**Why Learn C Programming?**

1. **Embedded Systems:**
   * C allows for close interaction with hardware and efficient code execution, making it suitable for embedded applications.
2. **Microcontroller Programming:**
   * Understanding C enables students to write firmware and control systems using these microcontrollers.
3. **Low-Level Programming:**
   * C provides a closer-to-hardware programming experience compared to higher-level languages. Low-level programming for tasks like interfacing with sensors, motors, and other hardware components.
4. **Efficiency and Performance:**
   * C is known for its efficiency and performance. In fields like signal processing or real-time systems.
5. **Compatibility with Hardware:**
   * C allows for direct memory manipulation and access, making it suitable for tasks where precise control over hardware resources is required.
6. **Industry Standards:**
   * Many industrial applications and tools in the field of EEE are written in C.
7. **Portability:**
   * Code written in C is often portable across different platforms, which is advantageous when working on diverse hardware and systems.
8. **Foundational Programming Skills:**
   * Learning C provides foundational programming skills that can be applied in other languages. Once you understand the principles of programming in C, transitioning to other languages becomes smoother.
9. **Firmware Development:**
   * C is commonly used for firmware development, which is crucial in EEE for programming the software that runs on hardware devices.

**C Programming Syllabus**

**1. Introduction to Programming:**

* Basics of programming languages.
* Overview of C programming language.
* Setting up the development environment.

**2. Basic C Programming Concepts:**

* Variables, data types, and operators.
* Input and output functions.
* Control flow: if-else statements, switch-case, loops (for, while, do-while).

**3. Functions and Modular Programming:**

* Function declaration and definition.
* Function prototypes.
* Function parameters and return values.
* Modular programming concepts.

**4. Arrays and Strings:**

* Array declaration and initialization.
* Multidimensional arrays.
* String handling functions.

**5.** **Pointers and Dynamic Memory Allocation:**

* Understanding pointers.
* Pointer arithmetic.
* Dynamic memory allocation.

**6. Structures and Unions:**

* Defining structures and unions.
* Accessing structure members.
* Nested structures.

**7. File Handling:**

* File operations (opening, reading, writing, closing).
* File pointers and random access.

**8. Preprocessor Directives:**

* Macros and conditional compilation.
* Header files and include directives.

**9. Bitwise Operations:**

* Understanding bitwise operators.
* Bit manipulation techniques.

**10. Introduction to Data Structures:** - Basic concepts of data structures (linked lists, stacks, queues). - Overview of algorithms.

**11. Introduction to Embedded C:** - Basics of programming microcontrollers. - Interfacing with hardware.

**12. Advanced Topics (Optional):** - Recursion. - Enumerations. - Advanced data structures (trees, graphs).

**13. Application Development:** - Developing small applications and projects. - Debugging techniques.

**Basics of Programming Languages**

**What is Programming Language?**

A programming language is a set of instructions written by a programmer to deliver instructions to the computer to perform and accomplish a task.

A programming language is a type of written language that tells computers what to do. Examples are: Python, Ruby, Java, JavaScript, C, C++, and C#. Programming languages are used to write computer programs and computer software.

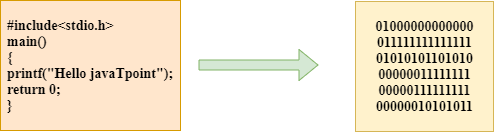
**Overview of C programming language.**

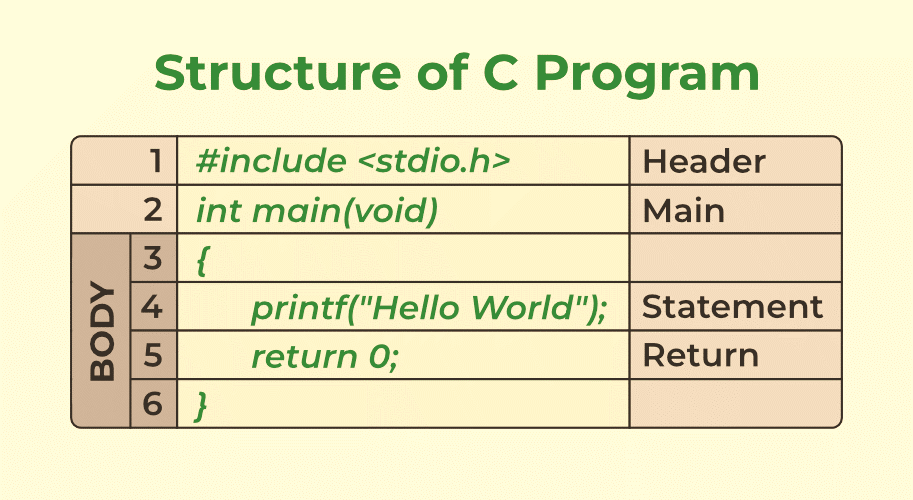
* + C is a procedural programming language
  + Developed by **Dennis Ritchie** in the year **1972** at **Bell Laboratories of AT&T Labs**.
  + It was mainly developed as a system programming language to write the UNIX operating system.

**Compilation process in C**

**What is a compilation?**

* The compilation is a process of converting the source code into object code.
* It is done with the help of the compiler.
* The compiler checks the source code for the syntactical or structural errors, and if the source code is error-free, then it generates the object code.





**#include <stdio.h>** includes the **standard input output** library functions. The printf() function is defined in stdio.h .

**int main()** The **main() function is the entry point of every program** in c language.

**printf()** The printf() function is **used to print data** on the console.

**return 0** The return 0 statement, returns execution status to the OS. The 0 value is used for successful execution and 1 for unsuccessful execution.

**#include <stdio.h>**

**int main() {**

**// Your program logic here**

**// Simulate an error condition**

**if (/\* some condition indicating an error \*/) {**

**printf("Error occurred.\n");**

**// Return a non-zero value to indicate an error**

**return 1;**

**}**

**// Normal termination, return 0**

**return 0;**

**}**

**Real Time Example for “return 0” and “return 1”**

**#include <stdio.h>**

**int main() {**

**int num;**

**printf("Enter an integer: ");**

**// Try to read an integer from the user**

**if (scanf("%d", &num) != 1) {**

**// scanf returns the number of successful conversions,**

**// so if it's not 1, the input was not a valid integer**

**printf("Invalid input. Please enter an integer.\n");**

**// Return a non-zero value to indicate an error**

**return 1;**

**}**

**// Check if the entered number is even or odd**

**if (num % 2 == 0) {**

**printf("%d is an even number.\n", num);**

**} else {**

**printf("%d is an odd number.\n", num);**

**}**

**// Normal termination, return 0**

**return 0;**

**}**

**2. BASIC C PROGRAMMING CONCEPTS**

**Variable:**

A **variable** is a name of the memory location.

It is used to store data.

Its value can be changed, and it can be reused many times.

Each variable should be given a unique name (identifier).

**Syntax:**

type variable\_list;

**Example Variable:**

**int** a;

**float** b;

**char** c;

We can also provide values while declaring the variables.

**Example:**

int a=10, b=20;//declaring 2 variable of integer type

float f=20.8;

char c='A';

**Rules for defining variables:**

* A variable can have alphabets, digits, and underscore.
* A variable name can start with the alphabet, and underscore only. It can't start with a digit.
* No whitespace is allowed within the variable name.
* A variable name must not be any reserved word or keyword, e.g. int, float, etc.
* Variable is Case Sensitive.

**Valid variable names:**

**int** a;

**int** \_ab;

**int** a30;

**Invalid variable names:**

**int** 2;

**int** a b;

**int** long;

**Datatype:**

* Data types are used while defining a variable or functions
* It’s important for the compiler to understand the type of predefined data

|  |  |  |
| --- | --- | --- |
| **Type** | **Size (bytes)** | **Format Specifier** |
| **int** | **2** | **%d** |
| **char** | **1** | **%c** |
| **float** | **4** | **%f** |
| **double** | **8** | **%lf** |

**Format specifiers:**

Format specifiers define the type of data to be printed on standard output. You need to use format specifiers whether you're printing formatted output with **printf()** or accepting input with **scanf().**

**Keywords in C:**

A keyword is a **reserved word**. You cannot use it as a variable name, constant name, etc. There are only **32** reserved words (keywords) in the C language.

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

**C Operators:**

An operator is a symbol that is used to perform operations.

**Arithmetic Operators**

An arithmetic operator performs mathematical operations such as addition, subtraction, multiplication, division etc on numerical values. This arithmetic operator is binary operator.

Binary operator : one **operator** operate **two operand.**

**a + b ,** here **“a”** and **“b”** isoperand **“+”** is operator.

**Operator Meaning of Operator**

+ addition or unary plus

- subtraction or unary minus

\* multiplication

/ division

% remainder after division (modulo division)

**Example: Arithmetic Operators**

**// Working of arithmetic operators**

**#include <stdio.h>**

**int main()**

**{**

**int a = 9,b = 4, c;**

**c = a+b;**

**printf("a+b = %d \n",c);**

**c = a-b;**

**printf("a-b = %d \n",c);**

**c = a\*b;**

**printf("a\*b = %d \n",c);**

**c = a/b;**

**printf("a/b = %d \n",c);**

**c = a%b;**

**printf("Remainder when a divided by b = %d \n",c);**

**return 0;**

**}**

**Output is :**

**a+b = 13**

**a-b = 5**

**a\*b = 36**

**a/b = 2**

**Remainder when a divided by b=1**

**Increment and Decrement Operators**

The increment ( ++ ) and decrement ( -- ) operators in C are unary operators for incrementing and decrementing the numeric values by 1 respectively.

Unary operators - meaning they only **operate** on a **single operand**.

// Working of increment and decrement operators

#include <stdio.h>

int main()

{

int a = 10, b = 100;

float c = 10.5, d = 100.5;

printf("++a = %d \n", ++a);

printf("--b = %d \n", --b);

printf("++c = %f \n", ++c);

printf("--d = %f \n", --d);

return 0;

}

Output is:

**++a = 11**

**--b = 99**

**++c = 11.500000**

**--d = 99.500000**

**Increment ++ and Decrement -- Operator as Prefix and Postfix**

**var** = 3;

**++var & var++**

If you use the ++ operator as a prefix like: **++var**, the value of var is incremented by 1; then it returns the value.

If you use the ++ operator as a postfix like: **var++,** the original value of var is returned first; then var is incremented by 1.

**Example ++ :**

#include<stdio.h>

int main()

{

int a=3, b=3;

printf("%d\n", a++);

printf("%d\n, ++b);

return 0;

}

Output is :

3

4

**--var & var--**

If you use the -- operator as a prefix like: **--var**, the value of var is decremented by 1; then it returns the value.

If you use the -- operator as a postfix like: **var--,** the original value of var is returned first; then var is decremented by 1.

#include<stdio.h>

int main()

{

int a=3 ,b=3;

printf("%d\n",a--);

printf("%d\n",--b);

return 0;

}

Output is :

3

2

**C Assignment Operators**

An assignment operator is used for assigning a value to a variable. The most common assignment operator is =

|  |  |  |
| --- | --- | --- |
| **Operator** | **Example** | **Same as** |
| **=** | **a = b** | **a = b** |
| **+=** | **a += b** | **a = a+b** |
| **-=** | **a -= b** | **a = a-b** |
| **\*=** | **a \*= b** | **a = a\*b** |
| **/=** | **a /= b** | **a = a/b** |
| **%=** | **a %= b** | **a = a%b** |

**Example: Assignment Operators**

**// Working of assignment operators**

**#include <stdio.h>**

**int main()**

**{**

**int a = 5, c;**

**c = a; // c is 5**

**printf("c = %d\n", c);**

**c += a; // c is 10**

**printf("c = %d\n", c);**

**c -= a; // c is 5**

**printf("c = %d\n", c);**

**c \*= a; // c is 25**

**printf("c = %d\n", c);**

**c /= a; // c is 5**

**printf("c = %d\n", c);**

**c %= a; // c = 0**

**printf("c = %d\n", c);**

**return 0;**

**}**

**Output is :**

**c = 5**

**c = 10**

**c = 5**

**c = 25**

**c = 5**

**c = 0**

**C Relational Operators**

* **A relational operator checks the relationship between two operands. If the relation is true, it returns 1; if the relation is false, it returns value 0.**
* **Relational operators are used in decision making and loops.**

|  |  |  |
| --- | --- | --- |
| **Operator** | **Meaning of Operator** | **Example** |
| **==** | **Equal to** | **5 == 3 is evaluated to 0** |
| **>** | **Greater than** | **5 > 3 is evaluated to 1** |
| **<** | **Less than** | **5 < 3 is evaluated to 0** |
| **!=** | **Not equal to** | **5 != 3 is evaluated to 1** |
| **>=** | **Greater than or equal to** | **5 >= 3 is evaluated to 1** |
| **<=** | **Less than or equal to** | **5 <= 3 is evaluated to 0** |

