C Language with explanations in Hindi

Part 2

C Functions

Functions

Ek function code ka ek block hota hai jo tab run hota hai jab usse call kiya jata hai.

Aap function mein data bhi pass kar sakte hain, jo "parameters" kehlaata hai.

Functions ka use kuch specific actions ko perform karne ke liye hota hai, aur yeh code reuse karne mein madadgar hote hain: ek baar code likho, aur jab chaaho use karo.

Predefined Functions

Toh aap already jaante hain ki function kya hota hai. Aap puri tutorial ke dauraan functions ka use kar rahe the!

For example, `main()` ek function hai, jo code ko execute karne ke liye use hota hai, aur `printf()` bhi ek function hai; jo screen par text print karta hai:

Example:

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

Create a Function

Apna function banane (ya declare karne) ke liye, aapko function ka naam specify karna padta hai, uske baad parentheses () aur curly brackets {}:

Syntax

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

Example Explained

- `myFunction()` function ka naam hai
- `void` ka matlab hai ki function koi value return nahi karega. Aap aage jaake return values ke baare mein seekhenge
- Function ke andar (body mein), aap woh code likhte hain jo aap chahte hain ki function kare

Call a Function

Jo functions aap declare karte hain, woh turant execute nahi hote. Unhe "later use ke liye save" kiya jata hai, aur woh tab execute hote hain jab unhe call kiya jata hai.

Function ko call karne ke liye, bas function ka naam likho, uske baad parentheses () aur semicolon ;

Example

Main function mein `myFunction()` ko call karte hain:

```
// Create a function
void myFunction() {
  printf("I just got executed!");
}

int main() {
  myFunction(); // call the function
  return 0;
}

// Outputs "I just got executed!"
```

Output:

```
I just got executed!
```

Aap function ko multiple baar bhi call kar sakte hain:

Example

```
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!
```

Calculate the Sum of Numbers

Aap function ke andar jo chaaho, woh likh sakte ho. Function ka main purpose yeh hota hai ki aap code ko save kar sakein, aur jab zaroorat ho tab usse execute kar sakein.

Jaise niche ke example mein, humne ek function banaya hai jo do numbers ka sum calculate karta hai. Jab aapko function execute karna ho (aur calculation perform karni ho), bas usse call karna padta hai:

Example

```
void calculateSum() {
  int x = 5;
  int y = 10;
  int sum = x + y;
  printf("The sum of x + y is: %d", sum);
}

int main() {
  calculateSum();  // call the function
  return 0;
}
```

Output:

```
The sum of x + y is: 15
```

Yeh ek simple example tha jo ek function ka use dikhata hai, jo alag-alag statements ko execute karta hai C mein. Lekin function ki asli power tab dikhai deti hai jab hum "parameters" pass karte hain, jaise agle chapter mein hum seekhenge. Isse function ko hum kisi bhi numbers ka sum calculate karne ke liye use kar sakte hain, na ki sirf fixed values 5 aur 10 ke liye.

C Function Parameters

Parameters and Arguments

Functions mein information pass ki ja sakti hai, jo function ke andar variables ki tarah kaam karti hai.

Parameters ko function ke naam ke baad, parentheses ke andar specify kiya jata hai. Aap jitne chahe parameters add kar sakte hain, bas unhe comma se separate karna hota hai.

Syntax

```
returnType functionName(parameter1, parameter2, parameter3) {
  // code to be executed
}
```

Example Explained

Niche ke example mein, function ek string (name) parameter ke roop mein leta hai. Jab function ko call kiya jata hai, hum ek naam pass karte hain, jo function ke andar use hota hai "Hello" aur har person ka naam print karne ke liye:

```
void myFunction(char name[]) {
  printf("Hello %s\n", name);
}

int main() {
  myFunction("Liam");
  myFunction("Jenny");
  myFunction("Anja");
  return 0;
}
```

Output:

```
Hello Liam
Hello Jenny
Hello Anja
```

Jab ek parameter function ko pass kiya jata hai, use argument kaha jata hai. Toh example ke hisaab se: `name` ek parameter hai, jabki `Liam`, `Jenny`, aur `Anja` arguments hain.

Multiple Parameters

Function ke andar, aap jitne chahe parameters add kar sakte hain:

Example

```
void myFunction(char name[], int age) {
  printf("Hello %s. You are %d years old.\n", name, age);
}

int main() {
  myFunction("Liam", 3);
  myFunction("Jenny", 14);
  myFunction("Anja", 30);
  return 0;
}
```

Output:

```
Hello Liam. You are 3 years old.
Hello Jenny. You are 14 years old.
Hello Anja. You are 30 years old.
```

Agar hum pehle wale "Calculate the Sum of Numbers" example ko dekhein, toh hum function parameters ka use karke ek sustainable program bana sakte hain:

Example

```
void calculateSum(int x, int y) {
  int sum = x + y;
  printf("The sum of %d + %d is: %d\n", x, y, sum);
}

int main() {
  calculateSum(5, 3);
```

```
  calculateSum(8, 2);
  calculateSum(15, 15);
  return 0;
}
```

Notes on Parameters

Jab aap multiple parameters ke saath kaam karte hain, toh yeh zaroori hai ki function call mein jitne parameters hain, utne arguments pass karein, aur woh arguments usi order mein hon.

Pass Arrays as Function Parameters

Aap arrays ko bhi function ke parameters ke roop mein pass kar sakte hain:

Example

```
void myFunction(int myNumbers[5]) {
  for (int i = 0; i < 5; i++) {
    printf("%d\n", myNumbers[i]);
  }
}

int main() {
  int myNumbers[5] = {10, 20, 30, 40, 50};
  myFunction(myNumbers);
```

```
  return 0;
}
```

Example Explained

Yeh function (`myFunction`) ek array ko parameter ke roop mein leta hai (`int myNumbers[5]`), aur array ke elements ko for loop ke saath print karta hai.

Jab function ko `main()` mein call kiya jata hai, hum array `myNumbers` pass karte hain, jisse array ke elements print ho jate hain.

Note: Jab aap function ko call karte hain, toh array ka sirf naam use karke argument pass karte hain (`myFunction(myNumbers)`), lekin function mein array ka pura declaration zaroori hai (`int myNumbers[5]`).

Return Values

Pichle examples mein jo `void` keyword use kiya gaya tha, uska matlab tha ki function koi value return nahi karega. Agar aap chaahte hain ki function koi value return kare, toh aapko `void` ki jagah koi data type (jaise `int`, `float`, etc.) use karna padega, aur function ke andar `return` keyword ka istemal karna padega:

Example

```
int myFunction(int x) {
  return 5 + x;
}
```

```
int main() {
  printf("Result is: %d", myFunction(3));
  return 0;
}
```

Output:

```
Result is: 8 (5 + 3)
```

Yeh example ek function ko return karne ke liye tha jo do parameters ka sum return karta hai:

Example

```
int myFunction(int x, int y) {
  return x + y;
}

int main() {
  printf("Result is: %d", myFunction(5, 3));
  return 0;
}
```

Output:

```
Result is: 8 (5 + 3)
```

Aap result ko variable mein bhi store kar sakte hain:

Example

```
int myFunction(int x, int y) {
  return x + y;
```

```
}

int main() {
  int result = myFunction(5, 3);
  printf("Result is = %d", result);
  return 0;
}
```

Output:

```
Result is = 8 (5 + 3)
```

Agar hum "Calculate the Sum of Numbers" example ko dobara dekhein, toh hum `return` ka use karke results ko alag-alag variables mein store kar sakte hain, jo program ko aur flexible aur controlable banata hai:

Example

```
int calculateSum(int x, int y) {
  return x + y;
}

int main() {
  int result1 = calculateSum(5, 3);
  int result2 = calculateSum(8, 2);
  int result3 = calculateSum(15, 15);

  printf("Result1 is: %d\n", result1);
  printf("Result2 is: %d\n", result2);
  printf("Result3 is: %d\n", result3);

  return 0;
}
```

Output:

```
Result1 is: 8
Result2 is: 10
Result3 is: 30
```

Tip:

Agar aapke paas kai sare result variables hain, toh unhe array mein store karna behtar hota hai:

Example

```
int calculateSum(int x, int y) {
  return x + y;
}

int main() {
  // Create an array
  int resultArr[6];

  // Call the function with different arguments and store
the results in the array
  resultArr[0] = calculateSum(5, 3);
  resultArr[1] = calculateSum(8, 2);
  resultArr[2] = calculateSum(15, 15);
  resultArr[3] = calculateSum(9, 1);
  resultArr[4] = calculateSum(7, 7);
  resultArr[5] = calculateSum(1, 1);

  for (int i = 0; i < 6; i++) {
    printf("Result%d is = %d\n", i + 1, resultArr[i]);
  }
```

```
  return 0;
}
```

Real-Life Example

Chaliye ek practical example dekhte hain jisme hum ek function banayenge jo fahrenheit se celsius mein conversion kare:

Example

```
// Function to convert Fahrenheit to Celsius
float toCelsius(float fahrenheit) {
  return (5.0 / 9.0) * (fahrenheit - 32.0);
}

int main() {
  // Set a fahrenheit value
  float f_value = 98.8;

  // Call the function with the fahrenheit value
  float result = toCelsius(f_value);

  // Print the fahrenheit value
  printf("Fahrenheit: %.2f\n", f_value);

  // Print the result
  printf("Convert Fahrenheit to Celsius: %.2f\n", result);

  return 0;
}
```

Output:

```
Fahrenheit: 98.80
Convert Fahrenheit to Celsius: 37.11
```

C Variable Scope

Scope

Ab jab aapko functions ka kaam samajh aa gaya hai, toh ye zaroori hai ki aap samjhein ki variables function ke andar aur bahar kaise behave karte hain.

C mein, variables sirf unhi regions ke andar accessible hote hain jahan unhe create kiya gaya hai. Isse scope kehte hain.

Local Scope

Agar ek variable function ke andar banaya gaya ho, toh wo us function ke local scope ka hissa hota hai, aur sirf us function ke andar use kiya ja sakta hai:

Example

```
void myFunction() {
  // Local variable that belongs to myFunction
  int x = 5;
```

```
  // Print the variable x
  printf("%d", x);
}

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

Ek local variable ko function ke bahar access nahi kiya ja sakta.

Agar aap isse function ke bahar access karne ki koshish karte hain, toh error hota hai:

Example

```
void myFunction() {
  // Local variable that belongs to myFunction
  int x = 5;
}

int main() {
  myFunction();

  // Print the variable x in the main function
  printf("%d", x);
  return 0;
}
```

Global Scope

Agar ek variable function ke bahar banaya jaye, toh wo global variable hota hai aur global scope ka part hota hai.

Global variables har scope mein available hote hain, chahe wo global ho ya local:

Example

```
// Global variable x
int x = 5;

void myFunction() {
  // We can use x here
  printf("%d", x);
}

int main() {
  myFunction();

  // We can also use x here
  printf("%d", x);
  return 0;
}
```

Naming Variables

Agar aap function ke andar aur bahar same variable ka naam use karte hain, toh C unhe alag-alag variables samajhta hai. Ek global scope mein hoga (bahar) aur ek local scope mein hoga (andar):

Example

Function local x ko print karega, aur phir global x ko print karega:

```
// Global variable x
int x = 5;

void myFunction() {
  // Local variable with the same name as the global variable (x)
  int x = 22;
  printf("%d\n", x); // Refers to the local variable x
}

int main() {
  myFunction();

  printf("%d\n", x); // Refers to the global variable x
  return 0;
}
```

Lekin aapko global aur local variable ka naam same nahi rakhna chahiye, kyunki ye confusion aur errors ka reason ban sakta hai.

Generally, aapko global variables se bachna chahiye, kyunki inhe kisi bhi function se access aur modify kiya ja sakta hai:

Example

`myFunction` se x ki value ko change karna:

```
// Global variable
int x = 5;

void myFunction() {
  printf("%d\n", ++x); // Increment the value of x by 1 and
print it
}

int main() {
```

```
  myFunction();

  printf("%d\n", x); // Print the global variable x
  return 0;
}

// The value of x is now 6 (no longer 5)
```

Conclusion

Summing up, jitna ho sake local variables ka use karo (achhe variable names ke saath). Isse aapka code maintain karna asaan hota hai aur samajhne mein bhi. Lekin aapko global variables existing C programs mein ya jab aap doosron ke saath collaborate karte ho tab mil sakte hain. Isliye scope ka kaise kaam karta hai aur kaise aap ise effectively use kar sakte hain, ye samajhna zaroori hai, taaki aapka code clear aur functional rahe.

C Function Declaration and Definition

Function Declaration and Definition

Aapne pehle ke chapters mein seekh liya hai ki aap C mein ek function bana sakte hain aur usse call kar sakte hain, is tarah:

Example

```c
// Create a function
void myFunction() {
  printf("I just got executed!");
}

int main() {
  myFunction(); // call the function
  return 0;
}
```

Ek function do parts se milkar banta hai:

1. **Declaration:** Function ka naam, return type, aur parameters (agar koi ho)
2. **Definition:** Function ka body — matlab wo code jo execute hoga

Example:

```c
void myFunction() { // declaration
  // the body of the function (definition)
}
```

Code ko optimize karne ke liye ye recommend kiya jata hai ki function ka declaration aur definition alag rakha jaye.

Aksar aap C programs mein dekhoge ki function declaration `main()` ke upar hota hai, aur function definition `main()` ke neeche likha hota hai.

Isse code zyada organized aur readable lagta hai.

Example

```c
// Function declaration
void myFunction();

// The main method
int main() {
  myFunction();  // call the function
  return 0;
}

// Function definition
void myFunction() {
  printf("I just got executed!");
}
```

What About Parameters (Parameters ke baare mein)

Agar hum function parameters aur return values wale example ko lein:

Example

```c
int myFunction(int x, int y) {
  return x + y;
}

int main() {
  int result = myFunction(5, 3);
  printf("Result is = %d", result);
  return 0;
}
```

Output:

```
8   (5 + 3)
```

Toh best practice ye mani jati hai ki aap ise is tarah likhein:

Example

```
// Function declaration
int myFunction(int x, int y);

// The main method
int main() {
  int result = myFunction(5, 3); // call the function
  printf("Result is = %d", result);
  return 0;
}

// Function definition
int myFunction(int x, int y) {
  return x + y;
}
```

Functions Calling Other Functions

Agar aapne pehle functions ko declare kar liya hai, toh ek function doosre function ko call bhi kar sakta hai.

Example

Ek function ko doosre function se call karna:

```
// Declare two functions, myFunction and myOtherFunction
void myFunction();
void myOtherFunction();
```

```c
int main() {
  myFunction(); // call myFunction (from main)
  return 0;
}

// Define myFunction
void myFunction() {
  printf("Some text in myFunction\n");
  myOtherFunction(); // call myOtherFunction (from myFunction)
}

