of the input data. For example, an algorithm with a time complexity of O(n) is said to have a linear time complexity, meaning that the number of operations performed by the algorithm is proportional to the size of the input data.

Big-Oh notation only provides an upper bound on the growth rate of the time complexity, and does not provide an exact measure of the running time of an algorithm. For example, an algorithm with a time complexity of O(n) may take 100 operations for an input of size 100, but only 10 operations for an input of size 10. The big O notation only indicates that the number of operations will not grow faster than linearly with the size of the input.

Big-Oh notation is a useful tool for comparing the performance of different algorithms and for evaluating the efficiency of a particular algorithm. Some common time complexities expressed using big O notation include O(1) for constant time, O(log n) for logarithmic time, O(n) for linear time, O(n log n) for log-linear time, and O(n^2) for quadratic time.

In summary, big O notation is a mathematical notation used to describe the upper bound on the growth rate of the time complexity of an algorithm. It provides a way to compare the performance of different algorithms and to evaluate the efficiency of a particular algorithm

Algorithms and flowcharts (Real Life Examples)

Algorithms and flowcharts are tools that are commonly used in a variety of real-life situations to represent and solve problems in a structured and efficient manner. Here are a few examples of how algorithms and flowcharts are used in real life:

1. Cooking recipes: Cooking recipes are often written as algorithms, with each step represented in a clear and sequential manner. For example, a recipe for making cookies might include steps such as: preheat oven, mix ingredients, roll dough, cut into shapes, bake for a specified time, and cool on a wire rack.
2. Banking transactions: Banks use algorithms and flowcharts to process transactions and manage customer accounts. For example, a flowchart might represent the steps involved in processing a customer deposit, including verifying the customer’s identity, verifying the deposit amount, updating the customer’s account balance, and printing a receipt.
3. GPS navigation: GPS navigation systems use algorithms and flowcharts to determine the most efficient route to a destination. For example, a flowchart might represent the steps involved in calculating the shortest route, including determining the starting and ending points, considering factors such as traffic and road conditions, and providing turn-by-turn instructions.
4. Sorting and searching algorithms: Sorting and searching algorithms are commonly used in real life to organize and find information. For example, a search algorithm might be used to find a specific item in a large database, while a sorting algorithm might be used to sort a list of names alphabetically.
5. Manufacturing processes: Manufacturing processes often use algorithms and flowcharts to represent the steps involved in producing a product. For example, a flowchart might represent the steps involved in making a car, including assembling the engine, installing the transmission, adding the wheels and body, and painting the car.

These are just a few examples of how algorithms and flowcharts are used in real life to solve problems and automate processes

Unit-5: Simple arithematic problems

addition

Here's an example of adding two numbers in C:

#include<studio.h>

int main() {
int a, b, sum;

```
printf("Enter two numbers: ");
scanf("%d%d", &a, &b);

sum = a + b;

printf("The sum of %d and %d is: %d\n", a, b, sum);

return 0;
```

}

This program first takes two numbers as input from the user using the `scanf()` function. Then, it calculates their sum and stores it in the variable `sum`. Finally, it prints the result using the `printf()` function

Multiplication of integers

Here's an example of multiplying two integers in C:

#include<studio.h>

int main() {
int a, b, product;

```
printf("Enter two integers: ");
scanf("%d%d", &a, &b);

product = a * b;

printf("The product of %d and %d is: %d\n", a, b, product);

return 0;
```

}

Determining if a number is +ve / -ve

Here’s an example of determining if a number is positive or negative in C:

#include<studio.h>

int main() {
int num;

```
printf("Enter a number: ");
scanf("%d", &num);

if (num >= 0) {
    printf("%d is a positive number.\n", num);
} else {
    printf("%d is a negative number.\n", num);
}

return 0;
```

}

This program takes a number as input from the user using the `scanf()` function. Then, it uses an `if-else` statement to determine whether the number is positive or negative. If the number is greater than or equal to 0, it is considered positive. Otherwise, it is considered negative. The result is printed using the `printf()` function.

determining if a number is even or odd

Here's an example of determining if a number is even or odd in C:

#include<studio.h>

int main() {
int num;

```
printf("Enter a number: ");
scanf("%d", &num);

if (num % 2 == 0) {
    printf("%d is an even number.\n", num);
} else {
    printf("%d is an odd number.\n", num);
}

return 0;
```

}

This program takes a number as input from the user using the `scanf()` function. Then, it uses the modulo operator `%` to check if the number is divisible by 2. If the remainder is 0, the number is considered even. Otherwise, it is considered odd. The result is printed using the `printf()` function

Maximum of 2 numbers, 3 numbers

Here's an example of finding the maximum of two numbers in C:

```
#include <stdio.h>

int main() {
    int num1, num2, max;

    printf("Enter two numbers: ");
    scanf("%d%d", &num1, &num2);

    max = (num1 > num2) ? num1 : num2;

    printf("The maximum of %d and %d is: %d\n", num1, num2, max);

    return 0;
}
```

This program takes two numbers as input from the user using the `scanf()` function. Then, it uses a ternary operator `?:` to determine the maximum of the two numbers. The result is stored in the variable `max` and is printed using the `printf()` function.

Here’s an example of finding the maximum of three numbers in C:

Here’s an example of finding the maximum of two numbers in C:

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

int main() {
    int num1, num2, max;

    printf("Enter two numbers: ");
    scanf("%d%d", &num1, &num2);

    max = (num1 > num2) ? num1 : num2;

    printf("The maximum of %d and %d is: %d\n", num1, num2, max);

    return 0;
}
```

This program takes two numbers as input from the user using the `scanf()` function. Then, it uses a ternary operator `?:` to determine the maximum of the two numbers. The result is stored in the variable `max` and is printed using the `printf()` function.

Here’s an example of finding the maximum of three numbers in C:

```
#include <stdio.h>

int main() {
    int num1, num2, num3, max;

    printf("Enter three numbers: ");
    scanf("%d%d%d", &num1, &num2, &num3);

    max = num1;
    if (num2 > max) max = num2;
    if (num3 > max) max = num3;

    printf("The maximum of %d, %d and %d is: %d\n", num1, num2, num3,
max);

    return 0;
}
```

This program takes three numbers as input from the user using the `scanf()` function. Then, it uses an `if` statement to determine the maximum of the three numbers. The result is stored in the variable `max` and is printed using the `printf()` function.

Sum of first n numbers

Here’s an example of finding the sum of the first n numbers in C:

```c
#include <stdio.h>

int main() {
    int n, i, sum = 0;

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

    for (i = 1; i <= n; i++) {
        sum += i;
    }

    printf("The sum of the first %d numbers is: %d\n", n, sum);

    return 0;
}
```

This program takes a positive integer `n` as input from the user using the `scanf()` function. Then, it uses a `for` loop to calculate the sum of the first `n` numbers. The result is stored in the variable `sum` and is printed using the `printf()` function

sum of given n numbers

Here's an example of finding the sum of `n` numbers in C:

```c
#include <stdio.h>

int main() {
    int n, i, num, sum = 0;

    printf("Enter the number of numbers: ");
    scanf("%d", &n);

    for (i = 1; i <= n; i++) {
        printf("Enter number %d: ", i);
        scanf("%d", &num);
        sum += num;
    }

    printf("The sum of the %d numbers is: %d\n", n, sum);

    return 0;
}
```

This program takes the number of numbers `n` as input from the user using the `scanf()` function. Then, it uses a `for` loop to take `n` numbers as input from the user and calculates the sum of the numbers. The result is stored in the variable `sum` and is printed using the `printf()` function.

Integer division

Here's an example of integer division in C:

```
#include <stdio.h>

int main() {
    int num1, num2, result;

    printf("Enter two numbers: ");
    scanf("%d%d", &num1, &num2);

    result = num1 / num2;

    printf("The result of %d / %d is: %d\n", num1, num2, result);

    return 0;
}
```

This program takes two numbers as input from the user using the `scanf()` function. Then, it performs integer division of the two numbers and stores the result in the variable `result`. The result is printed using the `printf()` function.

Note that integer division in C discards the fractional part and returns the largest integer that is less than or equal to the exact mathematical result. For example, `7 / 2` would give `3` as the result, not `3.5`

digit reversal

Digit reversing, Table generation for n jn c with example

Here’s an example of digit reversal in C:

```c
#include <stdio.h>

int main() {
    int num, reversed = 0, remainder;

    printf("Enter an integer: ");
    scanf("%d", &num);

    while (num != 0) {
        remainder = num % 10;
        reversed = reversed * 10 + remainder;
        num /= 10;
    }

    printf("The reversed number is: %d\n", reversed);

    return 0;
}
```

This program takes an integer as input from the user using the `scanf()` function. Then, it uses a `while` loop to reverse the digits of the number. In each iteration, the last digit of the number is obtained using the modulo operator (%) and is appended to the reversed number. The number is then divided by 10 to remove the last digit. The reversed number is stored in the variable `reversed` and is printed using the `printf()` function.

Table generation for n

Here’s an example of generating a multiplication table for a given number `n` in C:

```c
#include <stdio.h>

int main() {
    int n, i;

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

    for (i = 1; i <= 10; i++) {
        printf("%d * %d = %d\n", n, i, n * i);
    }

    return 0;
}
```

This program takes a positive integer `n` as input from the user using the `scanf()` function. Then, it uses a `for` loop to generate and print the multiplication table for `n` for the first 10 numbers. The result of each multiplication is calculated using `n * i` and is printed using the `printf()` function

factorial of n

Here’s an example of finding the factorial of a number in C:

```c
#include <stdio.h>

int main() {
    int num, i, factorial = 1;

    printf("Enter an integer: ");
    scanf("%d", &num);

    for (i = 1; i <= num; i++) {
        factorial *= i;
    }

    printf("The factorial of %d is: %d\n", num, factorial);

    return 0;
}
```

This program takes an integer as input from the user using the `scanf()` function. Then, it uses a `for` loop to calculate the factorial of the number. The loop starts from `1` and continues till the number itself. In each iteration, the value of `factorial` is updated by multiplying it with the current value of `i`. The final value of `factorial` is the factorial of the number. The result is printed using the `printf()` function.

sine series

Here’s an example of finding the sine series representation of an angle in C:

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

int main() {
    double angle, radian, sum = 0, term;
    int i, n, sign;

    printf("Enter the angle in degrees: ");
    scanf("%lf", &angle);

    // Convert the angle from degrees to radians
    radian = angle * M_PI / 180.0;

    printf("Enter the number of terms: ");
    scanf("%d", &n);

    sign = 1;
    for (i = 0; i < n; i++) {
        term = sign * pow(radian, 2 * i + 1) / (2 * i + 1);
        sum += term;
        sign = -sign;
    }

    printf("The sine of %lf degrees is approximately %lf\n", angle,
sum);

    return 0;
}
```

This program takes an angle in degrees as input from the user using the `scanf()` function. It then converts the angle to radians using the formula `radian = angle * M_PI / 180.0`. The number of terms in the sine series representation is also taken as input from the user.

The `for` loop is used to calculate the sum of the sine series representation of the angle. In each iteration, the value of the `term` is calculated using the formula `term = sign * pow(radian, 2 * i + 1) / (2 * i + 1)` where `sign` is either 1 or `-1` and is changed in each iteration to alternate between positive and negative terms. The value of `term` is added to the running sum in each iteration.

The final value of `sum` is the sine series representation of the angle. The result is printed using the `printf()` function.

cosine series

Here's an example of finding the cosine series representation of an angle in C:

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

int main() {
    double angle, radian, sum = 0, term;
    int i, n;

    printf("Enter the angle in degrees: ");
    scanf("%lf", &angle);

    // Convert the angle from degrees to radians
    radian = angle * M_PI / 180.0;

    printf("Enter the number of terms: ");
    scanf("%d", &n);

    for (i = 0; i < n; i++) {
        term = pow(-1, i) * pow(radian, 2 * i) / (2 * i)!;
        sum += term;
    }

    printf("The cosine of %lf degrees is approximately %lf\n", angle,
sum);

    return 0;
}
```

This program takes an angle in degrees as input from the user using the `scanf()` function. It then converts the angle to radians using the formula `radian = angle * M_PI / 180.0`. The number of terms in the cosine series representation is also taken as input from the user.

The `for` loop is used to calculate the sum of the cosine series representation of the angle. In each iteration, the value of the `term` is calculated using the formula `term = pow(-1, i) * pow(radian, 2 * i) / (2 * i)!` where `(2 * i)!` is the factorial of `2 * i`. The value of `term` is added to the running sum in each iteration.

The final value of `sum` is the cosine series representation of the angle. The result is printed using the `printf()` function.

Here's an example of finding the number of combinations (nCr) in C:

```
#include <stdio.h>

int factorial(int n) {
    int i, result = 1;
    for (i = 1; i <= n; i++) {
        result *= i;
    }
    return result;
}

int nCr(int n, int r) {
    return factorial(n) / (factorial(r) * factorial(n - r));
}

int main() {
    int n, r, result;

    printf("Enter the values of n and r: ");
    scanf("%d%d", &n, &r);

    result = nCr(n, r);

    printf("%dC%d = %d\n", n, r, result);

    return 0;
}
```

This program takes the values of `n` and `r` as input from the user using the `scanf()` function. The function `factorial(int n)` calculates the factorial of `n`. The function `nCr(int n, int r)` calculates the number of combinations `nCr` using the formula `nCr = n! / (r! * (n - r)!)`.

The final value of `result` is the number of combinations. The result is printed using the `printf()` function.

pascal triangle

Here’s an example of printing Pascal’s Triangle in C:

```
#include <stdio.h>

int factorial(int n) {
    int i, result = 1;
    for (i = 1; i <= n; i++) {
        result *= i;
    }
    return result;
}

int nCr(int n, int r) {
    return factorial(n) / (factorial(r) * factorial(n - r));
}

int main() {
    int i, j, rows, value;

    printf("Enter the number of rows: ");
    scanf("%d", &rows);

    for (i = 0; i < rows; i++) {
        for (j = 0; j <= i; j++) {
            value = nCr(i, j);
            printf("%d ", value);
        }
        printf("\n");
    }

    return 0;
}
```

This program takes the number of rows as input from the user using the `scanf()` function. The function `factorial(int n)` calculates the factorial of `n`, and the function `nCr(int n, int r)` calculates the number of combinations `nCr` using the formula `nCr = n! / (r! * (n - r)!)`.

The `for` loop is used to print the Pascal's Triangle. In each iteration of the outer `for` loop, a new row is printed. In each iteration of the inner `for` loop, the value of `nCr(i, j)` is calculated and printed, where `i` and `j` are the row number and the column number, respectively. The final output is a triangular pattern of numbers representing Pascal's Triangle.

Here's an example of finding the prime numbers and factors of a number in C:

```c
#include <stdio.h>

int is_prime(int num) {
    int i;
    if (num <= 1) {
        return 0;
    }
    for (i = 2; i < num; i++) {
        if (num % i == 0) {
            return 0;
        }
    }
    return 1;
}

void find_factors(int num) {
    int i;
    printf("Factors of %d: ", num);
    for (i = 1; i <= num; i++) {
        if (num % i == 0) {
            printf("%d ", i);
        }
    }
    printf("\n");
}

int main() {
    int num;

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

    if (is_prime(num)) {
        printf("%d is a prime number\n", num);
    } else {
        printf("%d is not a prime number\n", num);
        find_factors(num);
    }

    return 0;
}
```

This program takes a number as input from the user using the `scanf()` function. The function `is_prime(int num)` checks if the number is prime or not. If the number is less than or equal to 1, it returns 0, which indicates that the number is not prime. If the number is greater than 1, the program checks if it is divisible by any number other than 1 and itself. If it is divisible, the function returns 0, which indicates that the number is not prime. Otherwise, it returns 1, which indicates that the number is prime.

The function `find_factors(int num)` prints the factors of the number by checking if the number is divisible by each number from 1 to itself. If it is divisible, the divisor is printed.

The final output is either that the number is a prime number or it is not a prime number along with its factors.

perfect number

Here is an example of code in C that determines if a given number is a perfect number:

```
#include <stdio.h>

int main()
{
    int num, i, sum = 0;

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

    for (i = 1; i < num; i++)
    {
        if (num % i == 0)
        {
            sum += i;
        }
    }

    if (sum == num)
        printf("%d is a perfect number", num);
    else
        printf("%d is not a perfect number", num);

    return 0;
}
```

A perfect number is a positive integer that is equal to the sum of its positive divisors excluding itself. In this code, the sum of the positive divisors is calculated in a for loop and compared to the input number to determine if it is a perfect number or not.

GCD numbers

Here is an example of code in C that determines if a given number is a perfect number:

```
#include <stdio.h>

int main()
{
    int num, i, sum = 0;

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

    for (i = 1; i < num; i++)
    {
        if (num % i == 0)
        {
            sum += i;
        }
    }

    if (sum == num)
        printf("%d is a perfect number", num);
    else
        printf("%d is not a perfect number", num);

    return 0;
}
```

A perfect number is a positive integer that is equal to the sum of its positive divisors excluding itself. In this code, the sum of the positive divisors is calculated in a for loop and compared to the input number to determine if it is a perfect number or not.

swapping

Swapping in a C program means exchanging the values of two variables. Here’s an example:

```
#include <stdio.h>

void swap(int *a, int *b) {
  int temp = *a;
  *a = *b;
  *b = temp;
}

int main() {
  int x = 10, y = 20;
  printf("Before swapping: x = %d, y = %d\n", x, y);

  swap(&x, &y);

  printf("After swapping: x = %d, y = %d\n", x, y);
  return 0;
}
```

Output:

```
yamlCopy codeBefore swapping: x = 10, y = 20
After swapping: x = 20, y = 10
```

In this example, the `swap` function takes two pointers to integers as input and swaps their values using a temporary variable. The main function initializes two variables `x` and `y`, prints their values, calls the `swap` function, and finally prints their values again to show that they have been swapped.

Here’s a simple algorithm for swapping two values:

1. Declare a temporary variable `temp`.
2. Assign the value of the first variable to `temp`.
3. Assign the value of the second variable to the first variable.
4. Assign the value of `temp` to the second variable.

Here’s a flowchart for the swapping algorithm:

```
+-----------------+
|   temp = A      |
+-----------------+
        |
+-----------------+
|   A = B         |
+-----------------+
        |
+-----------------+
|   B = temp      |
+-----------------+
```

Note that this algorithm can be applied to any two values, not just variables. In the flowchart, `A` and `B` represent the two values being swapped.

Unit-6:Functions

basic types of function

In C programming, there are two basic types of functions:

1. Library functions: These are the built-in functions in C that are included in the C library. For example, `printf`, `scanf`, `strcpy`, etc. Library functions are used to perform specific tasks, such as reading from the standard input or writing to the standard output, and they return a value of specific data type.
2. User-defined functions: These are the functions that are defined by the user to perform specific tasks in the program. User-defined functions can either return a value or return nothing (void). To use a user-defined function, you need to write its definition and call it in the main program or in other functions.

In both cases, functions can take zero or more arguments as input and can return a value of any data type, such as int, float, char, etc. Functions are useful in breaking down complex problems into smaller and manageable tasks, and they help to increase the modularity and reusability of the code

Declaration and definition

In C programming, declaration and definition are two different concepts.

Declaration: A declaration is a statement that informs the compiler about the name, return type, and number of arguments of a function. The declaration does not provide the implementation of the function, but it tells the compiler that the function exists and can be called from other parts of the program. A declaration can be placed in a header file or in the main program.

Syntax:

```
return_type function_name(argument_type argument_name, argument_type
argument_name, ...);
```

Example:

```
int max(int a, int b);
```

In this example, the declaration of the function `max` informs the compiler that the function takes two integer arguments and returns an integer value.

Definition: A definition is a statement that provides the implementation of a function. The definition contains the code that specifies what the function does. A definition can be placed in a source file or in the main program.

Syntax:

```
return_type function_name(argument_type argument_name, argument_type
argument_name, ...) {
  // Function body
}
```

Example:

```
int max(int a, int b) {
  if (a > b) {
    return a;
  } else {
    return b;
  }
}
```

In this example, the definition of the function `max` implements the code that returns the maximum of two integers.

It is important to note that a function can be declared multiple times, but it must be defined only once in the program. If a function is declared in a header file, it should be defined in a source file. The declaration and definition should match exactly in terms of the return type, function name, and number and type of arguments.

function call

A function call is a statement that invokes a function and causes the code in the function definition to be executed. In a function call, you pass values to the function as arguments and the function returns a value or performs some action.

Syntax:

```
function_name(argument_value, argument_value, ...);
```

Example:

```
#include <stdio.h>

int max(int a, int b) {
  if (a > b) {
    return a;
  } else {
    return b;
  }
}

int main() {
  int x = 10, y = 20;
  int result = max(x, y);
  printf("The maximum of %d and %d is %d\n", x, y, result);
  return 0;
}
```

In this example, the function `max` takes two integer arguments `a` and `b` and returns the maximum of the two. The `main` function calls the `max` function by passing `x` and `y` as arguments, and it stores the result in the `result` variable. Finally, it prints the result.

When a function is called, the values of the arguments are passed to the corresponding parameters in the function definition. The function performs the actions specified in its definition, and if it returns a value, that value can be stored in a variable or used in an expression. If a function does not return a value, it is considered to return void

Types of function

In C programming, there are several types of functions based on the return type and the number and type of arguments they take. Here are some of the most common types:

1. Functions with no return type and no arguments: These functions perform a task and do not return any value. The return type of such functions is void.

Example:

```
#include <stdio.h>

void print_message() {
  printf("Hello, World!\n");
}

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

2. Functions with no return type and some arguments: These functions perform a task and take one or more arguments. The return type of such functions is void.

Example:

```
#include <stdio.h>

void print_sum(int a, int b) {
  int sum = a + b;
  printf("The sum of %d and %d is %d\n", a, b, sum);
}

int main() {
  print_sum(10, 20);
  return 0;
}
```

3. Functions with a return type and no arguments: These functions perform a task and return a value. The return type can be any data type, such as int, float, char, etc.

Example:

```
#include <stdio.h>

int square(int x) {
  return x * x;
}

int main() {
  int x = 10;
  int result = square(x);
  printf("The square of %d is %d\n", x, result);
  return 0;
}
```

4. Functions with a return type and some arguments: These functions perform a task, take one or more arguments, and return a value. The return type can be any data type, such as int, float, char, etc.

Example:

```c
#include <stdio.h>

int max(int a, int b) {
  if (a > b) {
    return a;
  } else {
    return b;
  }
}

int main() {
  int x = 10, y = 20;
  int result = max(x, y);
  printf("The maximum of %d and %d is %d\n", x, y, result);
  return 0;
}
```

These are some of the most common types of functions in C programming. Functions can be combined and nested to perform complex tasks. By dividing a program into functions, you can make the code more modular, readable, and reusable

Parameter passing

In C, when a function is called, the arguments passed to the function are passed to its parameters. There are three ways to pass parameters to a function in C:

1. Call by value: In this method, a copy of the argument value is passed to the function. The function operates on the copy, and any changes made to the parameter within the function are not reflected in the original argument.

Example:

```
#include <stdio.h>

void swap(int a, int b) {
  int temp = a;
  a = b;
  b = temp;
}

int main() {
  int x = 10, y = 20;
  printf("Before swapping: x = %d, y = %d\n", x, y);
  swap(x, y);
  printf("After swapping: x = %d, y = %d\n", x, y);
  return 0;
}
```

Output:

```
Before swapping: x = 10, y = 20
After swapping: x = 10, y = 20
```

As you can see, the values of `x` and `y` are not swapped because the `swap` function operates on the copy of the argument values and not on the original values.

2. Call by reference: In this method, a reference to the argument value is passed to the function. The function operates on the original value, and any changes made to the parameter within the function are reflected in the original argument.

Example:

```
#include <stdio.h>

void swap(int *a, int *b) {
  int temp = *a;
  *a = *b;
  *b = temp;
}

int main() {
  int x = 10, y = 20;
  printf("Before swapping: x = %d, y = %d\n", x, y);
  swap(&x, &y);
  printf("After swapping: x = %d, y = %d\n", x, y);
  return 0;
}
```

Output:

```
Before swapping: x = 10, y = 20
After swapping: x = 20, y = 10
```

In this example, the `swap` function takes two pointer arguments `a` and `b`, and it swaps the values pointed by these pointers. The `main` function passes the addresses of `x` and `y` to the `swap` function, and as a result, the values of `x` and `y` are swapped.