**Example: Relational Operators**

**// Working of relational operators**

**#include <stdio.h>**

**int main()**

**{**

**int a = 5, b = 5, c = 10;**

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

**printf("%d == %d is %d \n", a, c, a == c);**

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

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

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

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

**printf("%d != %d is %d \n", a, b, a != b);**

**printf("%d != %d is %d \n", a, c, a != c);**

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

**printf("%d >= %d is %d \n", a, c, a >= c);**

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

**printf("%d <= %d is %d \n", a, c, a <= c);**

**return 0;**

**}**

**Output is :**

**5 == 5 is 1**

**5 == 10 is 0**

**5 > 5 is 0**

**5 > 10 is 0**

**5 < 5 is 0**

**5 < 10 is 1**

**5 != 5 is 0**

**5 != 10 is 1**

**5 >= 5 is 1**

**5 >= 10 is 0**

**5 <= 5 is 1**

**5 <= 10 is 1**

**C Logical Operators**

**An expression containing logical operator returns either 0 or 1 depending upon whether expression results true or false. Logical operators are commonly used in decision making in C programming.**

|  |  |  |
| --- | --- | --- |
| **Operator** | **Meaning** | **Example** |
| **&&** | **Logical AND. True only if all operands are true** | **If c = 5 and d = 2 then, expression ((c==5) && (d>5)) equals to 0.** |
| **||** | **Logical OR. True only if either one operand is true** | **If c = 5 and d = 2 then, expression ((c==5) || (d>5)) equals to 1.** |
| **!** | **Logical NOT. True only if the operand is 0** | **If c = 5 then, expression !(c==5) equals to 0.** |

**Example : Logical Operators**

**// Working of logical operators**

**#include <stdio.h>**

**int main()**

**{**

**int a = 5, b = 5, c = 10, result;**

**result = (a == b) && (c > b);**

**printf("(a == b) && (c > b) is %d \n", result);**

**result = (a == b) && (c < b);**

**printf("(a == b) && (c < b) is %d \n", result);**

**result = (a == b) || (c < b);**

**printf("(a == b) || (c < b) is %d \n", result);**

**result = (a != b) || (c < b);**

**printf("(a != b) || (c < b) is %d \n", result);**

**result = !(a != b);**

**printf("!(a != b) is %d \n", result);**

**result = !(a == b);**

**printf("!(a == b) is %d \n", result);**

**return 0;**

**}**

**Output is :**

**(a == b) && (c > b) is 1**

**(a == b) && (c < b) is 0**

**(a == b) || (c < b) is 1**

**(a != b) || (c < b) is 0**

**!(a != b) is 1**

**!(a == b) is 0**

**Input functions**

**scanf()** is one of the commonly used function to take input from the user. The **scanf()** function reads formatted input from the standard input such as keyboards.

**Example 5: Integer Input/Output**

**#include <stdio.h>**

**int main()**

**{**

**int testInteger;**

**printf("Enter an integer: ");**

**scanf("%d", &testInteger);**

**printf("Number = %d",testInteger);**

**return 0;**

**}**

**Output is :**

**Enter an integer: 4**

**Number = 4**

**Example 6: Float and Double Input/Output**

**#include <stdio.h>**

**int main()**

**{**

**float num1;**

**double num2;**

**printf("Enter a number: ");**

**scanf("%f", &num1);**

**printf("Enter another number: ");**

**scanf("%lf", &num2);**

**printf("num1 = %f\n", num1);**

**printf("num2 = %lf", num2);**

**return 0;**

**}**

**Output is :**

**Enter a number: 12.523**

**Enter another number: 10.2**

**num1 = 12.523000**

**num2 = 10.200000**

**Example 7: C Character I/O**

**#include <stdio.h>**

**int main()**

**{**

**char chr;**

**printf("Enter a character: ");**

**scanf("%c",&chr);**

**printf("You entered %c.", chr);**

**return 0;**

**}**

**Output is :**

**Enter a character: g**

**You entered g**

**Example 8: ASCII Value**

**#include <stdio.h>**

**int main()**

**{**

**char chr;**

**printf("Enter a character: ");**

**scanf("%c", &chr);**

**// When %c is used, a character is displayed**

**printf("You entered %c.\n",chr);**

**// When %d is used, ASCII value is displayed**

**printf("ASCII value is %d.", chr);**

**return 0;**

**}**

**Output is :**

**Enter a character: g**

**You entered g.**

**ASCII value is 103.**

**I/O Multiple Values**

**#include <stdio.h>**

**int main()**

**{**

**int a;**

**float b;**

**printf("Enter integer and then a float: ");**

**// Taking multiple inputs**

**scanf("%d%f", &a, &b);**

**printf("You entered %d and %f", a, b);**

**return 0;**

**}**

**Output is :**

**Enter integer and then a float: -3**

**3.4**

**You entered -3 and 3.400000**

**Output functions**

**printf()** is one of the main output function. The function sends formatted output to the screen.

**Example 1: Print String Statement**

**#include <stdio.h>**

**int main()**

**{**

// Displays the string inside quotations

**printf("C Programming");**

**return 0;**

**}**

**Output is : C Programming**

**Example 2: Integer Output**

**#include <stdio.h>**

**int main()**

**{**

**int testInteger = 5;**

**printf("Number = %d", testInteger);**

**return 0;**

**}**

**Output is : Number = 5**

**Example 3: float and double Output**

**#include <stdio.h>**

**int main()**

**{**

**float number1 = 13.5;**

**double number2 = 12.4;**

**printf("number1 = %f\n", number1);**

**printf("number2 = %lf", number2);**

**return 0;**

**}**

**Output is :**

**number1 = 13.500000**

**number2 = 12.400000**

**Example 4: Print Characters**

**#include <stdio.h>**

**int main()**

**{**

**char chr = 'a';**

**printf("character = %c", chr);**

**return 0;**

**}**

**Output is : character = a**

**Control flow: if-else statements, switch-case, loops (for, while, do-while).**

**CONTROL FLOW**

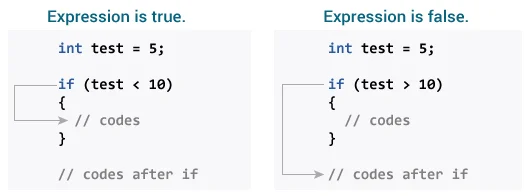
**IF STATEMENT**

How if statement works?

* The if statement evaluates the test expression inside the parenthesis ().
* If the test expression is evaluated to true, statements inside the body of if are executed.
* If the test expression is evaluated to false, statements inside the body of if are not executed.

**SYNTAX**

if (condition) {  
  *// block of code to be executed if the condition is true*  
}



**// Program to display a number if it is negative**

**#include <stdio.h>**

**int main()**

**{**

**int number;**

**printf("Enter an integer: ");**

**scanf("%d", &number);**

**// true if number is less than 0**

**if (number < 0)**

**{**

**printf("\nYou entered digit is negative = %d.\n", number);**

**}**

**printf("Detecting complete.");**

**return 0;**

**}**

**Output is :**

Enter an integer: -3

You Entered digit is negative = -3

Detection complete.

**(or)**

Enter an integer: 2

Detection complete.

**#include<stdio.h>**

**int main()**

**{**

**int time = 3;**

**if(time==4)**

**{**

**printf("Alarm Ringing because time is %d",time);**

**}**

**printf("\nNow time is %d 'o Clock",time);**

**return 0;**

**}**

**Output when time = 3**

Now time is 3 ‘o Clock

**Output when time = 4**

Alarm Ringing because time is 4

Now time is 4 ‘o Clock

**IF ELSE STATEMENT**

**What is the if-else statement?**

**“If”** statement executes a block of code if a specified condition is true. If the condition is false, **“else”** block of code can be executed.

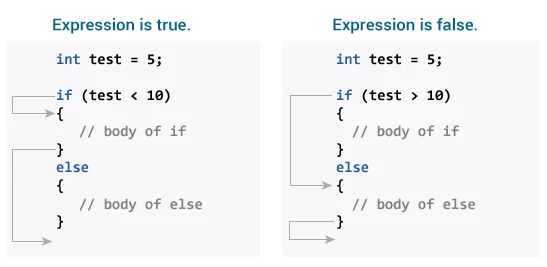
**How if...else statement works?**

If the test expression is evaluated to true,

* statements inside the body of if are executed.
* statements inside the body of else are skipped from execution.

If the test expression is evaluated to false,

* statements inside the body of else are executed
* statements inside the body of if are skipped from execution.



**SYNTAX**

if (condition) {  
  *// block of code to be executed if the condition is true*  
} else {  
  *// block of code to be executed if the condition is false*  
}

// Check whether an integer is odd or even

#include <stdio.h>

int main() {

int number;

printf("Enter an integer: ");

scanf("%d", &number);

// True if the remainder is 0

if (number%2 == 0) {

printf("%d is an even integer.",number);

}

else {

printf("%d is an odd integer.",number);

}

return 0;

}

Output is :

Enter an integer: 7

7 is an odd integer.

**if else if (or) if...else Ladder**

If…else if ladder used to test set of conditions in sequence. An if condition is tested only when all previous if conditions in if-else ladder is false. If any of the conditional expression evaluates to true, then it will execute the corresponding code block and exits whole if-else ladder.

**SYNTAX**

if (test expression1) {

// statement(s)

}

else if(test expression2) {

// statement(s)

}

else if (test expression3) {

// statement(s)

}

.

.

else {

// statement(s)

}

int myNum = 10; // Is this a positive or negative number?  
  
if (myNum > 0) {  
  printf("The value is a positive number.");  
} else if (myNum < 0) {  
  printf("The value is a negative number.");  
} else {  
  printf("The value is 0.");  
}

Output is : **The value is a positive number.**

**Example 2 :**

// Program to relate two integers using =, > or < symbol

#include <stdio.h>

int main() {

int number1, number2;

printf("Enter two integers: ");

scanf("%d %d", &number1, &number2);

//checks if the two integers are equal.

if(number1 == number2) {

printf("Result: %d = %d",number1,number2);

}

//checks if number1 is greater than number2.

else if (number1 > number2) {

printf("Result: %d > %d", number1, number2);

}

//checks if both test expressions are false

else {

printf("Result: %d < %d",number1, number2);

}

return 0;

}

Output is :

12

23

Result: 12 < 23

**Nested if...else**

Nested if statements mean an if statement inside another if statement.

**SYNTAX**

//check if the first condition holds

**if** (condition 1) {

if (condition 2)

{

do something

}

else

{

do something else

}

}

**else**

{

if (condition 3)

{

do something

}

else

{

do something else

}

}

**Example 1 :** analyse if the number is **even** or **odd**, and then if it is **even**, whether it is **divisible by 4** or not, and if it is **odd**, whether it is **divisible by 3** or not will be :

**#include <stdio.h>**

**int main()**

**{**

// variable to store the given number

**int n;**

//take input from the user

**scanf("%d", &n);**

//if else condition to check whether the number is even or odd

**if (n % 2 == 0)**

**{**

//the number is even

**printf("Even ");**

//nested if else condition to check if n is divisible by 4 or not

**if (n % 4 == 0)**

**{**

//the number is divisible by 4

**printf("and divisible by 4");**

**}**

**else**

**{**

//the number is not divisible by 4

**printf("and not divisible by 4");**

**}**

**}**

else

**{**

//the number is odd

printf("Odd ");

//nested if else condition to check if n is divisible by 3 or not

**if (n % 3 == 0)**

{

//the number is divisible by 3

**printf("and divisible by 3");**

}

else

{

//the number is not divisible by 3

**printf("and not divisible by 3");**

}

**}**

**return 0;**

**}**

Input is : **4**

Output is : **Even and not divisible by 4**

**Example 2 : Check if three numbers are equal**

#include <stdio.h>

int main() {

int a, b, c; // variables to store the three numbers

scanf("%d %d %d", &a, &b, &c); //take input from the user

//if else condition to check whether first two numbers are equal

if (a == b)

{

**//nested if else condition to check if c is equal to a and b**

**if (a == c)**

**{**

**//all are equal**

**printf("Yes");**

**}**

**else**

**{**

**//all are not equal**

**printf("No");**

**}**

}

else

{

//the first two numbers are not equal, so they are not equal

printf("No");

}

return 0;

}

Input 1 :

1 3 4

Output 1 :

No

Input 1 :

1 1 1

Output 1 :

Yes

**Example 3 : Which number is greatest among three numbers**

#include <stdio.h>

int main()

{

// variables to store the three numbers

int a, b, c;

//take input from the user

scanf("%d %d %d", &a, &b, &c);

//if else condition to check whether the first number is greater than the second

**if (a > b)**

**{**

**//nested if else condition to check if a>c**

**if (a > c)**

**{**

**//a is greatest**

**printf("%d", a);**

**}**

**else**

**{**

**//c is the greatest**

**printf("%d", c);**

**}**

**}**

**else**

**{**

**//nested if else condition to check if b>c**

**if (b > c) {**

**//b is greatest**

**printf("%d", b);**

**}**

**else**

**{**

**//c is the greatest**

**printf("%d", c);**

**}**

**}**

**return 0;**

**}**

**LOOP**

Loop is used to repeat a block of code until the specified condition is met.