// Define myOtherFunction
void myOtherFunction() {
  printf("Hey! Some text in myOtherFunction\n");
}
```

C Recursion

Recursion

Recursion ek technique hai jisme ek function khud ko hi call karta hai. Ye technique complicated problems ko chhote aur simple problems mein tod deti hai, jisse unhe solve karna easy ho jata hai.

Recursion samajhna thoda mushkil ho sakta hai, lekin isse samajhne ka best tareeka hai ki aap iske saath experiment karein.

Recursion Example

Do numbers ko add karna easy hai, lekin ek range ke numbers ko add karna thoda complex hota hai. Neeche diye gaye example mein, recursion ka use karke 1 se 10 tak ke numbers ko add kiya gaya hai — jisme problem ko chhote tasks mein divide kiya gaya hai (do numbers add karna):

Example

Recursion ka use karke 1 se 10 tak saare numbers add karo:

```
#include <stdio.h>

int sum(int k);

int main() {
  int result = sum(10);
  printf("%d", result);
  return 0;
}

int sum(int k) {
  if (k > 0) {
    return k + sum(k - 1);
  } else {
    return 0;
  }
}
```

Example Explained

Jab `sum()` function call hota hai, toh wo parameter `k` ko usse chhote numbers ke sum ke saath add karta hai, aur result return karta hai.
Jab `k` 0 ho jata hai, function sirf `0` return karta hai.

Program kuch is tarah se steps follow karta hai:

```
10 + sum(9)
10 + (9 + sum(8))
10 + (9 + (8 + sum(7)))
...
10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 + sum(0)
10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 + 0
```

Kyuki function `k = 0` hone par khud ko call nahi karta, program wahi ruk jata hai aur final result return karta hai.

Developer ko recursion likhte waqt **bahut careful** rehna chahiye, kyunki kabhi-kabhi galti se aisa function likh diya jata hai jo kabhi terminate nahi hota, ya fir bahut zyada memory aur processor power use karta hai.
Lekin agar sahi tarah likha jaye, toh recursion ek **efficient** aur **mathematically elegant** programming approach hai.

Example

Recursion ka use karke 5 se countdown karo:

```c
#include <stdio.h>

void countdown(int n);

int main() {
  countdown(5);
  return 0;
}

void countdown(int n) {
  if (n > 0) {
```

```
    printf("%d ", n);
    countdown(n - 1);
  }
}
```

Yahan function khud ko `n - 1` ke saath call karta rahta hai jab tak `n` 0 nahi ho jata.

Calculate Factorial with Recursion

Is example mein recursion ka use karke 5 ka factorial calculate kiya gaya hai:

Example

```
#include <stdio.h>

int factorial(int n);

int main() {
  printf("Factorial of 5 is %d", factorial(5));
  return 0;
}

int factorial(int n) {
  if (n > 1) {
    return n * factorial(n - 1);
  } else {
    return 1;
  }
}
```

Explanation:
Factorial ka matlab hota hai kisi number ko uske niche ke saare numbers se multiply karna, jab tak 1 tak na pahunch jaye.
For example:

```
5! = 5 * 4 * 3 * 2 * 1 = 120
```

Aur definition ke hisaab se,

```
0! = 1
```

Is tarah recursion ka use karke factorial ya aur bhi mathematical calculations easily ki ja sakti hain.

C Math Functions

Math Functions

C mein kuch mathematical functions available hote hain, jo aapko numbers par mathematical tasks perform karne ki suvidha dete hain.

Inhe use karne ke liye aapko apne program mein **math.h** header file include karni hoti hai:

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

Square Root

Agar aapko kisi number ka square root find karna ho, toh aap **sqrt()** function ka use kar sakte hain:

Example

```
printf("%f", sqrt(16));
```

Round a Number

- **ceil()** function ek number ko uske nearest integer ke upar round karta hai.
- **floor()** function ek number ko uske nearest integer ke neeche round karta hai.

Example

```
printf("%f", ceil(1.4));
printf("%f", floor(1.4));
```

Power

pow() function x ko y ki power par raise karta hai (x^y).

Example

```
printf("%f", pow(4, 3));
```

C Files

File Handling

C mein aap files create, open, read, aur write kar sakte hain, ek pointer variable ka use karke jo **FILE** type ka hota hai. Aur aap **fopen()** function ka use karte hain:

```
FILE *fptr;
fptr = fopen(filename, mode);
```

FILE ek data type hai, aur humein iske saath kaam karne ke liye ek pointer variable (jaise fptr) ki zarurat hoti hai. Abhi ke liye, yeh sirf ek cheez hai jo aapko files ke saath kaam karte waqt samajhni hoti hai.

fopen() function ko actually file open karne ke liye use kiya jata hai. Isme do parameters hote hain:

Parameter	Description
filename	Woh actual file ka naam jise aap open (ya create) karna chahte hain, jaise filename.txt
mode	Ek single character, jo batata hai ki aap file ke saath kya karna chahte hain (read, write, append):
w	File mein likhne ke liye
a	File mein naye data ko append karne ke liye
r	File ko padhne ke liye

Create a File

Agar aapko ek file create karni ho, toh aap **w** mode ka use karte hain **fopen()** function mein.

w mode file mein likhne ke liye use hota hai. Agar file pehle se exist nahi karti, toh yeh function ek nayi file bana dega:

Example

```
FILE *fptr;
```

```
// Create a file
fptr = fopen("filename.txt", "w");

// Close the file
fclose(fptr);
```

Note:
File aapke C program ki directory mein create hogi, agar aapne kuch aur specify nahi kiya ho.

Aapke computer par yeh kuch is tarah dikhega:

Tip: Specific Folder Mein File Create Karna

Agar aap file ko kisi specific folder mein create karna chahte hain, toh aap **absolute path** de sakte hain (yaad rakhein, backslashes ko double backslashes mein dena hota hai, jaisa humne strings mein special characters ke liye kiya tha):

```
fptr = fopen("C:\\directoryname\\filename.txt", "w");
```

Closing the File

Kya aapne humare example mein **fclose()** function dekha?

Jab hum file ka kaam khatam kar lete hain, tab yeh function file ko band kar deta hai.

Yeh ek **achhi practice** mani jati hai, kyunki isse yeh ensure hota hai ki:

1. **Changes properly save ho jayein**
2. **Dusre programs file ka use kar sakein** (agar aap chahein)
3. **Memory space free ho jaye**

C Write To Files

Write To a File

Chaliye, pichle chapter ke **w** mode ko phir se use karte hain aur us file mein kuch likhte hain jo humne abhi banayi thi.

w mode ka matlab hota hai ki file **writing** ke liye open ki gayi hai.
File mein content insert karne ke liye aap **fprintf()** function ka use kar sakte hain — isme aapko file pointer (jaise `fptr`) aur likhne wala text dena hota hai.

Example:

```
FILE *fptr;

// Open a file in writing mode
fptr = fopen("filename.txt", "w");

// Write some text to the file
fprintf(fptr, "Some text");

// Close the file
fclose(fptr);
```

Result:
Jab aap apne computer par file open karenge, toh usme yeh likha milega:

```
Some text
```

Note:
Agar aap kisi aisi file mein likhte hain jo **pehle se exist karti hai**, toh **purana content delete ho jata hai** aur naya content uski jagah likh diya jata hai.
Yeh baat dhyaan mein rakhni chahiye, warna aap galti se apna purana data overwrite kar sakte hain.

Example:

```
fprintf(fptr, "Hello World!");
```

Result:
Ab jab aap file open karenge, toh usme likha hoga:

```
Hello World!
```

(purane "Some text" ki jagah)

Append Content To a File

Agar aap file mein **naya content add** karna chahte hain bina purane content ko delete kiye, toh aap **a** mode ka use kar sakte hain.

a mode file ke **end mein content add (append)** karta hai.

Example:

```
FILE *fptr;

// Open a file in append mode
fptr = fopen("filename.txt", "a");

// Append some text to the file
fprintf(fptr, "\nHi everybody!");
```

```
// Close the file
fclose(fptr);
```

Result:
Jab aap apne computer par file open karenge, toh aapko kuch aisa dikhega:

```
Hello World!
Hi everybody!
```

Note:
Bilkul **w** mode ki tarah, agar file exist nahi karti hai, toh **a** mode automatically ek nayi file create kar dega jisme "appended" content likha hoga.

C Read Files

Read a File

Pichle chapter mein humne **w** aur **a** modes ka use karke file mein likha tha. Ab hum seekhenge ki file ko kaise **read** kiya jata hai.

File ko **read** mode mein kholne ke liye aap **r** mode ka use karte hain:

Example:

```
FILE *fptr;

// Open a file in read mode
fptr = fopen("filename.txt", "r");
```

Isse **filename.txt** file reading ke liye open ho jayegi.

File ko read karna C mein thoda kaam hai, lekin chinta mat kariye! Hum aapko step-by-step guide karenge.

Step 1: String Banayein

Sabse pehle, humein ek string create karni hogi jo file ke content ko store kar sake.
Maan lijiye, hum ek string banate hain jo **100 characters** tak ka content store kar sake:

Example:

```
FILE *fptr;

// Open a file in read mode
fptr = fopen("filename.txt", "r");

// Store the content of the file
char myString[100];
```

Step 2: Content Read Karna

File ka content read karne ke liye **fgets()** function ka use karte hain.

fgets() function ko 3 parameters chahiye:

1. **First parameter:** Jahan file ka content store hoga, jo humne `myString` array mein rakha hai.
2. **Second parameter:** Maximum size of data to read, jo humare case mein **100** hoga.

3. **Third parameter:** File pointer, jo file ko read karne mein madad karega (humare case mein **fptr**).

Example:

```
fgets(myString, 100, fptr);
```

Step 3: Content Print Karna

Ab hum string ko print kar sakte hain, jisme file ka content hoga:

Example:

```
FILE *fptr;

// Open a file in read mode
fptr = fopen("filename.txt", "r");

// Store the content of the file
char myString[100];

// Read the content and store it inside myString
fgets(myString, 100, fptr);

// Print the file content
printf("%s", myString);

// Close the file
fclose(fptr);
```

Result:
Jab file open karenge, toh output yeh hoga:

```
Hello World!
```

Note:
fgets() function sirf file ki pehli line ko read karta hai. Agar aapko yaad ho, toh **filename.txt** mein do lines thi.

Step 4: Multiple Lines Read Karna

Agar file mein multiple lines hain, toh **while loop** ka use karke har line ko read kar sakte hain:

Example:

```
FILE *fptr;

// Open a file in read mode
fptr = fopen("filename.txt", "r");

// Store the content of the file
char myString[100];

// Read the content and print it
while(fgets(myString, 100, fptr)) {
  printf("%s", myString);
}

// Close the file
fclose(fptr);
```

Result:
Jab aap file open karenge, toh output yeh hoga:

```
Hello World!
Hi everybody!
```

Good Practice: File Open Check

Agar aap kisi file ko **read** mode mein open karte hain jo exist nahi karti, toh **fopen()** function **NULL** return karega.

Tip: Ek achi practice yeh hai ki **NULL** check karne ke liye **if statement** ka use kiya jaye, aur agar file nahi milti, toh ek message print kiya jaye:

Example:

```
FILE *fptr;

// Open a file in read mode
fptr = fopen("loremipsum.txt", "r");

// Print some text if the file does not exist
if(fptr == NULL) {
  printf("Not able to open the file.");
}

// Close the file
fclose(fptr);
```

Result:

Agar file exist nahi karti hai, toh yeh message print hoga:

```
Not able to open the file.
```

Step 5: Sustainable Code

Ab hum apne "read a file" example ko thoda aur sustainable banate hain. Agar file exist karti hai, toh hum content read aur print karenge, aur agar file exist nahi karti, toh ek message print karenge:

Example:

```c
FILE *fptr;

// Open a file in read mode
fptr = fopen("filename.txt", "r");

// Store the content of the file
char myString[100];

// If the file exist
if(fptr != NULL) {

  // Read the content and print it
  while(fgets(myString, 100, fptr)) {
    printf("%s", myString);
  }

// If the file does not exist
} else {
  printf("Not able to open the file.");
}

// Close the file
fclose(fptr);
```

Result:

Agar file exist karti hai, toh output yeh hoga:

```
Hello World!
Hi everybody!
```

Agar file exist nahi karti hai, toh message yeh print hoga:

C Structures (structs)

Structures

Structures (yaani **structs**) ek tareeqa hai jisme aap kai related variables ko ek sath group kar sakte hain.

Har variable ko structure ka member kaha jata hai.

Array ke mukable, structure mein alag-alag data types (jaise **int**, **float**, **char**, etc.) ho sakte hain.

Create a Structure

Aap **struct** keyword ka use karke ek structure bana sakte hain aur uske members ko curly braces ke andar declare karte hain:

Example:

```
struct MyStructure {   // Structure declaration
  int myNum;           // Member (int variable)
  char myLetter;       // Member (char variable)
}; // End the structure with a semicolon
```

Structure ko access karne ke liye, aapko iske liye ek variable banana padta hai.

Example:

```
struct myStructure {
  int myNum;
  char myLetter;
```

```
};

int main() {
  struct myStructure s1;
  return 0;
}
```

Access Structure Members

Structure ke members ko access karne ke liye **dot (.) syntax** ka use hota hai.

Example:

```
// Create a structure called myStructure
struct myStructure {
  int myNum;
  char myLetter;
};

int main() {
  // Create a structure variable of myStructure called s1
  struct myStructure s1;

  // Assign values to members of s1
  s1.myNum = 13;
  s1.myLetter = 'B';

  // Print values
  printf("My number: %d\n", s1.myNum);
  printf("My letter: %c\n", s1.myLetter);

  return 0;
}
```

Output:

```
My number: 13
My letter: B
```

Create Multiple Structure Variables

Ab aap aasani se multiple structure variables bana sakte hain alag-alag values ke saath:

Example:

```
// Create different struct variables
struct myStructure s1;
struct myStructure s2;

// Assign values to different struct variables
s1.myNum = 13;
s1.myLetter = 'B';

s2.myNum = 20;
s2.myLetter = 'C';
```

What About Strings in Structures?

Yaad rakhein, C mein strings actually characters ka ek array hota hai. Aap directly string ko array ko assign nahi kar sakte. Maan lijiye:

Example:

```
struct myStructure {
  int myNum;
  char myLetter;
  char myString[30];  // String
};

int main() {
```

```
  struct myStructure s1;

  // Trying to assign a value to the string
  s1.myString = "Some text";

  // Trying to print the value
  printf("My string: %s", s1.myString);

  return 0;
}
```

An error will occur:

Error:

```
prog.c:12:15: error: assignment to expression with array type
```

Solution:

Iske liye aapko **strcpy()** function ka use karna padta hai, jaise:

Example:

```
struct myStructure {
  int myNum;
  char myLetter;
  char myString[30]; // String
};

int main() {
  struct myStructure s1;

  // Assign a value to the string using the strcpy function
  strcpy(s1.myString, "Some text");
```

```
  // Print the value
  printf("My string: %s", s1.myString);

  return 0;
}
```

Output:

```
My string: Some text
```

Simpler Syntax

Aap ek structure variable ko declare karte waqt uske members ko ek line mein bhi assign kar sakte hain:

Example:

```
// Create a structure
struct myStructure {
  int myNum;
  char myLetter;
  char myString[30];
};

int main() {
  // Create a structure variable and assign values to it
  struct myStructure s1 = {13, 'B', "Some text"};

  // Print values
  printf("%d %c %s", s1.myNum, s1.myLetter, s1.myString);

  return 0;
}
```

Output:

```
13 B Some text
```

Note:

Jo values aap insert karte hain unka order structure mein declared variable types ke saath match hona chahiye (jaise **13** for **int**, **'B'** for **char**, etc.).

Copy Structures

Aap ek structure ko doosre structure ko assign kar sakte hain. Jaise s1 ke values ko s2 mein copy karna:

Example:

```
struct myStructure s1 = {13, 'B', "Some text"};
struct myStructure s2;

s2 = s1;
```

Modify Values

Agar aap kisi structure ke values ko change karna chahte hain, toh aap **dot syntax** ka use karte hain. Aur string ko modify karte waqt **strcpy()** function ka use hota hai:

Example:

```
struct myStructure {
  int myNum;
```

```
  char myLetter;
  char myString[30];
};

int main() {
  // Create a structure variable and assign values to it
  struct myStructure s1 = {13, 'B', "Some text"};

  // Modify values
  s1.myNum = 30;
  s1.myLetter = 'C';
  strcpy(s1.myString, "Something else");

  // Print values
  printf("%d %c %s", s1.myNum, s1.myLetter, s1.myString);

  return 0;
}
```

Output:

```
30 C Something else
```

Modifying values especially useful hota hai jab aap ek structure ko doosre mein copy karte hain.

Example:

```
// Create a structure variable and assign values to it
struct myStructure s1 = {13, 'B', "Some text"};

// Create another structure variable
struct myStructure s2;

// Copy s1 values to s2
s2 = s1;
```

```
// Change s2 values
s2.myNum = 30;
s2.myLetter = 'C';
strcpy(s2.myString, "Something else");

// Print values
printf("%d %c %s\n", s1.myNum, s1.myLetter, s1.myString);
printf("%d %c %s\n", s2.myNum, s2.myLetter, s2.myString);
```

Output:

```
13 B Some text
30 C Something else
```

How are Structures Useful?

Maan lijiye aapko ek program likhna hai jo different cars ke baare mein information store kare, jaise brand, model, aur year. Structures ki madad se, aap ek single "Car template" bana sakte hain aur har car ka data usi template ke according store kar sakte hain.

Real-Life Example:

Cars ke baare mein information store karne ke liye structure ka use karte hain:

Example:

```
struct Car {
  char brand[30];
  char model[30];
  int year;
```

```
};

int main() {
  struct Car car1 = {"BMW", "X5", 1999};
  struct Car car2 = {"Ford", "Mustang", 1969};
  struct Car car3 = {"Toyota", "Corolla", 2011};

  printf("%s %s %d\n", car1.brand, car1.model, car1.year);
  printf("%s %s %d\n", car2.brand, car2.model, car2.year);
  printf("%s %s %d\n", car3.brand, car3.model, car3.year);

  return 0;
}
```

Output:

```
BMW X5 1999
Ford Mustang 1969
Toyota Corolla 2011
```

C Nested Structures

Nested Structs

Ek structure mein doosra structure member ke roop mein ho sakta hai. Isse nested

structure kaha jata hai, aur yeh tab useful hota hai jab aapko related data ko layers mein group karna ho:

Example

```
struct Owner {
  char firstName[30];
  char lastName[30];
};

struct Car {
  char brand[30];
  int year;
  struct Owner owner; // Nested structure
};

int main() {
  struct Owner person = {"John", "Doe"};
  struct Car car1 = {"Toyota", 2010, person};

  printf("Car: %s (%d)\n", car1.brand, car1.year);
  printf("Owner: %s %s\n", car1.owner.firstName,
car1.owner.lastName);

  return 0;
}
```

Yahan, Car structure ke andar ek aur structure (Owner) hai. Isse complex data ko organize karna aasaan ho jata hai, jaise ki ek car aur uske owner ka data.

C Structs and Pointers

Structures and Pointers

Aap structs ke saath pointers ka use kar sakte ho taaki aapka code zyada efficient bane — khaas kar jab aap structs ko functions mein pass karte ho ya unke values change karte ho.

Struct ke pointer ko use karne ke liye, bas * symbol lagana hota hai, jaise aap doosre data types ke saath karte ho.

Uske members ko access karne ke liye aapko -> operator ka use karna hota hai, . (dot) ke badle.

Example

```c
// Define a struct
struct Car {
  char brand[30];
  int year;
};

int main() {
  struct Car car = {"Toyota", 2020};

  // Declare a pointer to the struct
  struct Car *ptr = &car;

  // Access members using the -> operator
  printf("Brand: %s\n", ptr->brand);
  printf("Year: %d\n", ptr->year);

  return 0;
}
```

Passing Struct Pointers to Functions

Yahan dekhte hain kaise aap ek struct pointer ko function mein pass karke uske values change kar sakte ho:

Example

```c
struct Car {
  char brand[30];
  int year;
};

// Function that takes a pointer to a Car struct and updates the year
void updateYear(struct Car *c) {
  c->year = 2025;  // Change the year
}

int main() {
  struct Car myCar = {"Toyota", 2020};

  updateYear(&myCar);  // Pass a pointer so the function can change the year

  printf("Brand: %s\n", myCar.brand);
  printf("Year: %d\n", myCar.year);

  return 0;  }
```

Why Use Struct Pointers?

Structs ke saath pointers use karna useful hota hai jab:

1. **Aapko large amount of data copy karne se bachna ho.**
   Poora struct copy karne ke bajaye, sirf pointer pass kar sakte ho. Isse program fast chalta hai aur kam memory use hoti hai.
2. **Aapko function ke andar values change karni ho.**
   Agar aap ek struct pointer ko function mein pass karte ho, toh function directly original values ko change kar sakta hai.
3. **Aapko structs ko dynamically create karna ho (memory allocation ke through).**
   Pointers ke saath aap `malloc()` ka use karke program ke run time par structs create kar sakte ho.
   (Aap memory management ke baare mein aage ke chapter mein seekhoge.)

💡 **Tip:** Agar aap bade programs ya multiple values ke saath kaam kar rahe ho, toh struct pointers aapka code zyada clean aur efficient bana dete hain.

C Unions

C Unions

Ek **union** bhi struct ke jaise hota hai — matlab yeh bhi different data types ke members ko store kar sakta hai.

Lekin, **struct aur union mein kuch important differences hote hain:**

- **Struct** mein har member ke liye *alag memory* hoti hai.
- **Union** mein *saare members ek hi memory share karte hain*, matlab ek time par sirf ek value hi sahi hoti hai.

Zyada tar cases mein hum **structs** ka use karte hain, kyunki unmein ek saath multiple values store aur access ki ja sakti hain — jo most common requirement hoti hai.

Lekin **unions** tab useful hote hain jab aapko ek hi time par sirf ek type ka data store karna ho, aur memory save karni ho.

Declare a Union

Ek union banane ke liye `union` keyword ka use karte hain, aur usse variable declare karte hain (bilkul structs ki tarah):

Example

```
union myUnion {          // Union declaration
  int myNum;             // Member (int)
  char myLetter;         // Member (char)
  char myString[30];     // Member (char array)
};

int main() {
  union myUnion u1;  // Create a union variable with the name "u1":
  return 0;
}
```

Access Union Members

Structs ki tarah, union ke members ko access karne ke liye . (dot) syntax ka use hota hai.

⚠ **Important:** Kyunki saare members same memory share karte hain, agar aap ek member ka value change karte ho, toh doosre members ka value affect ho jata hai. Sirf *last assigned* member ka value valid hota hai.

Example

```
union myUnion {
  int myNum;
  char myLetter;
  char myString[30];
};

int main() {
  union myUnion u1;

  u1.myNum = 1000;

  // Since this is the last value written to the union, myNum no
longer holds 1000 - its value is now invalid
  u1.myLetter = 'A';

  printf("myNum: %d\n", u1.myNum); // This value is no longer
reliable
  printf("myLetter: %c\n", u1.myLetter); // Prints 'A'

  return 0;
}
```

Size of a Union

Union ka size hamesha uske **largest member** ke size ke barabar hota hai.

Example

```
union myUnion {
  int myNum;
  char myString[36];
};

int main() {
  union myUnion u1;
  printf("Size of union: %zu bytes\n", sizeof(u1));
  return 0;
}
```

Yahan, kyunki **myString** sabse bada member hai (36 bytes), toh poore union ka size bhi **36 bytes** hoga.

Agar yeh **struct** hota, toh size 40 bytes hota:
`myNum (4 bytes) + myString (36 bytes) = 40 bytes total.`

When to Use Unions

Unions ka use tab karein jab:

- Aapko **different types of data** ko *same memory location* mein store karna ho.
- Aap **ek time par sirf ek hi type** ka data use karte ho.
- **Memory saving** aapke program ke liye bahut important ho.

Tip: Unions memory-efficient programs ke liye helpful hote hain, jaise embedded systems, low-level hardware programming, ya aise cases jahan limited memory available hoti hai.

C typedef

C typedef

`typedef` keyword ka use aapko ek naye naam (alias) ko create karne ke liye hota hai, jo ek existing type ke liye hota hai. Isse complex declarations ko samajhna aasan ho jata hai, aur aapka code maintain karna bhi asaan hota hai.

For example, agar hum hamesha `float` likhte hain, toh hum ek naya type **Temperature** bana sakte hain jisse code zyada clear ho jata hai:

Example

```
#include <stdio.h>

typedef float Temperature;

int main()
 {
  Temperature today = 25.5;

  Temperature tomorrow = 18.6;

  printf("Today: %.1f C\n", today);
```

```
  printf("Tomorrow: %.1f C\n", tomorrow);
  return 0;
}
```

Yahan, **Temperature** bas `float` ka dusra naam hai. Lekin, code zyada expressive ho gaya hai: yeh batata hai ki yeh values temperature hain, sirf koi bhi floating-point numbers nahi.

Why Use typedef?

1. **Code ko simplify karta hai**: Type names short aur easily readable ho jaate hain.
2. **Clarity ko improve karta hai**: Intent ko zyada achhe se express karta hai (jaise **AGE** ka use instead of just `int`). Isse confusion avoid hoti hai jab bohot saare variables ek hi base type share karte hain (jaise `float` ya `double`).

Good to Know: Modern C mein, `typedef` ka use generally **struct**, **enum**, aur **function pointers** ke saath hota hai taaki code clean aur readable rahe.

---

typedef with struct

`typedef` ka use **struct** ke saath kaafi useful hota hai, kyunki yeh aapko har bar **struct** likhne se bachata hai:

Example

```c
#include <stdio.h>

// Without typedef:
struct Car {
  char brand[30];
  int year;
};

// With typedef:
typedef struct {
  char brand[30];
  int year;
} Car;

int main() {
  struct Car car1 = {"BMW", 1999}; // needs "struct"
  Car car2 = {"Ford", 1969}; // shorter with typedef

  printf("%s %d\n", car1.brand, car1.year);
  printf("%s %d\n", car2.brand, car2.year);
  return 0;
}
```

Yeh example dikhata hai kaise `typedef` **multiple struct variables** ke saath kaam karte waqt code ko asaan banata hai, jaise different car models:

Example

```c
#include <stdio.h>

// With typedef
```

```
typedef struct {
  char brand[30];
  char model[30];
  int year;
} Car;

int main() {
  Car car1 = {"BMW", "X5", 1999};
  Car car2 = {"Ford", "Mustang", 1969};
  Car car3 = {"Toyota", "Corolla", 2011};

  printf("%s %s %d\n", car1.brand, car1.model, car1.year);
  printf("%s %s %d\n", car2.brand, car2.model, car2.year);
  printf("%s %s %d\n", car3.brand, car3.model, car3.year);

  return 0;
}
```

Yahan aap dekh sakte hain ki `typedef` kaise code ko clean rakhta hai jab **structs** ko doosre **structs** ke andar nest kiya jata hai:

Example

```
#include <stdio.h>

// Define three structs with typedef
typedef struct {
  char firstName[20];
  char lastName[20];
} Owner;

typedef struct {
  char brand[20];
  int year;
  Owner owner;
} Car;
```

```
typedef struct {
  char name[30];
  Car featuredCar;
} Dealership;

int main() {
  Owner person = {"John", "Doe"};
  Car car1 = {"Toyota", 2010, person};
  Dealership d = {"City Motors", car1};

  printf("Dealership: %s\n", d.name);
  printf("Featured Car: %s (%d), owned by %s %s\n",
    d.featuredCar.brand,
    d.featuredCar.year,
    d.featuredCar.owner.firstName,
    d.featuredCar.owner.lastName);

  return 0;
}
```

Conclusion

Aapko **typedef** ka use karna hai ya nahi, yeh aapke upar hai. Aapka code bina `typedef` ke bhi kaam karega. Lekin, modern C mein **typedef** ka use code ko **shorter**, **clearer**, aur **easier to maintain** banane ke liye kiya jata hai.

C Struct Alignment and Padding

Struct Padding

Jab aap C mein ek struct banate ho, compiler kabhi-kabhi members ke beech extra padding bytes add kar sakta hai.

Yeh padding isliye hoti hai taaki program ko aapke computer par efficiently run karaya ja sake, kyunki most CPUs tab data ko zyada efficiently read karte hain jab data memory mein sahi tareeke se aligned hota hai.

Aapko iske baare mein jyada sochne ki zarurat tabhi hoti hai jab aap low-level memory ya file formats ke saath kaam kar rahe hote hain.

Short me: Padding ka matlab hai ki compiler kabhi-kabhi struct ke andar empty spaces add kar deta hai taaki data fast aur sahi tareeke se memory mein aligned ho.

Chaliye ek simple struct dekhte hain:

Example

```
struct Example {
  char a; // 1 byte
  int b;  // 4 bytes
  char c; // 1 byte
```