3. Call by address: This is similar to call by reference, but instead of using pointers, the function takes the address of the argument directly.

Example:

```
#include <stdio.h>

void swap(int &a, int &b) {
  int temp = a;
  a = b;
  b = temp;
}

int main() {
  int x = 10, y = 20;
  printf("Before swapping: x = %d, y = %d\n", x, y);
  swap(x, y);
  printf("After swapping: x = %d, y = %d\n", x, y);
  return 0;
}
```

Output:

```
Before swapping: x = 10, y = 20
After swapping: x
```

scope of variable

A variable's scope refers to the region of the program where the variable can be accessed. In other words, it defines the visibility and lifetime of a variable. The scope of a variable can be classified into two types:

1. Local scope: A variable with local scope is defined within a function or a block of code and can only be accessed within that specific block. Once the control leaves the block, the variable and its value are destroyed.
2. Global scope: A variable with global scope is defined outside of any function or block of code and can be accessed by any part of the program. Global variables are created and initialized when the program starts and their values persist until the program ends.

It is important to note that the same variable name can be used in different scopes, but the value and access of each variable is independent of each other. In such cases, the variable with the local scope takes precedence over the global variable with the same name.

Storage classes

Storage classes are a way to specify the lifetime and visibility of a variable in a program. They define how the memory for a variable is allocated and how the variable can be accessed. The most common storage classes are:

1. Automatic storage class: Variables declared inside a function with no storage class specified are automatically assigned the “automatic” storage class. These variables are also called “local variables”. They are stored on the stack and are created and destroyed each time the function is called. Example:

```
void func() {
  int x = 10;
  // x is an automatic variable
}
```

2. Register storage class: Variables declared with the “register” storage class are stored in CPU registers instead of in memory. This can increase the program’s performance, since access to CPU registers is faster than access to memory. However, the number of register variables is limited, so not all variables declared as “register” will actually be stored in a register. Example:

```
void func() {
  register int x = 10;
  // x is a register variable
}
```

3. Static storage class: Variables declared with the “static” storage class have a scope that is limited to the function in which they are declared, but their value is preserved between function calls. In other words, they are initialized only once and their value is retained between function calls. Example:

```
void func() {
  static int x = 10;
  // x is a static variable
}
```

4. External storage class: Variables declared outside of any function with no storage class specified are automatically assigned the “external” storage class. These variables are also called “global variables”. They can be accessed by any function in the program. Example:

```
int x = 10;
// x is an external variable
```

It is important to choose the right storage class for a variable, as it affects the performance and behavior of the program.

recursion

Recursion is a technique in computer programming where a function calls itself in order to solve a problem. A function that uses recursion is called a recursive function.

In a recursive function, the problem to be solved is divided into smaller sub-problems, and the function is called recursively on each sub-problem until a base case is reached. The base case is a stopping condition that stops the recursion.

For example, consider the problem of computing the factorial of a number. The factorial of a number `n` is defined as `n! = n * (n-1) * (n-2) * ... * 1`. To solve this problem using recursion, we can define a recursive function `factorial` that takes an integer `n` as input and returns `n!`. The function can be defined as follows:

```
int factorial(int n) {
  if (n == 0) {
    return 1;  // base case
  } else {
    return n * factorial(n-1);  // recursive call
  }
}
```

The function checks if the input `n` is equal to `0`, which is the base case. If `n` is equal to `0`, the function returns `1`, which is the factorial of `0`. If `n` is not equal to `0`, the function returns `n * factorial(n-1)`. This is the recursive call, where the function calls itself with `n-1` as the input. The recursion continues until the base case is reached.

For example, to compute the factorial of `5`, we can call the function as follows:

```
    int result = factorial(5);
```

The function will call itself with the following inputs: `factorial(5)`, `factorial(4)`, `factorial(3)`, `factorial(2)`, `factorial(1)`, `factorial(0)`. When the input is `factorial(0)`, the base case is reached and the function returns 1. The previous calls to the function return `1 * 2 * 3 * 4 * 5 = 120`, which is the factorial of 5.

It is important to note that recursion can be a powerful technique, but it can also be dangerous if not used correctly. A recursive function must have a clear base case and a correct recursive step to avoid infinite recursion, which can cause a stack overflow error.