**C programming has three types of loops:**

* for loop
* while loop
* do...while loop

**For Loop**

SYNTAX

for (**initializationStatement; testExpression; updateStatement**)

{

// statements inside the body of loop

}

**How for loop works?**

* The initialization statement is executed only once.
* Then, the test expression is evaluated. If the test expression is evaluated to false, the for loop is terminated.
* However, if the test expression is evaluated to true, statements inside the body of the for loop are executed, and the update expression is updated.
* Again the test expression is evaluated.
* This process goes on until the test expression is false. When the test expression is false, the loop terminates.

**Example For Loop :**

**// Print numbers from 1 to 10**

**#include <stdio.h>**

**int main() {**

**int i;**

**for (i = 1; i < 11; ++i)**

**{**

**printf("%d ", i);**

**}**

**return 0;**

**}**

**Output**

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

**Example 2:**

**// Program to calculate the sum of first n natural numbers**

**// Positive integers 1,2,3...n are known as natural numbers**

**#include <stdio.h>**

**int main()**

**{**

**int num, count, sum = 0;**

**printf("Enter a positive integer: ");**

**scanf("%d", &num);**

**// for loop terminates when num is less than count**

**for(count = 1; count <= num; ++count)**

**{**

**sum += count;**

**}**

**printf("Sum = %d", sum);**

**return 0;**

**}**

Output is :

Enter a positive integer: 10

Sum = 55

Example : 3

#include<stdio.h>

int main()

{

int i =0;

**for(;i<=5;i++); // Semicolon using end of the loop**

printf("%d",i);

return 0;

}

Output :

6

**While Loop**

SYNTAX

while (testExpression) {

// the body of the loop

}

**How while loop works?**

* The while loop evaluates the testExpression inside the parentheses ().
* If testExpression is true, statements inside the body of while loop are executed. Then, testExpression is evaluated again.
* The process goes on until testExpression is evaluated to false.
* If testExpression is false, the loop terminates (ends).

**Example:**

// Print numbers from 1 to 5

#include <stdio.h>

int main() {

int i = 1;

while (i <= 5)

{

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

++i;

}

return 0;

}

Output :

1

2

3

4

5

**Do While Loop**

SYNTAX

do {

// the body of the loop

}

while (testExpression);

**How do...while loop works?**

* The body of do...while loop is executed once. Only then, the testExpression is evaluated.
* If testExpression is true, the body of the loop is executed again and testExpression is evaluated once more.
* This process goes on until testExpression becomes false.
* If testExpression is false, the loop ends.

**EXAMPLE:**

// Program to add numbers until the user enters zero

#include <stdio.h>

int main() {

double number, sum = 0;

// the body of the loop is executed at least once

do {

printf("Enter a number: ");

scanf("%lf", &number);

sum += number;

}

while(number != 0.0);

printf("Sum = %.2lf",sum);

return 0;

}

**Output** :

Enter a number: 1.5

Enter a number: 2.4

Enter a number: -3.4

Enter a number: 4.2

Enter a number: 0

Sum = 4.70

**Switch case**

switch(expression) {

case x:

// code block

break;

case y:

// code block

break;

default:

// code block

}

**How it works:**

* The ***switch*** expression is evaluated once
* The value of the expression is compared with the values of each ***case***
* If there is a match, the associated block of code is executed
* The ***break*** statement breaks out of the switch block and stops the execution
* The ***default*** statement is optional, and specifies some code to run if there is no case match

**EXAMPLE 1 : Finding Days**

#include<stdio.h>

int main()

{

int day = 4;

switch (day)

{

**case 1:**

printf("Monday");

break;

**case 2:**

printf("Tuesday");

break;

**case 3:**

printf("Wednesday");

break;

**case 4:**

printf("Thursday");

break;

**case 5:**

printf("Friday");

break;

**case 6:**

printf("Saturday");

break;

**case 7:**

printf("Sunday");

break;

}

}

**Output is**

Thursday

**Example 2**

#include<stdio.h>

int main()

{

int number=0;

printf("enter a number:");

scanf("%d",&number);

switch(number)

{

**case 10:**

printf("number is equals to 10");

break;

**case 50:**

printf("number is equal to 50");

break;

**case 100:**

printf("number is equal to 100");

break;

**default:**

printf("number is not equal to 10, 50 or 100");

}

return 0;

}

**Output 1**

enter a number:4

number is not equal to 10, 50 or 100

**Output 2**

enter a number:50

number is equal to 50

**Example 3**

**#include <stdio.h>**

**int main () {**

**/\* local variable definition \*/**

**char grade = 'B';**

**switch(grade) {**

**case 'A' :**

**printf("Excellent!\n" );**

**break;**

**case 'B' :**

**case 'C' :**

**printf("Well done\n" );**

**break;**

**case 'D' :**

**printf("You passed\n" );**

**break;**

**case 'F' :**

**printf("Better try again\n" );**

**break;**

**default :**

**printf("Invalid grade\n" );**

**}**

**printf("Your grade is %c\n", grade );**

**return 0;**

**}**

**Ouput is :**

Well done

Your grade is B

**Nested Switch case Statement**

#include <stdio.h>

**int** main () {

**int** i = 10;

**int** j = 20;

**switch**(i) {

**case** 10:

         printf("the value of i evaluated in outer switch: %d\n",i);

**case** 20:

**switch**(j) {

**case** 20:

               printf("The value of j evaluated in nested switch: %d\n",j);

         }

   }

   printf("Exact value of i is : %d\n", i );

   printf("Exact value of j is : %d\n", j );

**return** 0;

}

**Output**

the value of i evaluated in outer switch: 10

The value of j evaluated in nested switch: 20

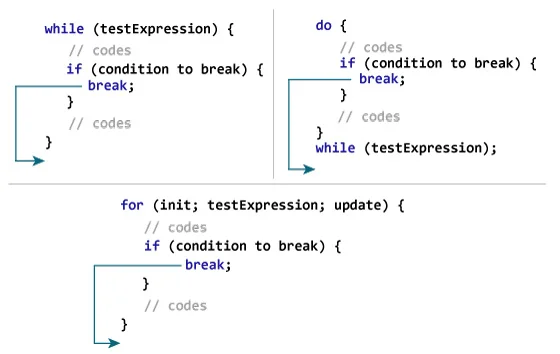
Exact value of i is : 10

Exact value of j is : 20

***Break and Continue***

**Break**

The break statement is used to **terminate** the loop **immediately**, Based on condition.

****

**For Loop Example** : Stop Loop When loop reach number 4

int i;

for (i = 0; i < 10; i++) {

if (i == 4) {

break;

}

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

}

**While Loop Example :**

int i = 0;

while (i < 10) {

if (i == 4) {

break;

}

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

i++;

}

**Output :**

0

1

2

3

**Example 1: break statement**

// Program to calculate the sum of numbers (10 numbers max)

// If the user enters a negative number, the loop terminates

#include <stdio.h>

int main()

{

int i;

double number, sum = 0.0;

for (i = 1; i <= 10; ++i)

{

printf("Enter n%d: ", i);

scanf("%lf", &number);

// if the user enters a negative number, break the loop

if (number < 0.0)

{

break;

}

sum += number; // sum = sum + number;

}

printf("Sum = %.2lf", sum);

return 0;

}

**Output :**

Enter n1: 2.4

Enter n2: 4.5

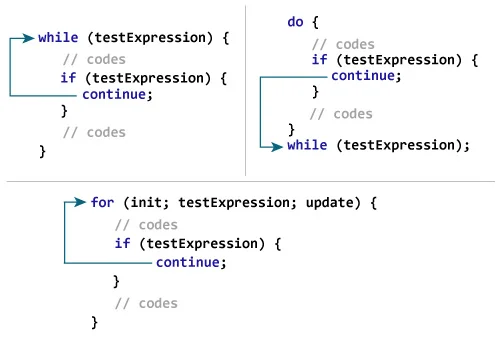
Enter n3: 3.4

Enter n4: -3

Sum = 10.30

**Continuous**

The continue statement is used to **skip** the **current** **iteration** of the loop, based on condition

****

**For Loop Example :** Skip number 4

int i;

for (i = 0; i < 10; i++) {

if (i == 4) {

continue;

}

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

}

**While Loop Example**

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

**Example 2: continue statement**

// Program to calculate the sum of numbers (10 numbers max)

// If the user enters a negative number, it's not added to the result

#include <stdio.h>

int main()

{

int i;

double number, sum = 0.0;

for (i = 1; i <= 10; ++i)

{

printf("Enter a n%d: ", i);

scanf("%lf", &number);

if (number < 0.0)

{

continue;

}

sum += number; // sum = sum + number;

}

printf("Sum = %.2lf", sum);

return 0;

}

**Output :**

Enter n1: 1.1

Enter n2: 2.2

Enter n3: 5.5

Enter n4: 4.4

Enter n5: -3.4

Enter n6: -45.5

Enter n7: 34.5

Enter n8: -4.2

Enter n9: -1000

Enter n10: 12

Sum = 59.70

**CHAPTER – 3 FUNCTIONS AND MODULAR PROGRAMMING**

**1.FUNCTION DEFINITION**

* A function is a block of code that performs a specific task.
* which only runs when it is called.

**Use of Function :**

Dividing a complex problem into smaller chunks makes our program easy to understand and reuse.

reusing code: Define the code once, and use it many times.

**Types of Function:**

* Inbuild Function or Predefined Function
* User Defined Function
* **Predefined Function**
* Standard library functions are **built-in functions.**
* These functions are defined in header files.
* Example : **main()** is a function, which is used to execute code, and **printf()** is a function; used to output/print text to the screen
* **User Defined Function**
* Functions created by the user are known as **user-defined functions.**
* To create A own function, specify the name of the function, followed by parentheses () and curly brackets {}

**2.FUNCTION PROTOTYPES**

FUNCTION **CREATION**

**syntax :**

void myFunction() {  
  // code to be executed  
}

* **myFunction()** is the name of the function
* **void** means that the function does **not** have a **return value.**
* **Inside** the **function**, add code that defines **what the function should do**

**Example :**

**void Hello()**

**{**

**printf("Hello");**

**}**

FUNCTION **CALL**

A **call** function performs a defined task from a user defined function.

**Syntax:**

myFunction();

**Example:**

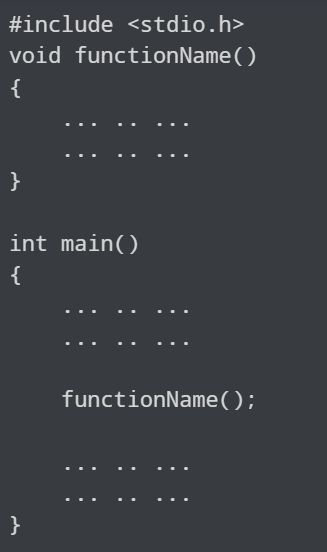
**int main()**

**{**

**Hello(); //Function Call**

**return 0;**

**}**



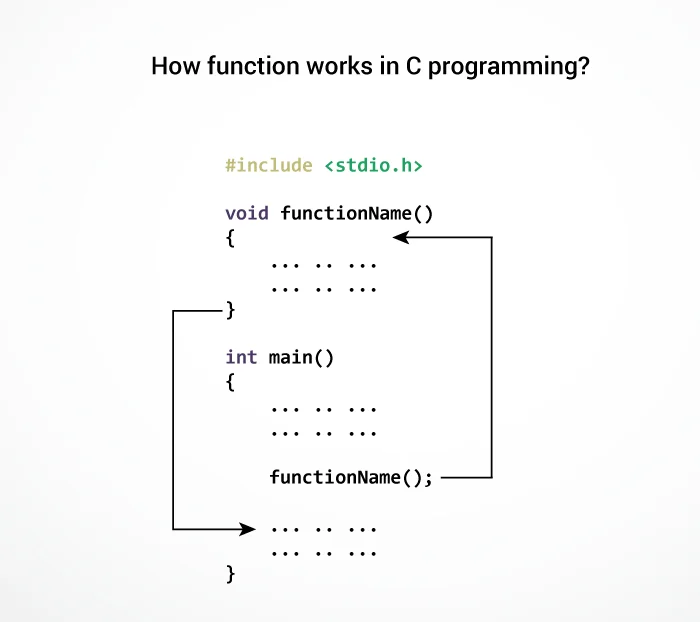
**How user-defined function works?**

* The execution of a program begins from the **main()** function.
* When the compiler encounters myFunction(); control of the program jumps to myFunction()

{

// statement

}

* And, the compiler starts executing the codes inside myFunction().
* The control of the program jumps back to the main() function once code inside the function definition is executed.
* Note, function names are identifiers and should be unique.

**Advantages of user-defined function**

* The program will be easier to understand, maintain and debug.
* Reusable codes that can be used in other programs
* A large program can be divided into smaller modules. Hence, a large project can be divided among many programmers.

**Example 1**

// Create a function  
void myFunction() {  
  printf("I just got executed!");  
}  
  
int main() {  
  **myFunction();** // call the function  
  return 0;  
}  
  
Output:

"I just got executed!"