```
};

int main() {
  printf("Size of struct: %zu bytes\n", sizeof(struct Example));
  return 0;
}
```

Aap expect karenge ki size 1 + 4 + 1 = 6 bytes ho, lekin yeh usually **12 bytes** print hota hai!

Kyun?

Compiler padding bytes add karta hai taaki `int` type (b) ka member ek aise memory address par start ho jo 4 ke multiple ho. Isse CPU ko usse jaldi read karne mein madad milti hai.

Yeh memory actually kis tarah arranged hoti hai:

Member	Bytes	Notes
a	1	Stored first
padding	3	Added so `b` starts at a multiple of 4
b	4	Aligned to 4-byte boundary
c	1	Stored next
padding	3	Added to make total size a multiple of 4
Total	12	**1 + 3 + 4 + 1 + 3 = 12 bytes**

How to Reduce Padding

Padding ka size depend karta hai ki struct ke members ko kis order mein rakhte ho. Agar aap larger types ko pehle group karte ho, toh struct ka size kam ho sakta hai:

Example

```
struct Example {
  int b;  // 4 bytes
  char a; // 1 byte
  char c; // 1 byte
};

int main() {
  printf("Size of struct: %zu bytes\n", sizeof(struct Example));

 // Usually 8 bytes
  return 0;
}
```

Ab struct ka size **12 bytes** ki jagah **8 bytes** ho gaya, kyunki padding ki zarurat kam padti hai.

Why It Matters

1. **Padding se programs fast chalte hain** kyunki data memory mein sahi alignment pe rehta hai.
2. Yeh struct ko **expected size se zyada bada** bana sakta hai.
3. **Reordering members** padding ko reduce karne aur memory bachane mein madad karta hai.
   Note: Aapko is baare mein zyada sochne ki zarurat nahi hoti, jab

tak aap low-level memory ya file formats ke saath kaam nahi kar rahe hote.

Structs vs Unions

Ab tak humne structs mein padding ke baare mein baat ki. Lekin **unions** ka kya hota hai?

Unions mein saare members ek hi memory location mein store hote hain, isliye unmein members ke beech padding nahi hota. Lekin **structs aur unions** dono ko alignment rules follow karne padte hain — data ko aise memory address par start hona chahiye jo unke type size ke multiple ho.

Example

```
struct S {
  char a;
  int b;
  char c;
};

union U {
  char a;
  int b;
  char c;
};
```

```c
int main() {
  printf("Struct size: %zu\n", sizeof(struct S));
  printf("Union size: %zu\n", sizeof(union U));
  return 0;
}
```

Typical result:

```
Struct size: 12
Union size: 4
```

Explanation:

- **Struct** - Members ek ke baad ek store hote hain, isliye unke beech padding add hota hai.
- **Union** - Sabhi members same memory share karte hain, isliye padding nahi hota. Sirf largest member ka size total size decide karta hai.

Feature	Struct	Union
Members stored	One after another	All share the same memory space
Padding between members	Yes	No
Aligned to	Each member's type	Largest member's type
Total size	Sum of members + padding	Size of largest member

Tip:

Aapko generally is baare mein zyada sochne ki zarurat nahi hoti, lekin yeh jaana achha hai ki **structs kabhi-kabhi jyada memory lete hain** expected size ke comparison mein, jabki **unions** ka size fixed hota hai.

Summary

1. Struct members apne data type ke size ke hisaab se align hote hain.
2. Compiler padding bytes add karta hai taaki data ko sahi tareeke se align kiya ja sake.
3. Struct ka total size aksar uske members ke sum se zyada hota hai.
4. Members ko reorder karne se padding ko reduce kiya ja sakta hai aur memory bachayi ja sakti hai.

C Enumeration (enum)

C Enumeration (enum)

Enum ek special data type hai jo ek group of constants (unchangeable values) ko represent karta hai.

Enum Kaise Banayein?

Enum banane ke liye `enum` keyword ka use karte hain, fir enum ka naam likhte hain, aur enum ke items ko comma se separate karte hain:

```
enum Level {
  LOW,
  MEDIUM,
  HIGH
};
```

Note karo ki last item ke baad comma dena zaroori nahi hai.

Uppercase ka use karna zaroori nahi hai, lekin yeh ek achhi practice mana jata hai.

"Enum" ka matlab hai "**enumerations**", jo ka matlab hai "specifically listed" (jo list mein diya gaya ho).

Enum Ka Use Kaise Karein?

Enum ko use karne ke liye, aapko uska ek variable banana padta hai.

`main()` function ke andar, `enum` keyword, fir enum ka naam (jaise `Level`), aur phir enum ka variable name (jaise `myVar`) specify karte hain:

```
enum Level myVar;
```

Ab aap `myVar` variable ko assign kar sakte ho, lekin assigned value enum ke kisi bhi item se honi chahiye (jaise `LOW`, `MEDIUM`, ya `HIGH`):

```
enum Level myVar = MEDIUM;
```

By default, pehla item (`LOW`) ka value 0 hota hai, doosra (`MEDIUM`) ka value 1, aur teesra (`HIGH`) ka value 2 hota hai. Agar aap `myVar` ko print karoge, to output `1` hoga, kyunki `MEDIUM` ka value 1 hai:

```c
int main() {
  // Create an enum variable and assign a value to it
  enum Level myVar = MEDIUM;

  // Print the enum variable
  printf("%d", myVar);

  return 0;
}
```

Enum Ke Values Ko Change Karna

Jaisa ki aap jaante ho, enum ka pehla item ka default value 0 hota hai, doosra ka 1, aur aise hi aage.

Agar aap chaahte ho ki values ko zyada meaningful banayein, to aap easily unhe change kar sakte ho:

```c
enum Level {
  LOW = 25,
  MEDIUM = 50,
  HIGH = 75
};
printf("%d", myVar); // Now outputs 50
```

Agar aap kisi specific item ko value assign karte ho, to baaki items apne aap update ho jaayenge:

```c
enum Level {
  LOW = 5,
  MEDIUM, // Now 6
```

```
  HIGH // Now 7
};
```

Enum Ko Switch Statement Mein Use Karna

Enums ko switch statements mein bhi use kiya jaata hai, jisme hum enum ke value ke basis par different cases handle kar sakte hain:

```
enum Level {
  LOW = 1,
  MEDIUM,
  HIGH
};

int main() {
  enum Level myVar = MEDIUM;

  switch (myVar) {
    case 1:
      printf("Low Level");
      break;
    case 2:
      printf("Medium level");
      break;
    case 3:
      printf("High level");
      break;
  }
  return 0;
}
```

Is example mein, `myVar` ki value `MEDIUM` hai, to output "Medium level" hoga.

Typedef Ke Saath Enum

Aap `typedef` ko enum ke saath use kar sakte ho, jisse aapko har baar `enum` likhne ki zaroorat nahi padti. Yeh kaafi convenient hota hai.

Without typedef:

```
enum Day {MON, TUE, WED, THU, FRI, SAT, SUN};
enum Day today = WED;
```

With typedef:

```
typedef enum {MON, TUE, WED, THU, FRI, SAT, SUN} Day;
Day today = WED;
```

Dono versions kaam karte hain, lekin `typedef` version zyada short aur readable hota hai. Ek example dekhte hain:

```c
#include <stdio.h>

typedef enum {MON, TUE, WED, THU, FRI, SAT, SUN} Day;

int main() {
  Day today = WED;
  if (today == WED) {
    printf("It is Wednesday!\n");
  }
  return 0;
}
```

Enums Ko Kab Aur Kyu Use Karein?

Enums ka use constants ko meaningful naam dene ke liye kiya jaata hai, jisse code ko samajhna aur maintain karna easy hota hai.

Aapko enums tab use karna chahiye jab aapke paas aise values ho jo change nahi hongi, jaise:

- Din (Days of the week)
- Mahine ke din (Days of the month)
- Colors
- Deck of cards
- Severity levels (Low, Medium, High)

Enums ka use karke aap code mein magic numbers ko eliminate kar sakte ho aur values ko zyada descriptive bana sakte ho, jisse aapka code zyada readable aur maintainable ho jaata hai.

Here's the **C Memory Management** explanation in **Hinglish**:

C Memory Management

Memory management is the process of handling how much memory a program uses through different operations.

Memory in C

C mein memory ka kaam samajhna zaroori hai. Jab aap koi basic variable banate ho, toh C automatically us variable ke liye memory reserve kar leta hai. Example ke liye:

- Ek `int` variable typically **4 bytes** memory use karta hai.
- Ek `double` variable typically **8 bytes** memory use karta hai.

Aap `sizeof` operator ka use karke different types ki size find kar sakte ho:

Example:

```
int myInt;
float myFloat;
double myDouble;
char myChar;

printf("%zu\n", sizeof(myInt));      // 4 bytes
printf("%zu\n", sizeof(myFloat));    // 4 bytes
printf("%zu\n", sizeof(myDouble));   // 8 bytes
printf("%zu\n", sizeof(myChar));     // 1 byte
```

Yeh jaana kyun zaroori hai?

Agar aapka program zyada memory use kare, ya unnecessary memory allocate kare, toh usse program ki performance slow ho sakti hai.

C mein aapko khud memory manage karni padti hai, jo ek thoda complicated kaam hai. Lekin agar sahi tareeke se kiya jaye, toh yeh kaafi powerful hota hai. Proper memory management se program ki performance improve hoti hai. Isliye yeh important hai ki aap memory ko release karna sikhein jab woh zaroori na ho, aur sirf utni memory use karein jitni task ke liye required ho.

Pointers aur Memory Addresses

Pehle chapters mein aapne memory addresses aur pointers ke baare mein padha hai. Inka memory management mein important role hota hai, kyunki aap pointers ke through directly memory ko access kar sakte ho.

Lekin careful rehna padta hai, kyunki agar aap pointers ko galat tareeke se handle karte ho, toh aap doosre memory locations mein stored data ko corrupt kar sakte ho.

Memory Management Ke Operations

Memory management mein kuch important operations hote hain:

1. **Memory Allocation**: Jab aap memory reserve karte ho kisi variable ke liye.
2. **Memory Reallocation**: Jab aap already allocated memory ka size badhate ho.
3. **Memory Deallocation (Freeing)**: Jab aap use hone ke baad memory ko free karte ho, taaki woh memory dusre processes ke liye available ho sake.

Agle chapters mein hum yeh operations detail mein samjhayenge, jismein aap sikhenge ki kis tareeke se memory ko allocate, reallocate aur free kiya jata hai.

Summary

Memory management C programming ka ek essential hissa hai, jo program ke performance ko optimize karne mein madad karta hai. Agar aap properly memory allocate aur free karte ho, toh aapka program efficient rahega aur memory leaks aur performance issues se bach sakte hain.

C Allocate Memory

The process of reserving memory is called allocation. The way to allocate memory depends on the type of memory.

C has two types of memory: Static memory and dynamic memory.

Static Memory

Static memory woh memory hoti hai jo program ke run hone se pehle reserve hoti hai. Isko **compile-time memory allocation** bhi kaha jata hai.

Jab aap koi variable create karte ho, C automatically uske liye memory allocate kar deta hai.

Example:

```
int students[20];
printf("%zu", sizeof(students)); // 80 bytes
```

Is example mein, **students** array ke liye **20 elements** reserve kiye jaate hain, jo typically **80 bytes** (20 * 4) ki memory use karta hai.

Lekin agar semester ke dauran, sirf **12 students** attend karte hain, toh baaki 8 elements ki memory waste ho gayi. Matlab aapko unnecessary space mil gaya.

Yeh program ko crash nahi karega, par agar aise kaafi code ho, toh program thoda **slow** ho sakta hai.

Agar aapko better control chahiye memory par, toh **Dynamic Memory** ka use karke aap yeh manage kar sakte ho.

Dynamic Memory

Dynamic memory woh memory hoti hai jo program ke run hone ke baad allocate hoti hai. Isko **runtime memory allocation** bhi kaha jata hai.

Static memory ke mukable dynamic memory mein aapke paas full control hota hai ki kitni memory aap use kar rahe ho. Aap code likh kar decide kar sakte ho ki aapko kitni memory ki zarurat hai aur usse allocate kar sakte ho.

Dynamic memory kisi variable ka part nahi hoti; aap usse **pointers** ke through access kar sakte ho.

Dynamic memory allocate karne ke liye **malloc()** ya **calloc()** functions ka use hota hai. In functions ko use karne ke liye **<stdlib.h>** header file include karni padti hai.

```
int *ptr1 = malloc(size);
int *ptr2 = calloc(amount, size);
```

- **malloc()** ek parameter leta hai, jo size hota hai (kitni memory allocate karni hai, bytes mein).
- **calloc()** do parameters leta hai:
  - **amount** - Kitni cheezein allocate karni hain.
  - **size** - Har item ka size (bytes mein).

Note:

- **malloc()** se allocated memory ka data unpredictable hota hai. Isliye us memory mein kuch likhna zaroori hai usse pehle, taaki unexpected values na aaye.
- **calloc()** memory ko zeroes se initialize karta hai, jo malloc se thoda kam efficient hota hai, lekin memory ko clear karne ke liye helpful hai.

Best practice yeh hai ki aap **sizeof** operator ka use karein:

```
int *ptr1, *ptr2;
ptr1 = malloc(sizeof(*ptr1));
ptr2 = calloc(1, sizeof(*ptr2));
```

Warning:
sizeof(ptr1)* se C ko yeh bataya jata hai ki pointer ke address par jo data store hai uska size measure karo. Agar aap **sizeof(ptr1) likhenge, toh yeh pointer ka size (usually 8 bytes) measure karega, na ki data ka size.

Note:
`sizeof` operator dynamic memory ke liye size nahi batata. Agar aap 5 **float** values ke liye memory reserve karte hain, toh **sizeof** sirf ek float ka size (4 bytes) dikhayega, puri 5 values ka size nahi.

Example with Dynamic Memory

Pehle wale students example ko dynamic memory ke saath improve karte hain:

```
int *students;
int numStudents = 12;
students = calloc(numStudents, sizeof(*students));
printf("%d", numStudents * sizeof(*students)); // 48 bytes
```

Is example mein humne **12 students** ke liye memory allocate ki hai, aur `calloc` function ka use kiya hai. Isme **numStudents** aur **sizeof(students)* ko multiply karke total memory size (48 bytes) calculate kiya gaya hai.

Notes

- Jab aap dynamic memory allocate karte hain, toh aapko **errors check** karna bhi zaroori hota hai. Agar memory allocate nahi ho rahi hai toh error handle karna padega.
- Program ke end mein aapko allocated memory ko **free** karna zaroori hai. Aap `free()` function ka use karke memory release kar sakte hain.

Stack Memory

For completeness, ek aur memory type hota hai **stack memory**, jo functions ke andar declare kiye gaye variables ke liye use hoti hai.

Jab bhi koi function call hota hai, toh **stack memory** allocate hoti hai. Jaise hi function return hota hai, stack memory free ho jati hai.

Stack memory ka ek important concept hai **recursion**, jo jab repeatedly call hoti hai toh stack memory quickly consume kar sakti hai. Agar recursion bohot zyada baar repeat hota hai, toh **stack overflow** ho sakta hai.

Summary

- **Static memory** fixed hoti hai aur compile time pe allocate hoti hai.
- **Dynamic memory** run time pe allocate hoti hai aur aapko full control milta hai kitni memory allocate karni hai.
- Stack memory function ke andar use hoti hai aur recursion ke saath careful rehna padta hai, varna stack overflow ho sakta hai.

C Access Memory

Access Dynamic Memory

Dynamic memory waise hi behave karta hai jaise arrays, aur iska data type pointer ke type ke basis par decide hota hai.

Jaise arrays mein hota hai, dynamic memory ko access karne ke liye aap index number se element ko refer karte hain:

```
ptr[0] = 12;
```

Aap pointer ko dereference karke pehle element ko access kar sakte hain:

```
*ptr = 12;
```

Example

Dynamic memory se read aur write karna:

```
// Allocate memory
int *ptr;
ptr = calloc(4, sizeof(*ptr));

// Write to the memory
*ptr = 2;
ptr[1] = 4;
ptr[2] = 6;

// Read from the memory
printf("%d\n", *ptr);
printf("%d %d %d", ptr[1], ptr[2], ptr[3]);
```

Data Types ke baare mein

Dynamic memory ka apna data type nahi hota, bas yeh bytes ka ek sequence hota hai. Memory ke andar jo data store hota hai, usse pointer ke data type ke hisaab se interpret kiya jata hai.

Is example mein, ek pointer ko 4 bytes ki space allocate karne ke baad usse ek **int** value ke roop mein ya phir 4 **char** values ke roop mein treat kiya ja sakta hai (jisme har **char** 1 byte ka hota hai).

Example

```
int *ptr1 = malloc(4);
char *ptr2 = (char*) ptr1;
ptr1[0] = 1684234849;
printf("%d is %c %c %c %c", *ptr1, ptr2[0], ptr2[1], ptr2[2],
ptr2[3]);
```

Explanation:

- Humne 4 bytes allocate ki hain `malloc(4)` se, jo ek `int` ke liye space hai (4 bytes).
- Fir humne pointer ko `char` pointer mein cast kiya, taaki har byte ko alag-alag **char** value ke roop mein dekha ja sake.
- Jab `ptr1[0]` ko `1684234849` assign kiya, toh yeh integer value har byte ke liye ek specific character code mein convert ho gayi.
- Is example mein, `*ptr1` integer value print karega, jabki `ptr2` ke through hum usi memory ko characters ke roop mein access karenge.

C Reallocate Memory

Reallocate Memory

Agar jo memory aapne reserve ki thi woh kaafi nahi hai, toh aap usko dobara allocate kar sakte hain, jisse woh zyada ho jaye.

Reallocation ka matlab hai ki jo aapne pehle memory allocate ki thi, uska size badalna. Yeh data ko safe rakhta hai jo us memory mein store tha.

Aap `realloc()` function ka use karke allocated memory ka size badal sakte hain.

`realloc()` function ko do parameters chahiye:

```
int *ptr2 = realloc(ptr1, size);
```

- Pehla parameter `ptr1` woh pointer hai jisme memory reallocate karni hai.
- Dusra parameter woh naya size hai jo aap allocate karna chahte ho, aur yeh size bytes mein hota hai.

`realloc()` function try karta hai memory ko `ptr1` ke address pe resize karne ki, aur agar yeh possible nahi hota, toh woh new address pe memory allocate kar leta hai aur naye address ko return karta hai.

Note: Jab `realloc()` function ek naya address return karta hai, toh purana address pe jo memory thi, woh ab valid nahi hoti, aur use karna unsafe ho sakta hai. Reallocation ke baad, hamesha new pointer ko purane variable mein assign kar dena chahiye taaki purana pointer accidental usage se bach sake.

Example

Memory ka size badhana:

```
int *ptr1, *ptr2, size;

// Allocate memory for four integers
size = 4 * sizeof(*ptr1);
ptr1 = malloc(size);
```

```
printf("%d bytes allocated at address %p \n", size, ptr1);

// Resize the memory to hold six integers
size = 6 * sizeof(*ptr1);
ptr2 = realloc(ptr1, size);

printf("%d bytes reallocated at address %p \n", size, ptr2);
```

NULL Pointer & Error Checking

`realloc()` function NULL pointer return karta hai agar woh zyada memory allocate nahi kar pata. Yeh bahut rare hota hai, lekin jab aapko apne code ko failproof banana ho toh is baat ko mind mein rakhna zaroori hai.

Example:

`realloc()` ke liye NULL pointer check karna:

```
int *ptr1, *ptr2;

// Allocate memory
ptr1 = malloc(4);

// Attempt to resize the memory
ptr2 = realloc(ptr1, 8);

// Check whether realloc is able to resize the memory or not
if (ptr2 == NULL) {
  // If reallocation fails
  printf("Failed. Unable to resize memory");
} else {
  // If reallocation is sucessful
  printf("Success. 8 bytes reallocated at address %p \n", ptr2);
  ptr1 = ptr2;  // Update ptr1 to point to the newly allocated
```

```
memory
}
```

Note: Jab bhi memory allocate karte hain, toh hamesha error checking karna zaroori hai (i.e., `if (pointer == NULL)`).

Note: Jab aap memory ka kaam khatam kar lete hain, toh us memory ko `free()` function ke through release karna zaroori hota hai. Yeh aapke program ko sahi se chalane mein madad karta hai aur aapke code ko maintainable aur efficient banata hai.

Example

Allocated memory ko free karna:

```
// Free allocated memory
free(ptr1);
```

Is tarah se, aap dynamic memory ko efficiently manage kar sakte hain, allocate aur reallocate kar sakte hain, aur usage ke baad release kar sakte hain.

Jab aapko ek block of memory ki ab zarurat nahi ho, toh usse deallocate karna chahiye. Deallocation ko "freeing" bhi kaha jata hai.

Dynamic memory tab tak reserved rehti hai jab tak usse deallocate na kiya jaye ya program khatam na ho jaye.

Jab memory deallocate kar di jaati hai, toh usse doosre programs ya aapke apne program ke kisi aur part ko allocate kiya ja sakta hai.

Free Memory

Memory ko deallocate karne ke liye `free()` function ka use karte hain:

```
free(pointer);
```

Yahan `pointer` parameter woh pointer hota hai jo us memory address ko point karta hai jise deallocate karna hai.

Example:

```
int *ptr;
ptr = malloc(sizeof(*ptr));  // Ek integer ke liye memory allocate karo

free(ptr);  // Memory free karo
ptr = NULL;  // Pointer ko NULL set karo, taaki accidentally use na ho
```

Yeh achi practice hai ki memory free karne ke baad pointer ko `NULL` set kar diya jaye, taaki aap accidentally us memory ko dobara use na karein.

Agar aap free ki hui memory ko dobara use karenge, toh doosre programs ya apne program ke kisi aur part ka data corrupt ho sakta hai.

Example

Ek working example jo error checking aur memory free karne ke process ko dikhata hai:

```c
int *ptr;
ptr = malloc(sizeof(*ptr)); // Allocate memory for one integer

// If memory cannot be allocated, print a message and end the main() function
if (ptr == NULL) {
  printf("Unable to allocate memory");
  return 1;  // Exit the program with an error code
}

// Set the value of the integer
*ptr = 20;

// Print the integer value
printf("Integer value: %d\n", *ptr);

// Free allocated memory
free(ptr);

// Set the pointer to NULL to prevent it from being accidentally used
ptr = NULL;
```

Memory Leaks

Memory leak tab hota hai jab dynamic memory allocate ki jati hai lekin free nahi ki jaati.

Agar memory leak kisi loop ya frequently call hone wali function mein hota hai, toh yeh bohot zyada memory consume kar sakta hai aur aapke computer ko slow kar sakta hai.

Agar dynamic memory ka pointer lose ho jata hai bina usse free kiye, toh bhi memory leak ho sakta hai. Yeh accidental ho sakta hai, isliye pointer ko track karna bohot zaroori hota hai.

Yahan kuch examples hain jahan pointer lose ho sakta hai:

Example 1

Pointer overwrite ho jata hai:

```
int x = 5;
int *ptr;
ptr = calloc(2, sizeof(*ptr));
ptr = &x;
```

Is example mein, jab pointer ko `x` ke address pe point karne ke liye set kiya gaya, tab `calloc()` se allocated memory ka access kho diya gaya.

Example 2

Pointer sirf function ke andar hota hai:

```
void myFunction() {
  int *ptr;
  ptr = malloc(sizeof(*ptr));
}

int main() {
  myFunction();
  printf("The function has ended");
  return 0;
}
```

Is example mein, function ke andar jo memory allocated thi, woh function ke end hone ke baad bhi allocated rehti hai, lekin uska access ab possible nahi hota. Is problem ko prevent karne ke liye, memory ko function ke end hone se pehle free karna chahiye.

Example 3

Pointer reallocation ke failure ke baad kho jata hai:

```
int* ptr;
ptr = malloc(sizeof(*ptr));
ptr = realloc(ptr, 2*sizeof(*ptr));
```

Agar `realloc()` memory ko resize nahi kar pata, toh woh `NULL` pointer return karega aur original memory address ko overwrite kar dega. Is case mein, original memory ka access ab nahi ho sakta.

Summary

Summary mein, C mein memory manage karte waqt kuch best practices follow karna zaroori hai:

- **Error checking:** Hamesha check karein ki memory allocation successful thi ya nahi (`NULL` return values ke liye check karna).
- **Memory leaks ko avoid karein:** Jo memory ab use nahi ho rahi usse hamesha free karna zaroori hai, nahi toh program slow ho sakta hai ya crash bhi ho sakta hai agar memory ka shortage ho jaye.
- **Pointer ko NULL set karein after freeing:** Jab memory free kar di ho, toh pointer ko NULL set karna chahiye taaki woh accidentally dobara use na ho.

C Structures and Dynamic Memory

Structures and Dynamic Memory

Aap structures ke saath bhi dynamic memory ka use kar sakte hain.

Yeh useful hota hai jab aapko pehle se pata nahi hota ki kitni structures aapko chahiye, ya jab aap memory save karna chahte hain aur sirf utni memory allocate karna chahte hain jo zaroori ho (jaise ek car dealership program mein, jahan cars ki number fix nahi hoti).

Structure ke liye Memory Allocate Karna

Aap `malloc()` function ka use karke structure ke pointer ke liye memory allocate kar sakte hain:

Example:

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

struct Car {
  char brand[50];
  int year;
};

int main() {
  // Allocate memory for one Car struct
  struct Car *ptr = (struct Car*) malloc(sizeof(struct
Car));
```

```
  // Check if allocation was successful
  if (ptr == NULL) {
    printf("Memory allocation failed.\n");
    return 1; // Exit the program with an error code
  }

  // Set values
  strcpy(ptr->brand, "Honda");
  ptr->year = 2022;

  // Print values
  printf("Brand: %s\n", ptr->brand);
  printf("Year: %d\n", ptr->year);

  // Free memory
  free(ptr);

  return 0;
}
```

Example Explained:

- `malloc()` ek struct ke liye memory allocate karta hai.
- `strcpy()` ka use string ko brand field mein copy karne ke liye hota hai.
- `->` ka use hum pointer ke through members ko access karne ke liye karte hain.
- `free()` function memory ko release karne ke liye use hota hai.

Note: `malloc()` se allocate ki gayi memory uninitialized hoti hai. Iska content undefined hota hai jab tak aap values assign nahi karte. Agar aap

chahein ki memory zero se initialized ho, toh `calloc()` use kar sakte hain.

Arrays of Structs ka Use Karna

Aap multiple structs ke liye ek saath memory allocate kar sakte hain, jaise ek array:

Example: 3 cars ke liye memory allocate karna

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

struct Car {
  char brand[50];
  int year;
};

int main() {
  struct Car *cars = (struct Car*) malloc(3 * sizeof(struct Car));

  if (cars == NULL) {
    printf("Memory allocation failed.\n");
    return 1 // Exit the program with an error code;
  }

  // Fill the data
  strcpy(cars[0].brand, "Ford");
  cars[0].year = 2015;

  strcpy(cars[1].brand, "BMW");
  cars[1].year = 2018;
```

```
  strcpy(cars[2].brand, "Volvo");
  cars[2].year = 2023;

  // Print the data
  for (int i = 0; i < 3; i++) {
    printf("%s - %d\n", cars[i].brand, cars[i].year);
  }

  free(cars);
  return 0;
}
```

Arrays ko `realloc()` se Grow Karna

Agar aapko baad mein aur elements chahiye, toh aap apne dynamic array ko `realloc()` ke saath resize kar sakte hain. Yeh block ko new location pe move kar sakta hai aur nayi pointer return karta hai. Hamesha nayi pointer ko temporary variable mein store karein taaki agar reallocation fail ho toh aap original memory kho na de.

Example: Structs ke array ko expand karna

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

struct Car {
  char brand[50];
  int year;
};

int main() {
  int count = 2;
```

```
  struct Car *cars = (struct Car*) malloc(count * sizeof(struct
Car));
  if (cars == NULL) {
    printf("Initial allocation failed.\n");
    return 1;
  }

  // Initialize first 2 cars
  strcpy(cars[0].brand, "Toyota"); cars[0].year = 2010;
  strcpy(cars[1].brand, "Audi");   cars[1].year = 2019;

  // Need one more car -> grow to 3
  int newCount = 3;
  struct Car *tmp = (struct Car*) realloc(cars, newCount *
sizeof(struct Car));
  if (tmp == NULL) {
    // 'cars' is still valid here; free it to avoid a leak
    free(cars);
    printf("Reallocation failed.\n");
    return 1;
  }
  cars = tmp;  // use the reallocated block

  // Initialize the new element at index 2
  strcpy(cars[2].brand, "Kia");
  cars[2].year = 2022;

  // Print all cars
  for (int i = 0; i < newCount; i++) {
    printf("%s - %d\n", cars[i].brand, cars[i].year);
  }

  free(cars);
  return 0;
}
```

Summary:

- **Dynamic memory allocation** ka use aap structures ke liye bhi kar sakte hain jab aapko number of structures ka pata na ho.
- **malloc()** se ek struct ke liye memory allocate karte hain, aur **realloc()** se aap existing memory block ko expand kar sakte hain.
- **free()** ka use memory ko release karne ke liye hota hai jab aapka kaam complete ho jaye.

C Memory Management Example

Real-Life Memory Management Example

Yeh example dynamic memory ka practical use dikhata hai, jisme hum ek list banate hain jo kisi bhi length ki ho sakti hai. C ke regular arrays ki length fix hoti hai aur unko change nahi kiya ja sakta, lekin dynamic memory ke saath hum apni list ko jitna lamba chahen utna bana sakte hain.