**Example 2**

void myFunction() {  
  printf("I just got executed!");  
}  
  
int main() {  
  **myFunction();**  
  **myFunction();**  
  **myFunction();**  
  return 0;  
}

**Output:**  
I just got executed!  
I just got executed!  
I just got executed!

**3.FUNCTION PARAMETERS AND RETURN**

**Different Aspects Of Function Calling**

* function without arguments and without return value
* function without arguments and with return value
* function with arguments and without return value
* function with arguments and with return value

**i)Example For Function Without Argument And Return Value**

**Example 1 :**

#include<stdio.h>

**void** printName();

**void** main ()

{

    printf("Hello ");

    printName();

}

**void** printName()

{

    printf("SMV");

}

**Output :**

Hello SMV

**Example 2 :**

#include<stdio.h>

**void** sum();

**void** main()

{

    printf("\nCalculate the sum of two numbers:");

    sum();

}

**void** sum()

{

**int** a,b;

    printf("\nEnter two numbers");

    scanf("%d %d",&a,&b);

    printf("The sum is %d",a+b);

}

**Output :**

Calculate the sum of two numbers:

Enter two numbers 10

24

The sum is 34

**ii)Example For Function Without Argument And With Return Value**

Example 1:

#include<stdio.h>

**int** sum();

**void** main()

{

**int** result;

    printf("\nCalculate the sum of two numbers:");

    result = sum();

    printf("%d",result);

}

**int** sum()

{

**int** a,b;

    printf("\nEnter two numbers");

    scanf("%d %d",&a,&b);

**return** a+b;

}

**Output :**

Calculate the sum of two numbers:

Enter two numbers 10

24

The sum is 34

**Example 2: program to calculate the area of the square**

#include<stdio.h>

**int** sum();

**void** main()

{

    printf(" Calculate the area of the square\n");

**float** area = square();

    printf("The area of the square: %f\n",area);

}

**int** square()

{

**float** side;

    printf("Enter the length of the side in meters: ");

    scanf("%f",&side);

**return** side \* side;

}

**Output :**

Calculate the area of the square

Enter the length of the side in meters: 10

The area of the square: 100.000000

**iii)Example For Function With Argument And Without Return Value**

**Example 1 :**

#include<stdio.h>

**void** sum(**int**, **int**);

**void** main()

{

**int** a,b,result;

    printf("\nCalculate the sum of two numbers:");

    printf("\nEnter two numbers:");

    scanf("%d %d",&a,&b);

    sum(a,b);

}

**void** sum(**int** a, **int** b)

{

    printf("\nThe sum is %d",a+b);

}

**Output :**

Calculate the sum of two numbers:

Enter two numbers 10

24

The sum is 34

**Example 2 :** **Program to calculate the average of five numbers.**

#include<stdio.h>

**void** average(**int**, **int**, **int**, **int**, **int**);

**void** main()

{

**int** a,b,c,d,e;

    printf("\nCalculate the average of five numbers:");

    printf("\nEnter five numbers:");

    scanf("%d %d %d %d %d",&a,&b,&c,&d,&e);

    average(a,b,c,d,e);

}

**void** average(**int** a, **int** b, **int** c, **int** d, **int** e)

{

**float** avg;

    avg = (a+b+c+d+e)/5;

    printf("The average of given five numbers : %f",avg);

}

Output :

Calculate the average of five numbers:

Enter five numbers:10

20

30

40

50

The average of given five numbers : 30.000000

**iv)Example for Function with argument and with return value**

**Example 1:**

#include<stdio.h>

**int** sum(**int**, **int**);

**void** main()

{

**int** a,b,result;

    printf("\nCalculate the sum of two numbers:");

    printf("\nEnter two numbers:");

    scanf("%d %d",&a,&b);

    result = sum(a,b);

    printf("\nThe sum is : %d",result);

}

**int** sum(**int** a, **int** b)

{

**return** a+b;

}

**Output :**

Calculate the sum of two numbers:

Enter two numbers:10

20

The sum is : 30

**Example 2: Program to check whether a number is even or odd**

#include<stdio.h>

**int** even\_odd(**int**);

**void** main()

{

**int** n,flag=0;

 printf("\nGoing to check whether a number is even or odd");

 printf("\nEnter the number: ");

 scanf("%d",&n);

 flag = even\_odd(n);

**if**(flag == 0)

 {

    printf("\nThe number is odd");

 }

**else**

 {

    printf("\nThe number is even");

 }

}

**int** even\_odd(**int** n)

{

**if**(n%2 == 0)

    {

**return** 1;

    }

**else**

    {

**return** 0;

    }

}

**Output :**

Going to check whether a number is even or odd

Enter the number: 100

The number is even

**Modular Programming**

Modular programming is the process of subdividing a program into separate sub-programs.

Ex : when written calculator program then we can write 4 modules(i.e., add, sub, multiply, divide).

Advantage of Modular programming approach

Ease of Use: Easy to handle Thousands and millions lines of code

Reusability: Reuse the functionality with a different interface without typing the whole program again.

Ease of Maintenance: It helps in less collision at the time of working on modules with working on a large application.

Modular Programming File Contains :

* Header file(.h) contains function and type declaration
  + Header file provide the interface to a module.
* Source file(.c) contains concrete implementations

Example : main.c

#include <stdio.h>

#include <stdlib.h>

#include "calc.h"

int main()

{

printf("a+b = %d",4,5,add(4,5));

printf("\na-b = %d",6,5,sub(6,5));

return 0;

}

Calc.c

int add(int a,int b)

{

return a+b;

}

int sub(int a,int b)

{

return a-b;

}

Calc.h

int add(int a,int b);

int sub(int a,int b);

Output :

a+b = 9

a-b = 1

Example 2 : main.c

#include <stdio.h>

#include <stdlib.h>

#include "hello.c"

int main()

{

hello();

return 0;

}

hello.c

hello()

{

int a,b;

scanf("%d%d",&a,&b);

printf("Ans = %d",a+b);

}

Output:

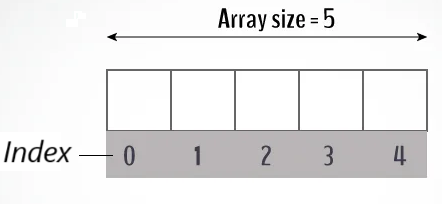
3

4

Ans = 7

**CHAPTER 4 : ARRAYS AND STRINGS**

**What is array ?**

Array is a **collection of elements** of the **same datatype** stored in **contiguous memory locations.** Each **element** in the array is accessed by its **index**. The index starts from **0** for the first element and goes up to **(size - 1)** for an array of size elements.

**Array Declaration**

* + We should specify the datatype of the array
  + Name of the array
  + Required size of the array with in square brackets.

**Syntax** : Single Dimensional Array

**Datatype array\_name[size];**

**Definition of syntax:**

* Datatype can be int,float,double,chat etc.
* Array\_name should be a valid variable name.
* Size is an integer value.
* Note : Size should not be a negative value.

**Example: Array Declaration**

**int age[10];**

**char name[100];**

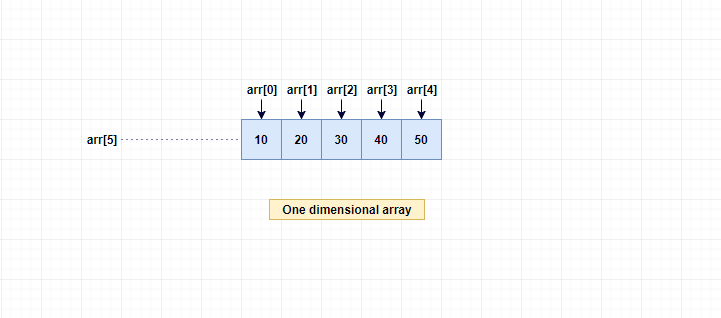
**float weight[5];**

**Initialization of Array**

We can initialize an array while declaring it.

**Example**

**int arr[5] = {10,22,12,42,21};**

****

let the compiler to determine the array size by not giving the initial array size while initializing it.

Note : In this case, array size will be determined by compiler.

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

**Getting input from user**

**#include<stdio.h>**

**int main()**

**{**

**int arr[5];**

**int i;**

**printf("Enter 5 integers\n");**

**for(i = 0; i < 5; i++)**

**{**

**scanf("%d",&arr[i]);**

**}**

**printf("The elements are\n");**

**for(i = 0; i < 5; i++)**

**{**

**printf("%d\n",arr[i]);**

**}**

**getchar();**

**return 0;**

**}**

**Output:**

**Enter 5 Integers**

**6**

**7**

**8**

**9**

**5**

**The elements are**

**6**

**7**

**8**

**9**

**5**

**Array Size:**

The size of an array is fixed upon declaration and cannot be changed during runtime.

**sizeof(myArray):**

* This expression returns the total size (in bytes) of the entire array myArray.
* It calculates the size by multiplying the number of elements in the array by the size of each element.
* For example, if you have an array of integers (int myArray[5]), and on your system, the size of an int is 4 bytes, then sizeof(myArray) would be 5 \* sizeof(int), which might be 20 bytes.

**sizeof(myArray[0]):**

* This expression returns the size (in bytes) of a single element in the array myArray.
* It gives you the size of the data type of the elements in the array.
* Using the same example, if you have an array of integers (int myArray[5]), and on your system, the size of an int is 4 bytes, then sizeof(myArray[0]) would be 4 bytes.

**#include<stdio.h>**

**int main()**

**{**

**int arr[3];**

**int size;**

**size = sizeof(arr);**

**printf("arr Array Entire size is = %d\n",size);**

**size = sizeof(arr[0]);**

**printf("arr Array Single element size is = %d\n",size);**

**getchar();**

**return 0;**

**}**

**Output:**

**arr Array Entire size is = 12**

**arr Array Single element size is = 4**

**Change Value of Array elements**

To change the value of a specific element, refer to the index number

#include <stdio.h>

int main()

{

int myNumbers[] = {25, 50, 75, 100};

myNumbers[0] = 33;

printf("%d", myNumbers[0]);

return 0;

}

Output : 33

**MULTIDIMENSIONAL ARRAY**

A multidimensional array is basically an array of arrays.

**Two-Dimensional Arrays**

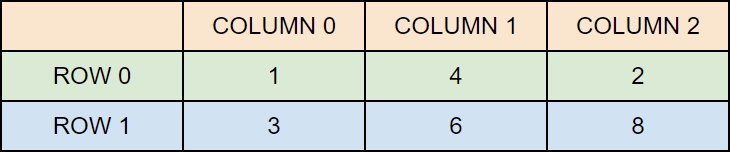
A 2D array is also known as a matrix (a table of rows and columns).

Syntax :

**Array\_DataType array\_Name[3][4];**

int matrix[2][3] = { {1, 4, 2}, {3, 6, 8} };

The first dimension represents the number of rows**[2]**, while the second dimension represents the number of columns**[3]**. The values are placed in row-order, and can be visualized like this:

****

**Access the Elements of a 2D Array**

To access an element of a two-dimensional array, you must specify the index number of both the row and column.

This statement accesses the value of the element in the first row (0) and third column (2) of the matrix array.

#include<stdio.h>

Int main()

{

int matrix[2][3] = { {1, 4, 2}, {3, 6, 8} };  
  
printf("%d", matrix[0][2]);  // Outputs 2

return 0;

}

Output: 2

**Change Elements in a 2D Array**

To change the value of an element, refer to the index number of the element in each of the dimensions:

The following example will change the value of the element in the first row (0) and first column (0):

#include<stdio.h>

int main()

{

int matrix[2][3] = { {1, 4, 2}, {3, 6, 8} };  
matrix[0][0] = 9;  
printf("%d", matrix[0][0]);  // Now outputs 9 instead of 1

return 0;

}

Output : 9

**Loop Through a 2D Array**

Print 2D Array Values using Loop

#include<stdio.h>

int main()

{

int matrix[2][3] = { {1, 4, 2}, {3, 6, 8} };  
  
int i, j;  
for (i = 0; i < 2; i++) {  
  for (j = 0; j < 3; j++) {  
    printf("%d\n", matrix[i][j]);  
  }  
}

return 0;

}

Output:

1

4

2

3

6

8

**Practice Array**

* 1. Write a program in C to store elements in an array and print them.

#include <stdio.h>

void main()

{

int arr[10];

int i;

printf("Input 10 elements in the array :\n");

for(i=0; i<10; i++)

{

printf("element - %d : ",i);

scanf("%d", &arr[i]);

}

printf("\nElements in array are: ");

for(i=0; i<10; i++)

{

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

}

printf("\n");

}

Output:

Input 10 elements in the array :

element - 0 : 1

element - 1 : 1

element - 2 : 2

element - 3 : 3

element - 4 : 4

element - 5 : 5

element - 6 : 6

element - 7 : 7

element - 8 : 8

element - 9 : 9

Elements in array are: 1 1 2 3 4 5 6 7 8 9

* 1. Write a program in C to read n number of values in an array and display them in reverse order.