Example

```
struct list {
  int *data; // Points to the memory where the list items are stored
  int numItems; // Indicates how many items are currently in the
list
  int size; // Indicates how many items fit in the allocated memory
};

void addToList(struct list *myList, int item);
```

```c
int main() {
  struct list myList;
  int amount;
  int i, j;

  // Create a list and start with enough space for 10 items
  myList.numItems = 0;
  myList.size = 10;
  myList.data = malloc(myList.size * sizeof(int));

  // Find out if memory allocation was successful
  if (myList.data == NULL) {
    printf("Memory allocation failed");
    return 1; // Exit the program with an error code
  }

  // Add any number of items to the list specified by the amount
variable
  amount = 44;
  for (i = 0; i < amount; i++) {
    addToList(&myList, i + 1);
  }
  // Display the contents of the list
  for (j = 0; j < myList.numItems; j++) {
    printf("%d ", myList.data[j]);
  }
  // Free the memory when it is no longer needed
  free(myList.data);
  myList.data = NULL;

  return 0;
}

// This function adds an item to a list
void addToList(struct list *myList, int item) {
```

```
  // If the list is full then resize the memory to fit 10 more items
  if (myList->numItems == myList->size) {
    int newSize = myList->size + 10;

    // Use a temporary pointer so we don't lose the original on
failure
    int *tmp = realloc(myList->data, newSize * sizeof(int));
    if (tmp == NULL) {
      printf("Memory resize failed\n");
      return; // Leave the list unchanged
    }

    // Only update fields after a successful reallocation
    myList->data = tmp;
    myList->size = newSize;
  }

  // Add the item to the end of the list
  myList->data[myList->numItems] = item;
  myList->numItems++;
}
```

Example Explained

Is example mein teen parts hain:

1. `myList` **structure** jo list ke data ko contain karta hai.
2. `main()` **function** jisme program run hota hai.
3. `addToList()` **function** jo list mein item add karta hai.

`myList` Structure

`myList` structure list ke baare mein sab information rakhta hai:

- `data`: Ye pointer hai jo dynamic memory ko point karta hai, jahan list ke contents store hote hain.
- `numItems`: Yeh batata hai ki list mein kitne items hain.
- `size`: Yeh batata hai ki allocated memory mein kitne items fit ho sakte hain.

Hum structure use karte hain taaki saari information ek saath pass ho sake functions mein.

`main()` Function

List ko initialize karo aur 10 items ke liye memory reserve karo:

```
myList.numItems = 0;
myList.size = 10;
myList.data = malloc(myList.size * sizeof(int));
```

`myList.numItems = 0`: List initially empty hai.

`myList.size = 10`: Humne 10 items ke liye memory reserve ki hai.

`malloc()` se memory allocate ki jati hai.

Memory allocation check karo:

```
// Find out if memory allocation was successful
if (myList.data == NULL) {
  printf("Memory allocation failed");
  return 1; // Exit the program with an error code
}
```

44 items list mein add karo:

```
// Add any number of items to the list specified by the amount variable
amount = 44;
for (i = 0; i < amount; i++) {
  addToList(&myList, i + 1);
}
```

List ko print karo:

```
// Display the contents of the list
for (j = 0; j < myList.numItems; j++) {
  printf("%d ", myList.data[j]);
}
```

Memory ko free karo jab use nahi ho:

```
free(myList.data);
myList.data = NULL;
```

`addToList()` Function

Yeh function list mein item add karta hai. Iska kaam hai:

Check karna ki list full hai ya nahi:

```
if (myList->numItems == myList->size) {
  int newSize = myList->size + 10;
```

Agar list full ho gayi hai (i.e., `numItems == size`), toh memory ko 10 items aur reserve karna padta hai.

Memory ko resize karo using `realloc()`:

```
// If the list is full then resize the memory to fit 10 more items
if (myList->numItems == myList->size) {
  int newSize = myList->size + 10;

  // Use a temp pointer so we don't lose the original on failure
  int *tmp = realloc(myList->data, newSize * sizeof(int));
  if (tmp == NULL) {
    printf("Memory resize failed\n");
    return; // Leave the list unchanged
  }

  // Only update fields after a successful reallocation
  myList->data = tmp;
  myList->size = newSize;
}
```

`realloc()` memory ko resize karta hai.

Agar `realloc()` **fail ho jata hai**, original memory ko loss hone se bachane ke liye temporary pointer (`tmp`) use kiya gaya hai.

Item ko list ke end mein add karo:

```
myList->data[myList->numItems] = item;

myList->numItems++;
```

Why 10 Items at a Time?

Memory allocation aur performance ka balance zaroori hota hai. Agar hum bohot zyada memory allocate karenge toh space waste ho sakta hai, lekin agar hum memory ko frequently resize karenge toh efficiency low ho sakti hai.

Humne yahan **10 items** ka choice kiya hai because yeh ek reasonable balance hai, lekin agar aapko pata ho ki exact number of items kitne honge, toh aap utni memory ek hi baar allocate kar sakte hain.

Summary:

- **Dynamic Memory** ke through hum ek list ko flexible size de sakte hain.
- `malloc()` se memory allocate karte hain, `realloc()` se memory ko resize karte hain aur `free()` se memory release karte hain.
- `addToList()` **function** memory ko resize karta hai agar list full ho jaye, aur item ko list ke end mein add karta hai.

C Errors

Errors

Even experienced C developers galtiyaan karte hain. Key ye hai ki aap seekhein kaise unhe spot aur fix karna hai!

Ye pages common errors aur helpful debugging tips cover karte hain, taaki aap samajh sakein ki problem kahan hai aur kaise solve karni hai.

Common Compile-Time Errors

Compile-time errors wo mistakes hoti hain jo aapke program ko compile hone se rok deti hain.

1) Missing semicolon:

Example

```
#include <stdio.h>

int main() {
  int x = 5
  printf("%d", x);
  return 0;
}
```

Result:

```
error: expected ',' or ';' before 'printf'
```

---

2) Using undeclared variables:

Example

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

```
  printf("%d", myVar);
  return 0;
}
```

Result:

```
error: 'myVar' undeclared
```

---

3) Mismatched types (e.g. ek string ko int mein assign karna):

Example

```
#include <stdio.h>

int main() {
  int x = "Hello";
  return 0;
}
```

Result:

```
error: initialization makes integer from pointer without a cast
```

Common Runtime Errors

Runtime errors tab hote hain jab program compile ho jata hai lekin run karte waqt crash karta hai ya galat behave karta hai.

1) Dividing by zero:

Example

```
#include <stdio.h>

int main() {
  int x = 10;
  int y = 0;
  int result = x / y;
  printf("%d\\n", result); // not possible
  return 0;
}
```

---

2) Accessing out-of-bounds array elements:

Example

```
#include <stdio.h>

int main() {
  int numbers[3] = {1, 2, 3};
  printf("%d\\n", numbers[8]); // element does not exist
  return 0;
}
```

---

3) Using freed memory:

Example

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

```c
int main() {
  int* ptr = malloc(sizeof(int));
  *ptr = 10;
  free(ptr);
  printf("%d\\n", *ptr); // Undefined behavior - accessing memory that was freed
  return 0;
}
```

Good Habits to Avoid Errors

- Hamesha variables ko initialize karo
- Meaningful variable names use karo
- Code ko clean rakho aur indentation use karo taaki organized rahe
- Functions ko short aur focused rakho
- Check karo ki loops aur conditions sahi se run ho rahe hain ya nahi
- Error messages dhyaan se padho — wo aksar batate hain problem kahan hai

C Debugging

Debugging

Debugging ek process hai jisme aap apne program ke errors (bugs) ko dhoondhte aur fix karte ho.

Bugs wo mistakes hoti hain jo aapke program ko crash kar deti hain, galat behave karwati hain, ya wrong output deti hain.

Debugging start karne se pehle, ensure karo ki aapka code clean aur organized hai:

- Proper indentation use karo taaki structure clear rahe.
- Variables ko clear, meaningful names do jo batayein wo kya store kar rahe hain.
- Clean code read karna easy hota hai – aur debug karna bhi!

Neeche diye gaye sections mein hum kuch common debugging techniques dekhenge.

---

1. Print Debugging

`printf()` ka use karo alag-alag points par values print karne ke liye taaki samajh pao problem kahan ho rahi hai:

```
int x = 10;
int y = 0;
printf("Before division\n"); // Debug output
int z = x / y; // Crashes (division by zero)
printf("After division\n"); // Never runs
```

Agar aapko “After division” print nahi hota dikha, iska matlab program `x / y` par crash ho gaya.

---

2. Check Variable Values

Variables print karo taaki pata chale kya wo expected values rakh rahe hain:

```
int x = 10;
int y = 5;
int result = x - y;

printf("Result: %d\n", result); // Result: 5
```

Expected 15? Matlab logic galat hai: `x + y` try karo instead.

---

3. Use a Debugger Tool

Visual Studio, Code::Blocks, aur VS Code jaise IDEs mein built-in debuggers hote hain jo C ke saath kaam karte hain. Ye tools aapko allow karte hain:

- Apne program ko pause karna using **breakpoints**
- Code ko line by line step through karna
- Variables ko watch karna aur dekhna unki unexpected values change hoti hui

Tip: Pehle `printf()` debugging se start karo. Jab comfortable ho jao, apne IDE ka debugger explore karo taaki aur deep insights mil sakein.

---

4. Learn from Error Messages

C compiler aur runtime errors aksar batate hain kya galat hua aur kahan. Example:

```
error: expected ';' before 'printf'
```

Simple solution: Missing semicolon fix karo!

Debugging with Safe Checks

Kuch bugs, jaise dividing by zero (jo is page ke first example mein hai), crash kar dete hain.

Agar aapko pata hai koi operation fail ho sakta hai, toh pehle hi check laga lo aur crash avoid karo:

Example

```
int main() {
  int x = 10;
  int y = 0;

  printf("Before division\n");

  if (y != 0) { // // Check that y is not zero before dividing
    int z = x / y;
    printf("Result: %d\n", z);
  } else {
    printf("Error: Division by zero!\n"); // // Print error message
instead of crashing
  }

  printf("After division\n");
  return 0;
}
```

Ab crash hone ke bajaye, program ek error message print karega aur run karta rahega. Ye safe aur debuggable code likhne ka important part hai.

Another Example - Out-of-Bounds Array Access

Array ke bahar elements access karna ek aur common mistake hai.

Neeche ke example mein, hum `printf()` use karte hain index value check karne ke liye use karne se pehle:

Example

```
int main() {
  int numbers[3] = {10, 20, 30};
  int index = 5;

  printf("Index = %d\n", index);
  if (index >= 0 && index < 3) { // Make sure the index is within
the valid range (0 to 2)
    printf("Value = %d\n", numbers[index]);
  } else {
    printf("Error: Index out of bounds!\n");
  }

  return 0;
}
```

Summary

- `printf()` ka use karo values print karne aur apne code ko trace karne ke liye.
- Error messages dhyaan se padho – wo aksar batate hain kya galat hua aur kahan.
- Checks add karo (jaise `if (y != 0)`) taaki crash hone se pehle hi problem handle ho jaye.
- Jab ready ho jao, IDE ka debugger use karo deep debugging ke liye.
- Debugging aapko apna program achhi tarah samajhne aur problems fast fix karne mein madad karti hai.

Error Handling

Debugging ka matlab hai development ke time apne code ke mistakes dhoondhna aur fix karna,
jabki **error handling** ek aisi technique hai jo program run karte waqt problems ko handle karti hai,
aur jab kuch galat hota hai toh specific code run karti hai.

C NULL

C NULL

NULL ek special value hai jo ek "null pointer" represent karti hai – yani ek pointer jo kahin point nahi kar raha hota.

Ye aapko empty ya invalid pointers use karne se bachata hai. Aap pointer ko **NULL** ke saath compare kar sakte ho taaki check kar sako ki use karna safe hai ya nahi.

Bahut saare **C** functions **NULL** return karte hain jab kuch galat hota hai. For example, `fopen()` **NULL** return karta hai agar file open nahi ho paati, aur `malloc()` **NULL** return karta hai agar memory allocation fail ho jaaye.
Hum isse ek `if` statement mein check kar sakte hain, aur agar kuch galat hota hai toh ek error message print kar sakte hain.

Is example mein, hum ek aisi file open karne ki koshish karte hain jo exist nahi karti.
Kyuki `fopen()` fail hota hai, wo **NULL** return karta hai aur hum ek error message print karte hain:

Example

```c
#include <stdio.h>

int main() {
  FILE *fptr = fopen("nothing.txt", "r");

  if (fptr == NULL) {
    printf("Could not open file.\n");
    return 1;
  }

  fclose(fptr);
  return 0;
}
```

Agar aap bahut zyada memory allocate karne ki koshish karte ho, toh `malloc()` fail ho sakta hai aur **NULL** return karega:

Example

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

int main() {
  int *numbers = (int*) malloc(100000000000000 *
sizeof(int));

  if (numbers == NULL) {
    printf("Memory allocation failed.\n");
    return 1;
  }

  printf("Memory allocation successful!\n");

  free(numbers);
  numbers = NULL;

  return 0;
}
```

Summary

- **NULL** ek null (empty) pointer represent karta hai
- Ye signal deta hai ki pointer kahin point nahi kar raha
- Aap pointer ko NULL se compare kar sakte ho taaki check ho sake ki wo use karne ke liye safe hai
- `malloc()` aur `fopen()` jaise functions fail hone par NULL return karte hain
- **Tip:** Hamesha check karo ki pointer NULL toh nahi hai use karne se pehle.
  Ye invalid memory access se hone wale crashes ko avoid karta hai.

C Error Handling

Error Handling in C

Error handling aapko apne program mein problems detect aur respond karne mein madad karta hai — jaise koi file open na ho paana, ya memory allocate na ho paana — taaki aapka program crash na kare ya unexpected behavior na dikhaye.

C language mein kuch aur languages ki tarah built-in exception handling (like try/catch) nahi hoti.
Iske badle, C mein **return values**, **global error codes**, aur helper functions jaise **perror()** aur **strerror()** use kiye jaate hain.

Using Return Values

Pichle chapter mein aapne seekha tha ki `fopen()` jaise functions NULL return karte hain jab kuch galat hota hai.

Aap NULL ko `if` statement ke saath check kar sakte ho taaki error detect aur handle kar sako before program crash ho.

Neeche ke example mein hum ek aisi file open karne ki koshish karte hain jo exist nahi karti.
Kyuki `fopen()` fail hota hai, wo NULL return karta hai aur hum ek error message print karte hain:

Example: fopen() fails

```c
#include <stdio.h>

int main() {
  FILE *fptr = fopen("nothing.txt", "r");

  if (fptr == NULL) {
    printf("Error opening file.\\n");
    return 1;
  }

  fclose(fptr);
  return 0;
}
```

Result:

```
Error opening file.
```

Get More Details

Agar aapko aur details chahiye ki kya galat hua, toh aap `perror()` function use kar sakte ho.

Ye ek custom error message print karta hai aur uske baad last occurred error ka description deta hai:

Example: perror() with fopen()

```c
#include <stdio.h>

int main() {
  FILE *f = fopen("nothing.txt", "r");

  if (f == NULL) {
    perror("Error opening file");
    return 1;
  }

  fclose(f);
  return 0;
}
```

Result:

```
Error opening file: No such file or directory
```

Using strerror() and errno

`errno` ek global variable hai jo last failed operation ka error code store karta hai.
Aap `<errno.h>` include karke ise access kar sakte ho, aur `strerror(errno)` use karke error code ko readable message mein convert kar sakte ho:

Example: strerror()

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

int main() {
  FILE *f = fopen("nothing.txt", "r");

  if (f == NULL) {
    printf("Error: %s\n", strerror(errno));
    return 1;
  }

  fclose(f);
  return 0;
}
```

Result:

```
Error: No such file or directory
```

Common Error Codes

Error constants `<errno.h>` mein defined hote hain.
Aap `errno` ko inke saath compare kar sakte ho taaki specific issue detect ho sake:

Error Code	Meaning
ENOENT	No such file or directory
EACCES	Permission denied
ENOMEM	Not enough memory
EINVAL	Invalid argument

Example: Custom message for ENOENT

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

int main() {
  FILE *f = fopen("nothing.txt", "r");

  if (f == NULL) {
    if (errno == ENOENT) {
      printf("The file was not found.\n");
    } else {
      printf("Some other file error occurred.\n");
    }
    return 1;
  }

  fclose(f);
  return 0;
}
```

Result:

```
The file was not found.
```

Using exit() to Stop the Program

Agar aap chahte ho ki error aate hi program turant band ho jaaye, toh aap `exit()` function use kar sakte ho.
Ye aapko ek status code return karne deta hai operating system ko.

Exit codes batate hain ki program successfully finish hua ya error ke saath:

- **0** means success
- **Non-zero values** (jaise 1 ya `EXIT_FAILURE`) indicate karte hain errors

Example: Using exit() on error

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

int main() {
  FILE *f = fopen("nothing.txt", "r");

  if (f == NULL) {
    printf("Failed to open file.\n");
```

```
    exit(1);
  }

  fclose(f);
  return 0;
}
```

Result:

```
Failed to open file.
```

Common Exit Status Codes

Code	Meaning
0	Success - the program completed normally
1	Error - something went wrong
EXIT_SUCCESS	Same as 0 (defined in <stdlib.h>)
EXIT_FAILURE	Same as a non-zero error code (also in <stdlib.h>)

Tip: Aap numbers ke badle `EXIT_SUCCESS` aur `EXIT_FAILURE` use kar sakte ho taaki code readable rahe.

Example: Using EXIT_FAILURE and EXIT_SUCCESS

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

int main() {
  FILE *f = fopen("nothing.txt", "r");
```

```
  if (f == NULL) {
    perror("Could not open nothing.txt");
    exit(EXIT_FAILURE); // More readable than exit(1)
  }

  fclose(f);
  return EXIT_SUCCESS;
}
```

Result:

```
Could not open nothing.txt: No such file or directory
```

Summary

- Bahut saare C functions NULL return karte hain jab kuch galat hota hai
- `perror()` ka use karo error message print karne ke liye
- `strerror(errno)` use karo error message string ke form mein lene ke liye
- `errno` last failed action ka error code store karta hai
- Aap `errno` ko ENOENT (file not found) ya ENOMEM (not enough memory) jaise values se compare kar sakte ho
- `exit()` ka use karo program ko early stop karne ke liye jab error aaye

C Input Validation

Input Validation

Jab users C program mein data enter karte hain, toh wo kabhi-kabhi unexpected cheezein type kar sakte hain.
Input validation ensure karta hai ki input sahi hai before program aage badhe.

Validation ke bina, aapka program crash kar sakta hai ya galat result de sakta hai!

Neeche diye gaye examples simple tarike dikhate hain jisse aap check kar sakte ho ki user ka input valid hai ya nahi.

Validate Number Range

Check karo ki number allowed range (for example, 1 se 5) ke beech hai ya nahi:

Example

```c
#include <stdio.h>

int main() {
  int number; // Variable to store the user's number

  do {
    printf("Choose a number between 1 and 5: ");
    scanf("%d", &number); // Read number input
    while (getchar() != '\n'); // Clear leftover characters from
```

```
input buffer
  } while (number < 1 || number > 5); // Keep asking until number is
between 1 and 5

  printf("You chose: %d\n", number); // Print the valid number
  return 0;
}
```

Example Result:

```
Choose a number between 1 and 5: 8
Choose a number between 1 and 5: -2
Choose a number between 1 and 5: 4
You chose: 4
```

Validate Text Input

Check karo ki name empty na ho. `fgets()` use karo aur pehla character check karo:

Example

```
#include <stdio.h>

int main() {
  int number; // Variable to store the user's number

  do {
    printf("Choose a number between 1 and 5: ");
    scanf("%d", &number); // Read number input
    while (getchar() != '\n'); // Clear leftover characters from
input buffer
  } while (number < 1 || number > 5); // Keep asking until number is
between 1 and 5
```

```
  printf("You chose: %d\n", number); // Print the valid number
  return 0;
}
```

Example Result:

```
Enter your name:
Enter your name:
Enter your name: John
Hello, John
```

Validate Integer Input

Ensure karo ki user ek number enter kare.
Agar wo kuch aur (jaise letter) enter karta hai, toh firse puchho using `fgets()` aur `sscanf()`:

Example

```
#include <stdio.h>

int main() {
  int number;       // Variable to store the user's number
  char input[100];  // Buffer to hold user input as a string

  printf("Enter a number: ");

  // Keep reading input until the user enters a valid integer
  while (fgets(input, sizeof(input), stdin)) {
    // Try to read an integer from the input string
    if (sscanf(input, "%d", &number) == 1) {
      break; // Success: break out of the loop
    } else {
```

```
      printf("Invalid input. Try again: "); // If not an integer, ask again
    }
  }

  // Print the valid number entered by the user
  printf("You entered: %d\n", number);
  return 0;
}
```

Example Result:

```
Enter a number: AB
Invalid input. Try again: 3.5
Invalid input. Try again: 35
You entered: 35
```

C Date and Time

Time and Date

C mein aap `<time.h>` header ka use karke dates aur times ke saath kaam kar sakte ho.

Ye library aapko current time lene, usse format karne, aur time-related calculations perform karne ki facility deti hai.

Getting the Current Time

`<time.h>` library mein kai functions hote hain jo date aur time measure karte hain.

For example, `time()` function current time return karta hai as a value of type `time_t`.

Aap `ctime()` ka use karke is time ko ek readable string mein convert kar sakte ho, jaise `"Mon Jun 24 10:15:00 2025"`:

Example

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

int main() {
  time_t currentTime;
  time(&currentTime); // Get the current time

  printf("Current time: %s", ctime(&currentTime));
  return 0;
}
```

Breaking Down the Time

Agar aapko date ya time ke individual parts (jaise year, month, ya hour) access karne hain, toh aap `localtime()` function ka use kar sakte ho.

Ye function current time (jo `time()` se milta hai) ko ek `struct tm` mein convert karta hai, jo ek special structure hai jisme date aur time alag-alag fields mein store hota hai.

Yahan ek example hai jisme hum current date aur time ke har part ko print karte hain:

Example

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

int main() {
  time_t now = time(NULL);          // Get current time
  struct tm *t = localtime(&now);  // Convert to local time structure

  printf("Year: %d\n", t->tm_year + 1900);  // Add 1900 to get the actual year
  printf("Month: %d\n", t->tm_mon + 1);     // Months are numbered from 0 to 11, so add 1 to match real month numbers (1-12)
  printf("Day: %d\n", t->tm_mday);
  printf("Hour: %d\n", t->tm_hour);
  printf("Minute: %d\n", t->tm_min);
  printf("Second: %d\n", t->tm_sec);
  return 0;
}
```

Note:

`tm_year` years ko 1900 se count karta hai, isliye hum 1900 add karte hain taaki full year mile.
`tm_mon` 0 se start hota hai (0 = January, 11 = December).

Note:

Hum `->` use karte hain kyunki `localtime()` ek pointer return karta hai `struct tm` ke liye.

Jaise aapne **Structs & Pointers** chapter mein seekha tha:
Agar aapke paas ek pointer hai struct ka, toh `->` use karke members access karo.
Dot (`.`) tab use karo jab aap directly struct ke saath kaam kar rahe ho, pointer ke saath nahi.

Formatting Date and Time

Aap `strftime()` ka use karke date aur time ko ek string ke form mein format kar sakte ho:

Example

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

int main() {
  time_t now = time(NULL);
  struct tm *t = localtime(&now);
  char buffer[100];

  strftime(buffer, sizeof(buffer), "%d-%m-%Y %H:%M:%S", t);
  printf("Formatted time: %s\n", buffer);
  return 0;
}
```

When to use Date and Time

C mein date aur time tab use karo jab aap chahte ho:

- Current time ya date display karna
- Errors ya user actions jaise events ko log karna
- Files ya messages mein timestamps add karna
- Measure karna ki koi process kitna time leti hai
- Actions ko schedule ya delay karna
- Random numbers generate karne ke liye seed set karna (taaki har run mein alag result mile)

C Random Numbers

Random Numbers

C mein aap random numbers generate kar sakte ho `rand()` function ke saath, jo `<stdlib.h>` library mein milta hai.

By default, `rand()` har baar program run karne par same sequence of numbers deta hai.
Har run par alag results paane ke liye, aap `srand()` ka use kar sakte ho ek "starting point" (seed) set karne ke liye.

Abhi aapne date aur time ke baare mein seekha — toh aap current time ko seed ke roop mein use kar sakte ho, kyunki time hamesha change hota hai.
Iske liye aap program mein `<time.h>` include karte ho.

Basic Random Number

`rand()` function ek random integer return karta hai.

Example

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

int main() {
  int r = rand();
  printf("%d\n", r);
  return 0;
}
```

Note:

Agar aap is program ko kai baar run karoge, toh har baar same numbers milenge.
Ye isliye hota hai kyunki humne abhi tak seed set nahi kiya hai.

Seeding the Random Generator

Har baar different numbers paane ke liye, aapko `rand()` ko ek starting point (seed) dena hota hai.

Ye kaam aap `srand()` ke saath karte ho. Ek common trick hai current time ko seed banana, kyunki time hamesha change hota rehta hai:

Example

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main() {
  srand(time(NULL));  // seed with current time

  printf("%d\n", rand());
  printf("%d\n", rand());
  printf("%d\n", rand());
  return 0;  }
```

Tip:

`srand()` ko sirf ek baar call karo — program ke start mein (usually `main()` ke beginning mein).
Loop ke andar baar-baar call mat karo.

Random Number in a Range

Aksar aapko chhoti range mein numbers chahiye hote hain, jaise 0 se 9 tak.
Ye aap modulo operator `%` ka use karke kar sakte ho:

Example

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main() {
  srand(time(NULL));

  int x = rand() % 10;  // 0..9
  printf("%d\n", x);
  return 0;
}
```

Real-Life Example: Rolling Dice

Ek common real-life example hai dice roll karna (six-sided dice).
Isse simulate karne ke liye hum `rand() % 6` use karte hain — ye 0 se 5 tak numbers deta hai,
aur agar hum +1 kar dein toh range 1 se 6 ho jaati hai:

Example

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main() {
  srand(time(NULL));

  int dice1 = (rand() % 6) + 1;
  int dice2 = (rand() % 6) + 1;
```

```
  printf("You rolled %d and %d (total = %d)\n", dice1,
dice2, dice1 + dice2);

  return 0;
}
```

Har baar program run karne par aapko 1 se 6 ke beech ke do random numbers milenge —
bilkul waise jaise aap real life mein do dice roll karte ho.

C Preprocessor and Macros

Preprocessor and Macros

C mein, preprocessor actual compilation start hone se pehle run hota hai.
Ye kaam karta hai jaise files include karna aur macros define karna.

Preprocessor commands # symbol se start hote hain aur unhe **directives** kaha jaata hai.

#include - Include Header Files

Aap pehle bhi `#include` directive kai baar dekh chuke ho — ye compiler ko batata hai ki ek file include karni hai.

Ye libraries ya custom header files add karne ke liye use hota hai:

Example

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

Standard libraries ke liye **angle brackets < >** use karo,
aur apni khud ki files ke liye **double quotes " "** use karo.

Tip:
Most commonly used libraries aapko humare **C Reference Documentation** mein mil jaayengi.

#define - Create a Macro

Ek **macro** ek name hota hai jo ek value (jaise PI) ya code ke piece ko represent karta hai,
aur use `#define` directive se define kiya jaata hai.

Neeche ke example mein, `PI` ko 3.14 se replace kiya jaata hai before program compile hota hai.
Matlab, jahan bhi `PI` likha hoga, compiler usse 3.14 se replace kar dega:

Example

```
#define PI 3.14

int main() {
```

```
  printf("Value of PI: %.2f\n", PI);
  return 0;
}
```

Macros parameters bhi le sakte hain, bilkul functions ki tarah:

Example

```
#define SQUARE(x) ((x) * (x))

int main() {
  printf("Square of 4: %d\n", SQUARE(4));
  return 0;
}
```

Parameter wale macros shortcut ki tarah kaam karte hain,
lekin parentheses ka use zarur karo taaki galtiyaan na ho.

#ifdef and #ifndef - Conditional Compilation

`#ifdef` aur `#ifndef` directives allow karte hain ki aap apne code ke parts ko include ya skip karo,
depending on whether koi macro defined hai ya nahi.

Isse **conditional compilation** kaha jaata hai,
aur ye debugging ya program ke alag-alag versions banane ke liye useful hota hai.

Example

```
#define DEBUG

int main() {
  #ifdef DEBUG
    printf("Debug mode is ON\n");
  #endif
```

```
  return 0;
}
```

Agar `DEBUG` defined hai, toh message print hoga.
Agar defined nahi hai, toh wo part of code skip kar diya jaayega.

Create Your Own Header Files

Next chapter mein aap seekhenge kaise apni **header files** banani hain aur apne code ko multiple files mein organize karna hai using **"modular programming."**

C Organize Code

Organize Your Code - Modular Programming

C programming mein, **modular programming** ka matlab hota hai apne code ko chhote, reusable parts mein todna.