#include <stdio.h>

void main()

{

int i,n,a[100];

printf("Input the number of elements to store in the array :");

scanf("%d",&n);

printf("Input %d number of elements in the array :\n",n);

for(i=0;i<n;i++)

{

printf("element - %d : ",i);

scanf("%d",&a[i]);

}

printf("\nThe values store into the array are : \n");

for(i=0;i<n;i++)

{

printf("% 5d",a[i]);

}

printf("\n\nThe values store into the array in reverse are :\n");

for(i=n-1;i>=0;i--)

{

printf("% 5d",a[i]);

}

printf("\n\n");

}

Output:

Input the number of elements to store in the array :3

Input 3 number of elements in the array :

element - 0 : 2

element - 1 : 5

element - 2 : 7

The values store into the array are :

2 5 7

The values store into the array in reverse are :

7 5 2

* 1. Write a program in C to find the sum of all elements of an array.

#include <stdio.h>

void main()

{

int a[100];

int i, n, sum=0;

printf("\n\nFind sum of all elements of array:\n");

printf("--------------------------------------\n");

printf("Input the number of elements to be stored in the array :");

scanf("%d",&n);

printf("Input %d elements in the array :\n",n);

for(i=0;i<n;i++)

{

printf("element - %d : ",i);

scanf("%d",&a[i]);

}

for(i=0; i<n; i++)

{

sum += a[i];

}

printf("Sum of all elements stored in the array is : %d\n\n", sum);

}

Output:

Find sum of all elements of array:

--------------------------------------

Input the number of elements to be stored in the array :3

Input 3 elements in the array :

element - 0 : 2

element - 1 : 5

element - 2 : 8

Sum of all elements stored in the array is : 15

* 1. Write a program in C to count the total number of duplicate elements in an array.

#include <stdio.h>

int main()

{

int arr[100];

int n,mm=1,ctr=0;

int i, j;

printf("Input the number of elements to be stored in the array :");

scanf("%d",&n);

printf("Input %d elements in the array :\n",n);

for(i=0;i<n;i++)

{

printf("element - %d : ",i);

scanf("%d",&arr[i]);

}

for (i = 0; i < n; i++)

{

for (j = i + 1; j < n; j++)

{

if (arr[i] == arr[j])

{

ctr++;

break;

}

}

}

printf("Total number of duplicate elements found in the array: %d\n", ctr);

return 0;

}

Output:

Input the number of elements to be stored in the array :5

Input 5 elements in the array :

element - 0 : 1

element - 1 : 1

element - 2 : 2

element - 3 : 3

element - 4 : 3

Total number of duplicate elements found in the array: 2

* 1. Write a program in C to print all unique elements in an array.

#include <stdio.h>

int main()

{

int arr1[100], n,ctr=0;

int i, j, k;

printf("\n\nPrint all unique elements of an array:\n");

printf("------------------------------------------\n");

printf("Input the number of elements to be stored in the array: ");

scanf("%d",&n);

printf("Input %d elements in the array :\n",n);

for(i=0;i<n;i++)

{

printf("element - %d : ",i);

scanf("%d",&arr1[i]);

}

printf("\nThe unique elements found in the array are: \n");

for(i=0; i<n; i++)

{

ctr=0;

for(j=0,k=n; j<k+1; j++)

{

/\*Increment the counter when the seaarch value is duplicate.\*/

if (i!=j)

{

if(arr1[i]==arr1[j])

{

ctr++;

}

}

}

if(ctr==0)

{

printf("%d ",arr1[i]);

}

}

printf("\n\n");

}

Output:

Print all unique elements of an array:

------------------------------------------

Input the number of elements to be stored in the array: 4

Input 4 elements in the array :

element - 0 : 3

element - 1 : 2

element - 2 : 2

element - 3 : 5

The unique elements found in the array are:

3 5

**STRING**

**What is String?**

* String is a array of Characters (or) String are Collection of Character
* Ended with Null Character (or) Each String Ended with \0.
* ASCII Value of Null Character is 0
* The C String is stored as an array of characters

**STRING DECLARATION**

**Syntax:**

**char *string\_name*[*size*];**

Note : [size] means number of characters strings will store

**Example:**

char str[10];

**Syntax Explanation:**

**char**

**str**

**[10]**

**;**

**End of the Statement**

**Size of String**

**Variable Name**

**Datatype**

**STRING INITIALIZATION**

**Syntax:**

char variable\_name[r] = {list of string};

**Example:**

* 1. Assigning a String Literal without Size

char str\_1[] = “Programming”;

* 1. Assigning a String Literal with a Predefined Size

char str\_2[15] = “Programming”;

3. Assigning Character by Character with Size

char str\_3[11] = {‘P’, ’r’, ’o’, ’g’, ’r’, ’a’, ’m’, ’m’, ’i’, ’n’, ’g’};

4. Assigning Character by Character without size

char str[] = { ‘P’, ’r’, ’o’, ’g’, ’r’, ’a’, ’m’, ’m’, ’i’, ’n’, ’g’};

**Don’t Use Like this : Assigning Values to Strings**

Arrays and Strings do not support the assignment operator once it is declared.

**For example,**

char c[100];

c = "C programming"; // Error! array type is not assignable.

**Read String From The User**

* 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>

int main()

{

char name[20];

printf("Enter name: ");

scanf("%s", name);

printf("Your name is %s.", name);

return 0;

}

**Output 1:**

Enter name: SMV

Your name is SMV.

**Output 2 :**

Enter name: SMV Computer Center

Your name is SMV

//In this case SMV only print remaining words not printing.

Why because whitespace equals to ‘\0’ . It means terminate sentence.

**Example programming for ‘\0’**

#include<stdio.h>

int main()

{

printf("Program\0World");

return 0;

}

**Output**

Program

**Example 2**

#include<stdio.h>

int main()

{

char text[] = {'H','e','l','l','o','\0','W','o','r','l','d'};

printf("%s",text);

return 0;

}

**Output**

Hello

**How to print Full Sentence Include Whitespace**

**Example:**

#include <stdio.h>

#include <string.h>

int main() {

char str1[20];

printf("Enter Sentace = ");

scanf("%[^\n]s",&str1);

printf("%s",str1);

return 0;

}

**Output**

Enter Sentence = SMV Computer Center

SMV Computer Center

**Practice String**

1.Write a program to get string from user and print it

#include<stdio.h>

int main()

{

char myStr[20];

printf("Enter Your Name : ");

scanf("%[^\n]s ",&myStr);

printf("Hi %s",myStr);

return 0;

}

Output:

Enter your Name: Lenovo

Hi Lenovo

**BASIC STRING FUNCTIONS**

* 1. **Strlen() Function**

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

**Example 1**

#include<stdio.h>

#include <string.h>

int main(){

char ch[20]={'C', 'P', 'r', 'o', 'g', 'r', 'a', 'm', '\0'};

printf("Length of string is: %d",strlen(ch));

return 0;

}

**Output**

Length of string is: 8

**Example 2**

#include<stdio.h>

#include <string.h>

int main(){

char ch[20]={"CProgram\0"};

printf("Length of string is: %d",strlen(ch));

return 0;

}

**Output**

Length of string is: 8

**Example 3 (Method 1)**

#include<stdio.h>

#include <string.h>

int main()

{

char str[] = "SMVComputerCenter";

// Calculate the length of the string using the strlen()

// function and store it in the variable 'length'

size\_t length = strlen(str);

// Print the length of the string

printf("String: %s\n", str);

printf("Length: %zu\n", length);

return 0;

}

**Output**

String : SMVComputerCenter

Length: 17

**[OR]**

**Example 3 (Method 2)**

#include<stdio.h>

#include <string.h>

int main()

{

char str[] = "SMVComputerCenter";

// Calculate the length of the string using the strlen()

// function and store it in the variable 'length'

**int length = strlen(str);**

// Print the length of the string

printf("String: %s\n", str);

**printf("Length: %d\n", length);**

return 0;

}

**Output**

String : SMVComputerCenter

Length: 17

* 1. **strcpy() Function**

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

**Example 1**

#include<stdio.h>

#include <string.h>

int main(){

char ch[20]={'C', 'P', 'r', 'o', 'g', 'r', 'a', 'm', '\0'};

char ch2[20];

strcpy(ch2,ch);

printf("Value of second string is: %s",ch2);

return 0;

}

**Output**

Value of second string is: CProgram

**Example 2**

#include<stdio.h>

#include <string.h>

int main(){

char ch[20]={“CProgram”};

char ch2[20];

strcpy(ch2,ch);

printf("Value of second string is: %s",ch2);

return 0;

}

**Output**

Value of second string is: CProgram

* 1. **strcat() Function**

The strcat(first\_string, second\_string) function concatenates two strings and result is returned to first\_string.

**Example 1**

#include<stdio.h>

#include <string.h>

int main(){

char ch[10]={'h', 'e', 'l', 'l', 'o', '\0'};

char ch2[10]={'c', '\0'};

strcat(ch,ch2);

printf("Value of first string is: %s",ch);

return 0;

}

**Output:**

Value of first string is: helloc

**Example 2**

#include<stdio.h>

#include <string.h>

int main(){

char ch[10]={"Hello "};

char ch2[10]={"C"};

strcat(ch,ch2);

printf("Value of first string is: %s",ch);

return 0;

}

**Output**

Value of first string is: Hello C

* 1. **strcmp() Function**

The strcmp(first\_string, second\_string) function compares two string and returns 0 if both strings are equal. Otherwise, it retrun positive and negative value based on string input.

**Possible return values from the strcmp() function**

|  |  |
| --- | --- |
| **Return Value** | **Description** |
| 0 | When both the strings are **equal.** |
| <0 | If the ASCII value of a character of the **first string** is **less than** the ASCII value of a character of the **second string**, then the function will return **negative value.** |
| >0 | If the ASCII value of a character of the **first string** is **greater than** the ASCII value of a character of the **second string**, then the function will return **positive value.** |

**Example 1**

#include<stdio.h>

#include <string.h>

int main(){

char str1[20],str2[20];

printf("Enter 1st string: ");

gets(str1);//reads string from console

printf("Enter 2nd string: ");

gets(str2);

printf("%d",strcmp(str1,str2));

return 0;

}

**Output:**

Enter 1st string: C Program

Enter 2nd string: C Program

0

**Example 2**

**#include<stdio.h>**

**#include<string.h>**

**int** main()

{

//ASCI value of A is less than P

**char** str1[] = "PrepBytes";

**char** str2[] = "ArepBytes";

**int** ans = strcmp(str1, str2);

**if** (ans==0)

printf("Strings are equal");

**else**

printf("Strings are unequal");

printf("\nValue of result: %d" , ans);

**return** 0;

}

**Output**

Strings are unequal

Value of result: 1

**Example 3**

**#include<stdio.h>**

**#include<string.h>**

**int** main()

{

//ASCI value of Z is more than P

**char** str1[] = "PrepBytes";

**char** str2[] = "ZrepBytes";

**int** ans = strcmp(str1, str2);

**if** (ans==0)

printf("Strings are equal");

**else**

printf("Strings are unequal");

printf("\nValue of result: %d" , ans);

**return** 0;

}

**Output**

Strings are unequal

Value of result: -1

**Example 4**

#include<stdio.h>

#include <string.h>

int main(){

char str1[20],str2[20];

printf("Enter 1st string: ");

gets(str1);//reads string from console

printf("Enter 2nd string: ");

gets(str2);

if(strcmp(str1,str2)==0)

printf("Strings are equal");

else

printf("Strings are not equal");

return 0;

}

**Output Model 1:**

Enter 1st string: C Program

Enter 2nd string: C Program

Strings are equal

**Output Model 2:**

Enter 1st string: C Program

Enter 2nd string: c program

Strings are not equal

* 1. **strrev() Function**

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

**Example 1**

#include<stdio.h>

#include <string.h>

int main(){

char str[20];

printf("Enter string: ");

gets(str);//reads string from console

printf("String is: %s",str);

printf("\nReverse String is: %s",strrev(str));

return 0;

}

**Output:**

Enter string: C Program

String is: C Program

Reverse String is: margorP C

* 1. **strlwr() Function**

The strlwr(string) function returns string characters in lowercase.

**Example 1**

#include<stdio.h>

#include <string.h>

int main(){

char str[20];

printf("Enter string Capital Letter: ");

gets(str);//reads string from console

printf("String is: %s",str);

printf("\nLower String is: %s",strlwr(str));

return 0;

}

**Output:**

Enter string Capital Letter: C PROGRAM

String is: C PROGRAM

Lower String is: c program

* 1. **strupr() Function**

The strupr(string) function returns string characters in uppercase.

**Example1**

#include<stdio.h>

#include <string.h>

int main(){

char str[20];

printf("Enter string Small Letter: ");

gets(str);//reads string from console

printf("String is: %s",str);

printf("\nUpper String is: %s",strupr(str));

return 0;

}

**Output:**

Enter string Small Letter: c program

String is: c program

Upper String is: C PROGRAM

**CHAPTER 5 : POINTERS AND DYNAMIC MEMORY ALLOCATION**

**UNDERSTANDING POINTERS**

Two types of Variables,

* Data Variables – Data Variables used for store values in memory
* Pointer Variables – Pointer Variables used for store memory address of Variable.

**What is Pointer?**

* + Pointer variables are use to store the addresses.
  + Pointer variables which holds the address of similar type of data.
  + Pointer is like an integer which holds a number.
  + Pointer variable is declared with their type.
  + Size of the pointer is equal to size of int. (According to type of processer).

**Why pointers?**

**Using pointers, we can directly communicate with the hardware. This is very useful in embedded programming. In embedded programming, we read and write data into hardware address** whereas normal c programming we read data from the user and we will show output to the user.

**Pointer Definition: -**

A pointer can be used to store the **memory address**of other variables, functions, or even other pointers.

**Pointer Declaration: -**

Declare a pointer, we use the **( \* ) dereference operator** before its name.

Syntax:

**Datatype \* ptr;**

Where,

Ptr is the name of the pointer.

Datatype is the type of data it is pointing to.

Example

int \*ptr;

float \*ptr1;

char \*ptr2;

**Pointer Initialization: -**

Pointer initialization is the process where we **assign** some **initial value** to **the pointer variable**. We generally use the **( & ) addressof operator** to **get** the **memory** **address** of a variable and then **store** it in the **pointer variable.**

Example:

int Rupee = 10;

int \*ptr;

ptr = &Rupee;

[or]

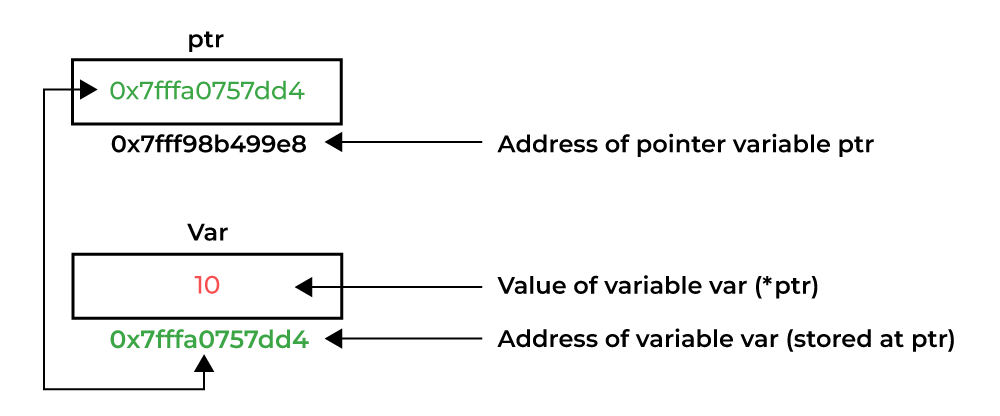
Pointer Definition – The pointer is declared and initialized at the same time.

Example :

int var = 10;

int \*ptr = &var;

**Pointer Dereferencing**

Dereferencing a pointer is the process of accessing the value stored in the memory address specified in the pointer. We use the same **( \* ) dereferencing operator** that we used in the pointer declaration.

**asterisk \*** operator to get the value stored in an address.

**Finding Address of The Variable**

#include <stdio.h>

int main()

{

  int var = 5;

  printf("var: %d\n", var);

  printf("address of var: %d", &var);

  return 0;

}

**Output:**

var: 5

address of var: 6422300

**Example 1:**

#include<stdio.h>

int main()

{

    //Variable Initialization

    int a = 10;

    //Pointer Initialization

    int \*p;

    //Pointer Declaration

    p = &a;

    printf("\nValue of A : %d",a);

    printf("\nAddress of A : %d",&a);

    printf("\nValue of P : %d",p);

    printf("\nAddress of P : %d",&p);

    //Pointer Dereferencing

    printf("\nP Dereferencing  : %d",\*p);

    return 0;

}

**Output:**

Value of A : 10

Address of A : 1032248460

Value of P : 1032248460

Address of P : 1032248448

P Dereferencing : 10

**Example 2:**

Note: - %p format specifier, it is  used to print the pointer type data

#include<stdio.h>

main() {

   int x = 50;

   int \*ptr = &x;

   printf("The address is: %p, the value is %d", ptr, \*ptr);

}

**Output**:

The address is: 0061FF18, the value is 50

**Example 3:**

#include <stdio.h>

int main()

{

   int\* pc, c;

   c = 22;

   printf("Address of c: %p\n", &c);

   printf("Value of c: %d\n\n", c);  // 22

   pc = &c;

   printf("Address of pointer pc: %p\n", pc);

   printf("Content of pointer pc: %d\n\n", \*pc); // 22

   c = 11;

   printf("Address of pointer pc: %p\n", pc);

   printf("Content of pointer pc: %d\n\n", \*pc); // 11

   \*pc = 2;

   printf("Address of c: %p\n", &c);

   printf("Value of c: %d\n\n", c); // 2

   return 0;

}

**Output:**

Address of c: 0061FF18

Value of c: 22

Address of pointer pc: 0061FF18

Content of pointer pc: 22

Address of pointer pc: 0061FF18

Content of pointer pc: 11

Address of c: 0061FF18

Value of c: 2

**CLASSIFICATION OF POINTER**

According to the Datatype pointers are classified into five types

* 1. Integer pointer
  2. float pointer
  3. String pointer
  4. Void pointer
  5. Null pointer

According to the pointer used in a program it divided into three types

1. Pointer to an array
2. Pointer to a function
3. Pointer to a structure

**Integer pointer**

Integer pointer accepts integer values and it allocates memory based on system architecture.

int a = 10,\*ptr;

[or]

int a = 10;

int \*ptr;

Example – 1 - Pointer using %p (Prints Memory Address)

**Note : %p is a format specifier , %p is used to printing the address of a pointer**

#include<stdio.h>

int main()

{

    int a,\*ptr;

    ptr = &a;

    printf("A Value = %d",a);

    printf("\nptr Dereference = %d",\*ptr);

    printf("\nA Address = %p",&a);

    printf("\nptr Value = %p",ptr);

    printf("\nptr Address = %p",&ptr);

    printf("\n------------------------------------");

    a = 10;

    ptr = &a;

    printf("\nA Value = %d",a);

    printf("\nptr Dereference = %d",\*ptr);

    printf("\nA Address = %p",&a);

    printf("\nptr Value = %p",ptr);

    printf("\nptr Address = %p",&ptr);

    return 0;

}

**Output**

A Value = 3776512

ptr Dereference = 3776512

A Address = 0061FF1C

ptr Value = 0061FF1C

ptr Address = 0061FF18

------------------------------------

A Value = 10

ptr Dereference = 10

A Address = 0061FF1C

ptr Value = 0061FF1C

ptr Address = 0061FF18

Example : 2 – Pointer using %u (unsigned integer)

Note:- **%u** is an unsigned integer format specifier, An unsigned Integer means the variable can hold only a positive value.

#include<stdio.h>

int main()

{

    int a,\*ptr;

    ptr = &a;

    printf("A Value = %d",a);

    printf("\nptr Dereference = %d",\*ptr);

    printf("\nA Address = %u",&a);

    printf("\nptr Value = %u",ptr);

    printf("\nptr Address = %u",&ptr);

    printf("\n------------------------------------");

    a = 10;

    ptr = &a;

    printf("\nA Value = %d",a);

    printf("\nptr Dereference = %d",\*ptr);

    printf("\nA Address = %u",&a);

    printf("\nptr Value = %u",ptr);

    printf("\nptr Address = %u",&ptr);

    return 0;

}

**Output**

A Value = 2961408

ptr Dereference = 2961408

A Address = 6422300

ptr Value = 6422300

ptr Address = 6422296

------------------------------------

A Value = 10

ptr Dereference = 10

A Address = 6422300

ptr Value = 6422300

ptr Address = 6422296

**Float Pointer**

A float pointer only stores an address of a float variable.

float fv = 3.14,\*ptr;

[or]

float fv = 3.14;

float \*ptr;

Example 1:

#include<stdio.h>

int main()

{

    float var = 3.1415,\*ptr;

    ptr = &var;

    printf("Address of var = %p\n",&var);

    printf("ptr is pointing to an address  %p\n",ptr);

    printf("Value stored at ptr = %f",\*ptr);

    return 0;

}

**Output:**

Address of var = 0061FF18

ptr is pointing to an address  0061FF18

Value stored at ptr = 3.141500

**String Pointer**

#include<stdio.h>

int main()

{

    char str[6] = "Hello";

    char \*ptr = str;

    printf("String Using Array = %s", str);

    printf("\nString Pointer = %s", ptr);

    return 0;

}

**Output**

String Using Array = Hello

String Pointer = Hello

**Note: - why using char \*ptr = str; instead of char \*ptr = &str;**

In C, an array name represents the **address of the first element of the array**. When you declare a character array like char str[6] = "Hello";, the **variable str itself represents the address of the first character ('H' in this case).**

So, when you declare a pointer and assign it the value of the array, you don't need to use the address-of operator (&). Therefore, **char \*ptr = str; is correct because str is already a pointer to the first element of the array.**

On the other hand, char \*ptr = &str; would be incorrect because it would make ptr point to the entire array (the address of the array itself), not just the first element. This would result in issues when trying to use ptr as a string because it would include the null terminator and potentially garbage values after the string.

**For Example:**

#include<stdio.h>

int main()

{

    char str[6] = "Hello";

    // Using the array name directly as a pointer

    char \*ptr\_array = str;

    // Using the address-of operator to get the pointer

    char \*ptr\_address = &str[0];

    printf("String Using Array = %s\n", str);

    printf("String Using Array Pointer = %s\n", ptr\_array);

    printf("String Using Address-of Operator = %s\n", ptr\_address);

    return 0;

}

**Output**

String Using Array = Hello

String Using Array Pointer = Hello

String Using Address-of Operator = Hello

**How to Print address of String Variable and Pointer**

#include<stdio.h>

int main()

{

    char str[6] = "Hello";

    // Using the array name directly as a pointer

    char \*ptr\_array = str;

    // Print the address of the array

    printf("Address of str: %p\n", (void \*)str); //Type Casting

    printf("Address of ptr\_array: %p\n", (void \*)ptr\_array); //Type Casting

    return 0;

}

**Output**

Address of str: 0061FF16

Address of ptr\_array: 0061FF16

**Why Type casting Need here :** (void \*)str , (void \*)ptr\_array);

The **(void \*)** type cast is used when printing addresses to match the **%p** specifier's expectations.

**Example : 2 Assign String Using pointers**

Note: - By using array there is chance to wastage of memory and not expandable, But by using pointer it occupies many character without range and no memory wastage.

#include<stdio.h>

int main()

{

    char a[] = "hello from pointer";

    char \*ptr = "hello from pointer";

    printf("string using array = %s\n",a);

    printf("string using pointer = %s\n",ptr);

    return 0;

}

**Output:**

string using array = hello from pointer

string using pointer = hello from pointer

**Void Pointer**

Void pointer is a pointer which has no associated datatype with it.

It can point to any data of any datatype and can be typecasted to any type.

Example :

#include<stdio.h>

int main()

{

    int n = 10;

    void \*ptr = &n;

    printf("%d",\*(int\*)ptr); //Type Casting

    return 0;

}

Output: 10

**Use of Void Pointer**

**malloc** and **calloc** function returns a void pointer. Due to this reason, they can allocate a memory for any type of data.

**NULL** **Pointers**

A NULL Pointer is a pointer that does not point to any memory location. It represents an invalid memory location.

When a NULL value is assigned to a pointer, then the pointer is considered as NULL Pointer.

Example:

#include<stdio.h>

int main()

{

    int \*ptr = NULL;

    return 0;

}

**Use of NULL Pointer**

It is used to initialize a pointer when that pointer isn’t assigned any valid memory address yet.

It is Useful for handling errors when using malloc function.

**Fact about NULL Pointer**

1.The value of NULL is 0. We can either use NULL or 0 but this 0 is written in context of pointers and is not equivalent to the integer 0.

Example:

#include<stdio.h>

int main()

{

    int \*ptr = NULL;

    printf("%d",ptr);

    return 0;

}

Output : 0

2.Size of the NULL Pointer depends upon the platform and is similar to the size of the normal pointers.

Example :

#include<stdio.h>

int main()

{

    printf("Size of NULL = %d",sizeof(NULL));

    return 0;

}

Output: Size of NULL = 4

**1.Pointer to an array**

* Pointers to an array **points** the **address** of **memory block of an array variable**.
* Arrays names act as pointers to their starting points.
* if a pointer variable stores the base address of an array then we can manipulate all the array elements using the pointer variable only.
* We can create an array of pointers to store multiple addresses of different variables.

Example 1 :

// Array of pointers to store multiple addresses of different variables.

#include <stdio.h>

int main() {

    // Declare some variables

    int a = 10;

    float b = 3.14;

    char c = 'A';

    // Declare an array of pointers

    void \*ptrArray[3];

    // Assign addresses of variables to the array

    ptrArray[0] = &a;

    ptrArray[1] = &b;

    ptrArray[2] = &c;

    // Access and print the values using the pointers

    printf("Value of a: %d\n", \*(int \*)ptrArray[0]);

    printf("Value of b: %f\n", \*(float \*)ptrArray[1]);

    printf("Value of c: %c\n", \*(char \*)ptrArray[2]);

    return 0;

}

**Output:**

Value of a: 10

Value of b: 3.140000

Value of c: A

Definition : An array name is generally treated as a pointer to the first element of the array and if we store the base address of the array in another pointer variable, then we can easily manipulate the array using pointer arithmetic in a C Program.

Example : 2

#include <stdio.h>

int main()

{

  // array declaration and initialization

  int arr[5] = {2, 4, 6, 8, 10}, i;

  for(i = 0; i < 5; i++)

  {

    // printing the elements address and value at

    // arr[i] using \*(arr + i) syntax

    printf("[index %d] Address : %u, Value : %d\n", i, (arr + i), \*(arr + i));

  }

  return 0;

}

Output:

[index 0] Address : 6422280, Value : 2

[index 1] Address : 6422284, Value : 4

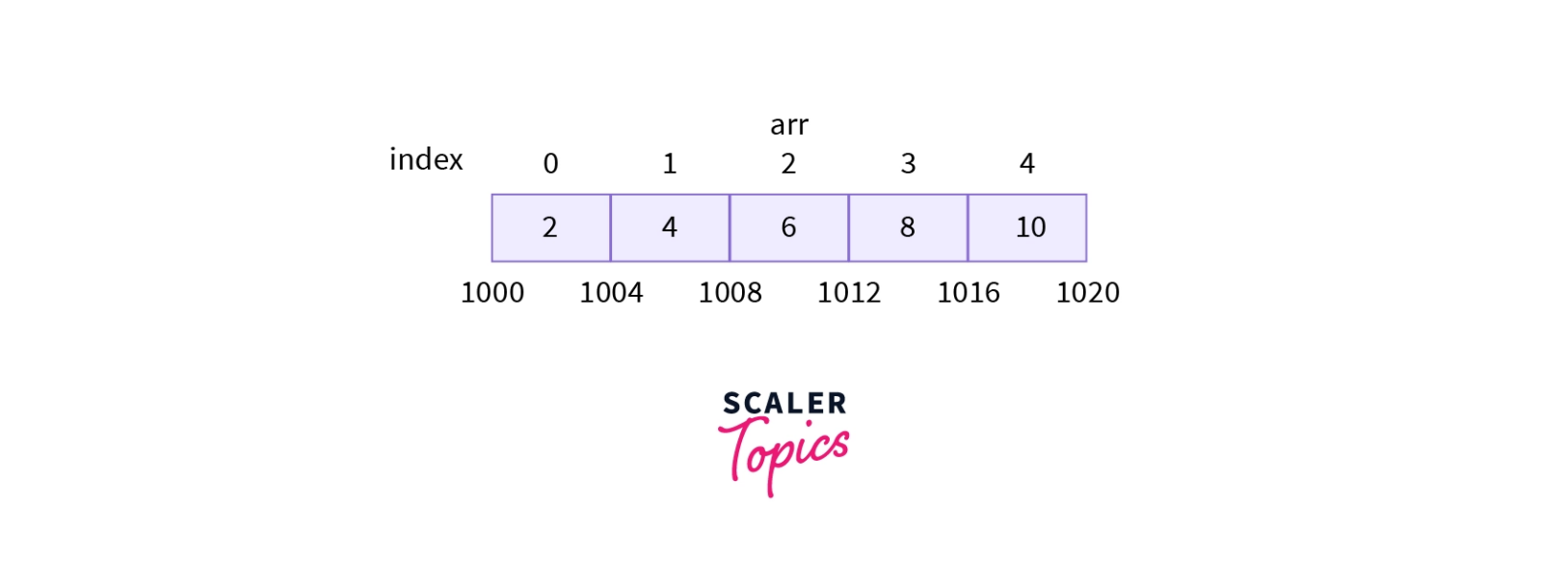
[index 2] Address : 6422288, Value : 6

[index 3] Address : 6422292, Value : 8

[index 4] Address : 6422296, Value : 10

**Explanation :**

* We have declared and initialized an integer array **arr**, array representation



* **(arr + i)** represents the address of the value at index **i**, so **\*(arr + i)** will give the value at ith index **(address(arr + i) = address(arr[i])),** it is used to print the addresses of the array elements as the value of **i** changes from **0-4.**
* **\*** is a dereferencing operator used for printing the value at the provided address. **\*(arr + i)** will print the values of the array at consecutive addresses as the value of i changes from 0-4.

**Array of Pointers to Character**

Major **use** case of Array of pointers is while **storing multiple strings.**

**Syntax:**

char \*array\_name[size];

**Example:**

#include<stdio.h>

int main() {

    // Declaring an array of pointers to characters

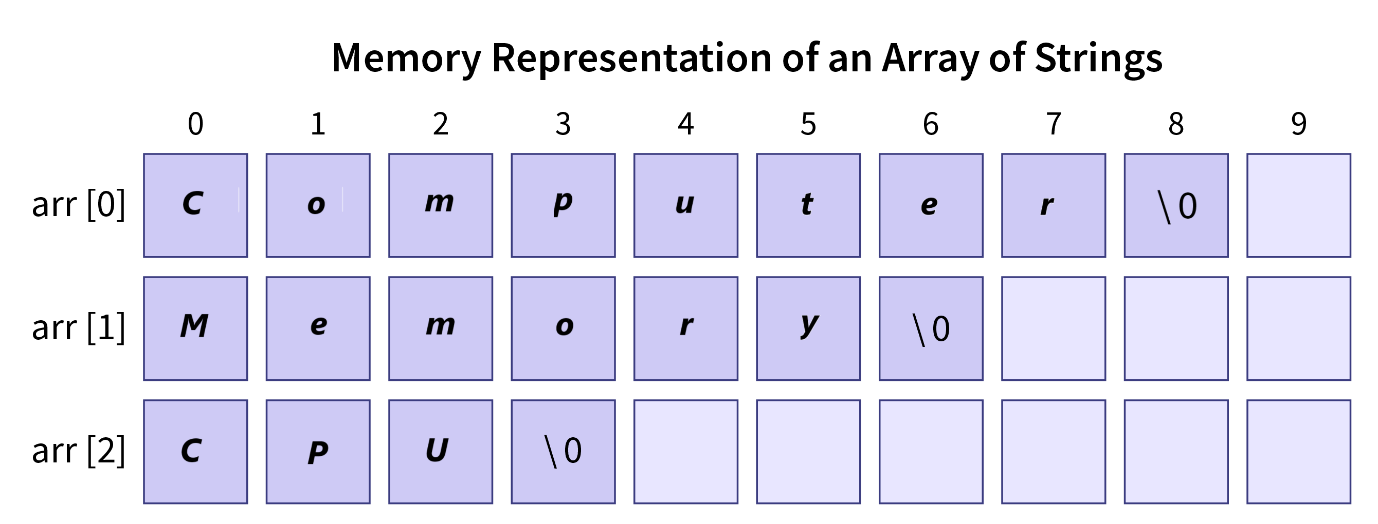
    char \*names[] = {"Computer", "Memory", "CPU"};

    // Accessing and printing the second name

    printf("%s\n", names[1]);  // Outputs: Jane

    return 0;

}

**Output: Memory**

**Pointer to a Function**

A pointer pointing to the address of a function is called **function pointer.**

**Example:**

**//** how functions have a memory address like variables

#include <stdio.h>

void test() {

    // test function that does nothing

    return ;

}

int main() {

    int a = 5;

    // printing the address of variable a

    printf("Address of variable = %p\n", &a);

    // printing the address of function main()

    printf("Address of a function = %p",test);

    return 0;

}

**Output:**

Address of variable = 0061FF1C

Address of a function = 00401460

**Example : 1**

//Pointer To a Function

#include <stdio.h>

void msg();

void msg() {

   printf("This Sentence Print Using Pointer");

}

int main() {

    void (\*funptr)();

    funptr = msg;

    funptr();

    return 0;

}

**Output:** This Sentence Print Using Pointer

**Example : 2**

//Pointer To a Function with Parameters [Print Sentence N Number of Times]

#include <stdio.h>

void msg();

void msg(int num\_times) {

    int n;

    for(n=1;n<=num\_times;n++)

    {

           printf("\nRoll Number : %d",n);

    }

}

int main() {

    void (\*funptr)(int);

    funptr = msg;

    funptr(5);

    return 0;

}

**Output:**

Roll Number : 1

Roll Number : 2

Roll Number : 3

Roll Number : 4

Roll Number : 5

**Example 3:**

#include<stdio.h>

int addition ();

int main ()

{

    int result;

    int (\*ptr)();

    ptr = &addition;

    result = (\*ptr)();

    printf("The sum is %d",result);

}

int addition()

{

    int a, b;

    printf("Enter two numbers?");

    scanf("%d %d",&a,&b);

    return a+b;

}

**Output:**

Enter two numbers?4 5

The sum is 9

**Pointer to a Structure**

* **Structure pointer** points to the address of the structure variable in the memory block.
* This pointer can be used to access and change the value of structure members.
* This way, structures and pointers in C can be used to create and access user-defined data types conveniently.

**Syntax to Define a Structure**

struct structure\_name {

// Here define different **members** of the structure

    data\_type member\_variable\_1;

    data\_type member\_variable\_2;

    // ...

    data\_type member\_variable\_N;

};

**Note: - struct**keyword is used to create a new data type

**Syntax to Define a Structure Pointer**

* Structure pointer in C is declared using the keyword struct followed by structure name to which the pointer will point to followed by pointer name.
* A structure pointer can only hold the address of a variable of the same structure type.

struct structure\_name \*structure\_pointer;

**Initialization of Structure Pointer**

**structure\_pointer = &structure\_variable;**

Also, the structure pointer can be initialized during the time of declaration.

struct structure\_type \*structure\_pointer = &structure\_variable;

**Accessing Structure Member Using Pointer**

There are two ways to access the values of structure members using pointers -

**1.** Using asterisk (\*) and dot (.) operator with the structure pointer.

**2.** Using membership or arrow (->) operator.

**Example : 1**

//Structure pointer Access Using asterisk and dot

#include<stdio.h>

struct emp\_detail

{

    int empid;

    char empname[20];

    float salary;

};

int main()

{

    struct emp\_detail emp\_1 = {101,"Hansi",50000};

    struct emp\_detail \*ptr;

    ptr = &emp\_1;

    printf("Employee ID : %d\n",(\*ptr).empid);

    printf("Employee Name : %s\n",(\*ptr).empname);

    printf("Empoyee Salary : %.2f",(\*ptr).salary);

    return 0;

}

Output:

Employee ID : 101

Employee Name : Hansi

Empoyee Salary : 50000.00

**Example : 2**

//Structure pointer Access member using arrow

#include<stdio.h>

struct stu\_detail

{

    int regno;

    char name[20];

    char dept[10];

};

int main()

{

    struct stu\_detail stu\_1 = {1,"Hansi","EEE"};

    struct stu\_detail \*ptr;

    ptr = &stu\_1;

    printf("Student Register Number : %d\n",ptr->regno);

    printf("Student Name : %s\n",ptr->name);

    printf("Student Department : %s",ptr->dept);

    return 0;

}

Output:

Student Register Number : 1

Student Name: Hansi

Student Department: EEE

**Pointer Arithmetic**

Pointer arithmetic is used for manipulating memory location using pointer.

* Increment and decrement
* Addition and subtraction
* Comparison
* Assignment

**Increment Pointer Address:**

* When you increment a pointer , it moves to the next memory location based on the size of the data it points to.
* For example, if you have an integer pointer and you increment it, it will jump to the next integer-sized space in memory (usually 4 bytes).
* The same applies to float pointers; they move by the size of a float (typically 4 bytes) when incremented.

**Example 1:**

#include<stdio.h>

void main()

{

int n = 10;

int \*a;

a = &n;

printf("Before Increment = %d \n",a);

a = a+1;

printf("After Increment = %d \n",a);

}

**Output:**

Before Increment = 6422296

After Increment = 6422300

**Example 2:**

#include<stdio.h>

int main()

{

    int a = 5;

    int \*p,\*r;

    p=&a;

    r=p+1;

    printf("\nSize of Integer : %d",sizeof(a));

    printf("\nP Value : %d",p);

    printf("\nR Value : %d",r);

    return 0;

}

**Output:**

Size of Integer : 4

P Value : 6422292

R Value : 6422296

**Decrement Pointer Address:**

* When a pointer is decremented in C, it moves backward by an amount equal to the size of the data type it is pointing to.

Example 1:

#include<stdio.h>

void main()

{

    int n = 10;

    int \*a;

    a = &n;

    printf("Before Decrement = %d \n",a);

    a = a-1;

    printf("After Decrement = %d \n",a);

    return 0;

}

**Output:**

Before Decrement = 6422296

After Decrement = 6422292

**Example 2:**

#include<stdio.h>

int main()

{

    int a = 5;

    int \*p,\*r;

    p=&a;

    r=p-1;

    printf("\nSize of Integer : %d",sizeof(a));

    printf("\nP Value : %d",p);

    printf("\nR Value : %d",r);

    return 0;

}

**Output:**

Size of Integer : 4

P Value : 6422292

R Value : 6422288

**Pointer Addition**

#include<stdio.h>

int main()

{

  int number=50;

  int \*p;//pointer to int

  p=&number;//stores the address of number variable

  printf("Address of p variable is %u \n",p);

  p=p+3;   //adding 3 to pointer variable

  printf("After adding 3: Address of p variable is %u \n",p);

  return 0;

}

**Output:**

Address of p variable is 6422296

After adding 3: Address of p variable is 6422308

**Pointer Subtraction**

#include<stdio.h>

int main()

{

  int number=50;

  int \*p;//pointer to int

  p=&number;//stores the address of number variable

  printf("Address of p variable is %u \n",p);

  p=p-3; //subtracting 3 from pointer variable

  printf("After subtracting 3: Address of p variable is %u \n",p);

  return 0;

}

**Output:**

Address of p variable is 6422296

After adding 3: Address of p variable is 6422308

**Subtract Between two pointers Address**

#include<stdio.h>

void main ()

{

    int i = 100;

    int \*p = &i;

    int \*temp;

    temp = p;

    p = p + 3;

    printf("Pointer Subtraction: %d - %d = %d",p, temp, p-temp);

}

**Output:**

Pointer Subtraction: 6422304 - 6422292 = 3

**Illegal arithmetic with pointers**

Address + Address = illegal

Address \* Address = illegal

Address % Address = illegal

Address / Address = illegal

Address & Address = illegal

Address ^ Address = illegal

Address | Address = illegal

~Address = illegal

**Comparing the Pointers:**

* Use relational operators (<,>,<=,>=) and equality operators(==,!=) to compare pointers.
* Only possible when both pointers point to same array.
* Output depends upon the relative positions of both the pointers.

**Example:**

#include<stdio.h>

int main ()

{

    int a[] = {1,2,3,4,5,6};

    int \*p = &a[3];

    int \*q = &a[5];

    printf("%d <= %d = %d\n",\*p,\*q,p<=q);

    printf("%d >= %d = %d\n",\*p,\*q,p>=q);

    q=&a[3];

    printf("%d == %d = %d",\*p,\*q,p==q);

    return 0;

}

**Output:**

4 <= 6 = 1

4 >= 6 = 0

4 == 4 = 1

**Finding Output:**

#include<stdio.h>

int main ()

{

    int a[] = {5,16,7,89,45,32,23,10};

    int \*p = &a[1],\*q=&a[5];

    printf("%d ",\*(p+3));

    printf("%d ",\*(q-3));

    printf("%d ",q-p);

    printf("%d ",p<q);

    printf("%d ",\*p<\*q);

    return 0;

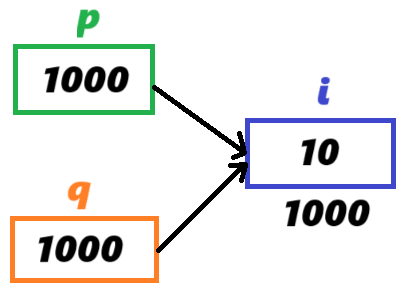
}

**Output:**

* 1. 45 7 4 1 1
  2. 45 4 7 1 1
  3. 44 7 4 1 0
  4. 45 7 4 0 1

**Pointer Assignment**

* + Pointer Assignment using for assign content, one pointer to another pointer.

Example : 1

#include<stdio.h>

int main ()

{

    int i = 10;

    int \*p, \*q;

    p = &i;

    q = p;

    printf("p = %d , q = %d",\*p,\*q);

    return 0;

}

**Output :** p =10 , q = 10

**Example : 2**

#include<stdio.h>

int main ()

{

    int i = 1, \*q, \*p;

    p = &i;

    q = p;

    \*q = 5;

    printf("Address of i = %d\n",&i);

    printf("Store i Address in 'p' = %d\n",p);

    printf("Assign i address to q using p = %d\n",q);

    printf("After Changing P Value using q = %d\n",\*p);

    return 0;

}

**Output:**

Address of i = 6422292

Store i Address in 'p' = 6422292

Assign i address to q using p = 6422292

After Changing P Value using q = 5

**Note :** q = p is not same as \*q = \*p

**Example : 3**

#include<stdio.h>

int main ()

{

    int i = 10, j = 20;

    int \*p, \*q;

    p = &i;

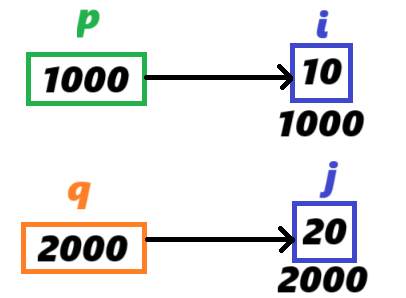
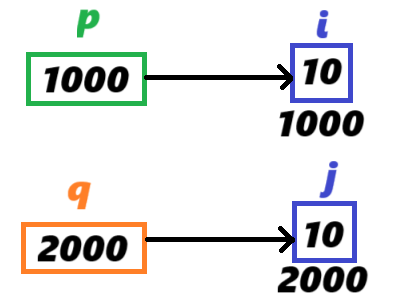
    q = &j;

    printf("Before -> p = %d , q = %d\n",\*p,\*q);

    \*q = \*p;

    printf("After -> p = %d , q = %d\n",\*p,\*q);

    return 0;

}

**Output:**

Before -> p = 10 , q = 20

After -> p = 10 , q = 10

**Dynamic Memory Allocation**

Dynamic memory allocation in c language enables the C programmer to **allocate memory at runtime**. Dynamic memory allocation in c language is possible by 4 functions of stdlib.h header file.

* malloc()
* calloc()
* realloc()
* free()

Difference between static memory allocation and dynamic memory allocation

|  |  |
| --- | --- |
| **Static Memory Allocation** | **Dynamic Memory Allocation** |
| Memory is allocated at compile time. | Memory is allocated at run time. |
| Memory can't be increased while executing program. | Memory can be increased while executing program. |

Problems Faced in Static Memory Allocation

* If you are allocating memory for an array during compile time then you have to **fix the size at the time of declaration.** Size is fixed and user cannot increase of decrease the size of the array at run time.
* If the values stored by the user in the array at run times is **less** than the size specified then there will be wastage of memory.
* If the value stored by the user in the array at run time is **more** than the size specified then the program may crash or misbehave.

**Memory Layout**

Text/Code Segment

Initialized Data

Uninitialized Data

Heap

Stack

* Heap is the segment of memory where dynamic memory allocation take place.
* Unlike stack where memory is allocated or deallocated in a defined order, heap is an area of memory where memory is allocated or deallocated without any order or randomly.
* malloc(), calloc(), realloc(), free() these build in function help in allocating or deallocating some memory space at run time.

**Note:-**

* **Pointers** play an important role in dynamic memory allocation.
* Allocated memory can only be accessed through **pointers**.

What do this method when dynamic memory allocation

|  |  |
| --- | --- |
| **malloc()** | allocates single block of requested memory. |
| **calloc()** | allocates multiple block of requested memory. |
| **realloc()** | reallocates the memory occupied by malloc() or calloc() functions. |
| **free()** | frees the dynamically allocated memory. |

What is **malloc()** ?

malloc is the short name for “memory allocation” and is used to dynamically allocate a single large block of contiguous memory according to the size specified.

Syntax :

**(void\*)malloc(size\_t size)**

**malloc** function simply allocates a memory block according to the size specified in the heap and on success it returns a **pointer** pointing to the first byte of the allocated memory else returns NULL.

**size\_t** is defined in <stdlib.h> as unsigned int.

**Why void Pointer?**

malloc doesn’t have an idea of what it is pointing to.

It merely allocates memory requested by the user without knowing the type of data to be stored inside the memory.

After The void pointer can be typecasted to an appropriate type.

**Ex :**

int \*ptr = (int\*)malloc(4)

here malloc allocates 4 bytes of memory in the heap and the address of the first byte is stored in the pointer ptr.

**Example Program:**

#include<stdio.h>

#include<stdlib.h>

void main()

{

    int i, n;

        printf("Enter the number of integers = ");

        scanf("%d",&n);

        int \*ptr = (int\*)malloc(n\*sizeof(int));

        //Chech Memory Available or Not

        if(ptr == NULL)

        {

          printf("Memory not available.");

          exit(1);

        }

        for(i=0;i<n;i++)

        {

          printf("Enter an integer = ");

          scanf("%d", ptr+i);

        }

        for(i=0;i<n;i++)

        {

          printf("Number %d = %d\n",i,\*(ptr+i));

        }

         return 0;

}

**Output:**

Enter the number of integers = 5

Enter an integer = 4

Enter an integer = 5

Enter an integer = 6

Enter an integer = 3

Enter an integer = 2

Number 0 = 4

Number 1 = 5

Number 2= 6

Number 3= 3

Number 4= 2

What is **calloc()**?

**calloc** is a short name for “contiguous allocation” calloc() function is used to dynamically allocate multiple blocks of memory.

It is different from malloc in two ways:

* calloc() needs two arguments instead of just one

Syntax : **void \*calloc(size\_t n, size\_t size);**

Here **size\_t n** 🡪 Mention number Block

**size\_t size** 🡪 Size of Each Block

Difference between **malloc()** and **calloc()**

* Different 1

**calloc**

**int \*ptr = (int \*)callco(10, sizeof(int));**

An equivalent **malloc** call:

**int \*ptr = (int \*)malloc(10\*sizeof(int));**

* Different 2

**calloc**

Memory allocated by calloc is initialized to zero.

**malloc**

Memory allocated by mallco is initialized with some garbage value.

Note:

malloc and calloc both return NULL when sufficient memory is not available in the heap.

**Example Program:**

#include<stdio.h>

#include<stdlib.h>

void main()

{

    int i, n;

        printf("Enter the number of integers = ");

        scanf("%d",&n);

        int \*ptr = (int\*)calloc(n,sizeof(int));

        //Chech Memory Available or Not

        if(ptr == NULL)

        {

          printf("Memory not available.");

          exit(1);

        }

        for(i=0;i<n;i++)

        {

          printf("Enter an integer = ");

          scanf("%d", ptr+i);

        }

        for(i=0;i<n;i++)

        {

          printf("Number %d = %d\n",i,\*(ptr+i));

        }

         return 0;

}

**Output:**

Enter the number of integers = 4

Enter an integer = 5

Enter an integer = 4

Enter an integer = 3

Enter an integer = 2

Number 0 = 5

Number 1 = 4

Number 2 = 3

Number 3 = 2

What is **realloc()**?

realloc() function is used to change the size of the memory block without losing the old data.

Syntax: **void \*realloc(void \*ptr, size\_t newSize)**

Here , **ptr** 🡪 Pointer to the previously allocated memory.

**newSize** 🡪 Give New Size, how many time you want extend memory.

On failure, realloc returns NULL.

Example:

int \***ptr** = (int \*)malloc(sizeof(int)); //previously ptr have 4 bytes

ptr = (int \*)**realloc**(**ptr**, **2\*sizeof(int)**); // Now ptr have 8 bytes

* This will allocate memory space of 2\*sizeof(int)
* Also, this function moves the contents of the old block to new block and the data of the old is not lost.
* We may lose the data when the new size is smaller than the old size.
* Newly allocated bytes are uninitialized.

**Example Program:**

#include<stdio.h>

#include<stdlib.h>

void main()

{

  int i;

  int \*ptr = (int \*)malloc(2\*sizeof(int));

  if(ptr == NULL)

  {

    printf("Memory not available");

    exit(1);

  }

  printf("Enter the two numbers : \n");

  for(i = 0; i<2; i++)

  {

    scanf("%d", ptr+i);

  }

  ptr = (int \*)realloc(ptr, 4\*sizeof(int));

  if(ptr == NULL)

  {

    printf("Memory not available!");

    exit(1);

  }

  printf("Enter 2 more numbers : \n");

  for(i=2; i<4; i++)

  {

    scanf("%d", ptr+i);

  }

  for(i=0; i<4; i++)

  {

    printf("%d ",\*(ptr+i));

  }

  return 0;

}

**Output:**

Enter the two numbers :

5

4

Enter 2 more numbers :

3

2

5 4 3 2

What is **free()**?

**free()** function is used to release the dynamically allocated memory in heap.

Syntax: **void free(ptr);**

The memory allocated in heap will not be released automatically after using the memory. The space remains there and can’t be used.

It is the programmer’s responsibility to release the memory after use.

Example:

int main()

{

int \*ptr = (int \*)malloc(4\*sizeof(int));

. . . . . . .

free(ptr);

}

**Example Program:**

// Program to calculate the sum of n numbers entered by the user

#include <stdio.h>

#include <stdlib.h>

int main() {

  int n, i, \*ptr, sum = 0;

  printf("Enter number of elements: ");

  scanf("%d", &n);

  ptr = (int\*) malloc(n \* sizeof(int));

  // if memory cannot be allocated

  if(ptr == NULL) {

    printf("Error! memory not allocated.");

    exit(0);

  }

  printf("Enter elements:\n");

  for(i = 0; i < n; ++i) {

    scanf("%d", ptr + i);

    sum += \*(ptr + i);

  }

  printf("Sum = %d", sum);

  // deallocating the memory

  free(ptr); **// releasing the memory at the end**

ptr = NULL; // Assign NULL to the pointer after releasing the memory.

  return 0;

}

**Output:**

Enter number of elements: 5

Enter elements:

6

3

2

1

1

Sum = 13