Isse aapka code **read**, **maintain**, aur **debug** karna easy ho jaata hai.

Sabse common way C programs ko organize karne ka hota hai **separate .c files** aur **.h header files** ka use karna.

Yeh approach chhote beginner programs ke liye zaruri nahi hoti,
lekin bade projects ke liye ya experienced programmers ke liye bahut useful hoti hai,
jo apna code clean aur well-structured rakhna chahte hain.

Why Use Header Files?

- Functions declare karne ke liye jo kisi aur file mein defined hoti hain
- Variables, constants, ya macros ko multiple files ke beech share karne ke liye
- Code ko logically organized modules mein divide karne ke liye

Tip: Header files usually function declarations, macros, aur struct definitions contain karti hain.

Example: Creating a Header File

Is example mein aap seekhoge kaise apni header file create karni hai
aur usse multiple files mein code organize karna hai.

Chalo ek simple **calculator module** banate hain jisme ek header file aur ek source file hogi.

---

1. calc.h

Sabse pehle ek header file banao jiska naam ho `calc.h`, jisme functions declare karoge:

The calc.h file:

```
#ifndef CALC_H
#define CALC_H

int add(int x, int y);
int subtract(int x, int y);

#endif
```

Example Explained

`#ifndef`, `#define`, aur `#endif` lines ko **include guard** kaha jaata hai.

Ye ensure karti hain ki file ek se zyada baar include na ho by mistake, kyunki aisa hone par compilation errors aa sakte hain.

Ye ek common aur recommended practice hai sabhi C header files mein.

Is file mein do functions ke declarations hain: `add()` aur `subtract()`.

Writing the Function Definitions

2. calc.c

Ab function definitions likho `calc.c` file ke andar:

The calc.c file:

```
#include "calc.h"
```

```c
int add(int x, int y) {
  return x + y;
}

int subtract(int x, int y) {
  return x - y;
}
```

Ye file un functions ko define karti hai jo `calc.h` mein declare kiye gaye the.

Using the Module in main.c

3. main.c

Ab main program likho `main.c` file mein, aur header file include karo taaki calculator functions use kar sako:

The main.c file:

```c
#include <stdio.h>
#include "calc.h"

int main() {
  printf("5 + 5 = %d\n", add(5, 5));
  printf("6 - 4 = %d\n", subtract(6, 4));
  return 0;
}
```

Result:

```
5 + 5 = 10
6 - 4 = 2
```

How to Compile Multiple Files

Jab aapka program multiple files mein split hota hai,
toh aapko unhe saath mein compile karna hota hai.

Example:

```
gcc main.c calc.c -o program
```

Ye compiler ko batata hai ki `main.c` aur `calc.c` ko use karke ek executable banaye jiska naam ho **program**.

C Storage Classes

C Storage Classes

Storage classes ye define karti hain ki variables ka **lifetime**, **visibility**, aur **memory location** kya hoga.

C mein 4 main storage class specifiers hote hain:

- `auto`
- `static`
- `register`
- `extern`

Difference Between Scope and Storage Classes

Scope batata hai ki ek variable **kahaan use** ho sakta hai,
jabki **storage class** batata hai ki wo variable **kitni der tak exist karega** aur **kahaan store** hoga.

Ye chapter C Scope chapter ka continuation hai.

---

auto

`auto` keyword local variables ke liye use hota hai.
Ye default storage class hoti hai un variables ke liye jo functions ke andar declare kiye jaate hain,
isliye ise explicitly likhna rarely zaruri hota hai.

Example

```c
int main() {
  auto int x = 50;  // Same as just: int x = 50;
  printf("%d\n", x);
  return 0;
}
```

---

static

`static` keyword ek variable ya function ke **lifetime** aur **visibility** ko change karta hai:

- **Static local variables** apni value ko function calls ke beech preserve karte hain.

- **Static global variables/functions** sirf usi file ke andar visible hote hain.

Example

```c
void count() {
  static int myNum = 0; // Keeps its value between calls
  myNum++;
  printf("num = %d\n", myNum);
}

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

Result:

```
num = 1
num = 2
num = 3
```

Tip:

Agar aap static keyword hata doge, to har baar variable dobara initialize ho jaayega,
aur output hamesha `num = 1` hi rahega.

---

register

`register` keyword suggest karta hai ki variable ko CPU register mein store kiya jaaye
(for faster access).

Aap register variable ka address `&` operator se **nahi le sakte**.

Note:

`register` keyword aaj ke time mein mostly obsolete hai — modern compilers khud hi best variables ko registers mein store kar lete hain.

Example

```c
int main() {
  register int counter = 0;
  printf("Counter: %d\n", counter);
  return 0;
}
```

---

extern

`extern` keyword compiler ko batata hai ki koi variable ya function **kisi aur file mein defined** hai.

Ye multiple source files ke saath kaam karte waqt commonly use hota hai.

File 1: main.c

```c
int main() {
```

```
    register int counter = 0;
    printf("Counter: %d\n", counter);
    return 0;
}
```

File 2: data.c

```
int shared = 50;  // Definition of the variable
```

Compile both files together:

```
gcc main.c data.c -o program
```

C Bitwise Operators

C Bitwise Operators

Note: Ye C ka thoda advanced topic hai. Agar aap programming mein naye ho, to pehle ye tricky lag sakta hai — bitwise operators mainly special cases mein use hote hain jaise system programming, hardware control, ya performance optimizations.

C mein bitwise operators aapko numbers ke **bits** (1s aur 0s) ke saath directly kaam karne dete hain.

Har integer computer mein binary mein store hota hai, matlab ye bits (binary digits) ke through represent hota hai jo ya to 0 ya 1 hote hain.

Bitwise operators aapko ye bits **compare**, **combine**, **shift**, ya **flip** karne dete hain.

Note: Bitwise operations sirf integer types (jaise int, char, ya long) pe kaam karte hain.

Tip: Agar aap binary numbers se abhi tak familiar nahi hain, to pehle ye pages dekhein:

- Intro to Programming - Binary Numbers
- Intro to Programming - Bits and Bytes

List of Bitwise Operators

Operator	Name	Description
&	AND	Sets each bit to 1 if both bits are 1
|	OR	Sets each bit to 1 if at least one bit is 1
^	XOR	Sets each bit to 1 if only one bit is 1
~	NOT	Inverts all the bits
<<	Left Shift	Shifts bits to the left (multiplies by powers of 2)
>>	Right Shift	Shifts bits to the right (divides by powers of 2)

Binary Example

Do integers se shuru karte hain:

```
int a = 6;  // 0110 in binary
int b = 3;  // 0011 in binary
```

Alag bitwise operators ka result:

Operation	Binary Result	Decimal Result
a & b	0010	2
a | b	0111	7
a ^ b	0101	5
~a	...1001	-7 (most systems)
a << 1	1100	12
a >> 1	0011	3

Decimal and Binary Values

Decimal	Binary (16-bit)
0	0000000000000000
1	0000000000000001
2	0000000000000010
3	0000000000000011
4	0000000000000100
5	0000000000000101
6	0000000000000110
7	0000000000000111
8	0000000000001000
9	0000000000001001

Decimal	Binary (16-bit)
10	0000000000001010
11	0000000000001011
12	0000000000001100

Tip: Har step left shift se value double ho jaati hai.
Jaise 0000000000000100 = 4, kyunki right se third bit set hai ($2^2 = 4$).

Bitwise AND (&)

& operator har bit ko compare karta hai aur 1 return karta hai **sirf tab** jab dono bits 1 ho.

```
int a = 6;    // 0110
int b = 3;    // 0011

int result = a & b;
printf("Result: %d\n", result); // 2 (0010)
```

Bitwise OR (|)

| operator bit ko 1 set karta hai agar **koi bhi ek bit 1** ho.

```
int a = 6;    // 0110
int b = 3;    // 0011

int result = a | b;
printf("Result: %d\n", result); // 7 (0111)
```

Bitwise XOR (^)

^ operator 1 return karta hai jab bits **different** ho.

```
int a = 6;    // 0110
int b = 3;    // 0011

int result = a ^ b;
printf("Result: %d\n", result); // 5 (0101)
```

Bitwise NOT (~)

~ operator har bit ko invert karta hai (0 → 1, 1 → 0).

```
int a = 5; // 00000101

int result = ~a;
printf("Result: %d\n", result); // -6 (most systems)
```

Note: ~ ka result negative numbers ke storage (usually two's complement) pe depend karta hai.
Jaise 5 (00000101) → 11111010 → interpreted as -6.

Left Shift (<<)

<< operator bits ko left shift karta hai aur right mein 0 fill karta hai.
Ye **powers of 2 se multiply** karne ke barabar hai.

```
int a = 3; // 00000011

int result = a << 2;
printf("Result: %d\n", result); // 12 (3 * 2^2)
```

Right Shift (>>)

operator bits ko right shift karta hai.
Ye **powers of 2 se divide** karne ke barabar hai.
Signed integers mein sign bit preserved ho sakti hai system pe depend karta hai.

```
int a = 12; // 00001100

int result = a >> 2;
printf("Result: %d\n", result); // 3 (12 / 2^2)
```

Real-Life Example: Flags and Permissions

Bitwise operators aksar multiple options ek integer mein store karne ke liye use hote hain.

```
#define READ  1  // 0001
#define WRITE 2  // 0010
#define EXEC  4  // 0100

int permissions = READ | WRITE;  // user can read and write

if (permissions & READ) {
  printf("Read allowed\n");
}
if (permissions & WRITE) {
  printf("Write allowed\n");
}
if (permissions & EXEC) {
  printf("Execute allowed\n");
}
```

Is example mein, user ke paas **READ** aur **WRITE** permission hai, lekin **EXEC** nahi hai.

Summary

- Bitwise operators integers ke **individual bits** pe kaam karte hain
- `&`: AND - dono bits 1 honi chahiye
- `|`: OR - koi bhi bit 1 ho sakti hai
- `^`: XOR - sirf ek bit 1 ho
- `~`: NOT - saare bits flip kar deta hai
- `<<`: Left shift - powers of 2 se multiply
- `>>`: Right shift - powers of 2 se divide

C Fixed Width Integers
Fixed-Width Integers
C mein, **int**, **short**, aur **long** types ka size computer aur system ke according alag ho sakta hai.
For example, ek system pe **int** 2 bytes le sakta hai, aur doosre system pe 4 bytes.

Is problem ko solve karne ke liye, C mein `<stdint.h>` header diya gaya hai jisme **fixed-width integer types** hote hain — ye har computer pe **same size (number of bits)** ke hote hain.

The most common fixed-width types are:

Type	Size	Range	Printf Specifier
int8_t	8 bits (1 byte)	-128 to 127	%d
uint8_t	8 bits (1 byte)	0 to 255	%u
int16_t	16 bits (2 bytes)	-32,768 to 32,767	%d
uint16_t	16 bits (2 bytes)	0 to 65,535	%u
int32_t	32 bits (4 bytes)	-2,147,483,648 to 2,147,483,647	%d
uint32_t	32 bits (4 bytes)	0 to 4,294,967,295	%u
int64_t	64 bits (8 bytes)	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	%lld
uint64_t	64 bits (8 bytes)	0 to 18,446,744,073,709,551,615	%llu

---

Note:
Letter **u** ka matlab hai *unsigned*, yani ye type sirf non-negative values (0 se upar) store kar sakta hai.
Iska matlab hai ki aapko double maximum positive value milti hai compared to signed version, lekin negative numbers store nahi kar sakte.

Using Fixed-Width Integers

Example:

```c
#include <stdio.h>
#include <stdint.h>  // needed for fixed-width integers

int main() {
  int8_t a = 100;         // 8-bit integer
  int16_t b = 30000;      // 16-bit integer
  int32_t c = 2000000;    // 32-bit integer
  int64_t d = 9000000000; // 64-bit integer

  printf("%d\n", a);
  printf("%d\n", b);
  printf("%d\n", c);
  printf("%lld\n", d); // use %lld for 64-bit
  return 0;
}
```

When to Use Fixed-Width Integers?

Normally, simple programs ke liye aapko in types ki zarurat nahi hoti — ek normal **int** usually kaafi hota hai.
Lekin fixed-width integers zaruri hote hain jab:

- Aap **embedded systems** likh rahe ho (jaise microcontrollers ya small devices).
- Aap **file formats** ke saath kaam kar rahe ho jahan exact size matter karta hai.
- Aap **network data transfer** kar rahe ho aur chahte ho ki results har machine pe same hoin.

Real Life Example

Socho aap ek program bana rahe ho jo device ka **battery level** dikhata hai. Battery percentage hamesha 0 se 100 ke beech hota hai, to yahan **int** jaisa large type use karne ki zarurat nahi.
Instead, aap **uint8_t** use kar sakte ho — ye exactly **1 byte (8 bits)** ka hota hai aur 0 se 255 tak values store kar sakta hai.

Example:

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

int main() {
  uint8_t battery = 87; // battery level percentage

  printf("Battery level is %u out of 100\n", battery);

  return 0;
}
```

Yahan humne memory bachayi hai **uint8_t** use karke, jo perfect hai chhoti aur well-defined range (jaise battery percentage) ke liye.

C Projects

Projects and Practical Applications

Seekho kaise apni C knowledge ko real-world projects mein apply karein.

Is section mein hum mini applications banayenge using wo features jo aapne poore tutorial mein seekhe hain.

Why Build Projects?

Projects C seekhne ka ek important part hain. Chhote se start karo aur dheere-dheere aur features add karte jao:

- Samjho real programs kaise structured hote hain
- Practice karo kaise concepts (jaise functions, loops, file handling) combine kiye jaate hain
- Apni debugging aur problem-solving skills improve karo
- Job interviews aur coding exercises ke liye prepare ho jao

Tip: Jitna zyada build karoge, utna achha samjhoge.

Project Examples

Aap simple input/output wale chhote projects se start kar sakte ho. Jaise ek program likho jo:

- Aapse aapka naam pooche
- Aapse aapki age pooche
- Print kare: `Hi <name>! You will turn <age+1> next year.`

Jab aap comfortable ho jao, thode bade projects try karo jisme loops, conditions, aur arrays use hote hain:

- Ek small shopping list program banao (items store karo aur print karo)
- Guess a Number Game
- Calculate a Student's Average

Aur jab aapke skills aur improve ho jaayein, tab aur advanced projects banao jisme functions, structures aur file handling ho:

- Simple Calculator
- Address Book
- To-Do List
- Quiz Game

Project: Calculate a Student's Average

Chalo ek program banate hain jo student ke multiple grades ka average calculate kare.

Program user se 1 se 5 tak grades input lega, phir average calculate karega aur letter grade (A se F tak) display karega.

Example:

```c
// This function returns a letter grade based on the average
char gradeFunction(double avg) {
  if (avg >= 90) return 'A';
  else if (avg >= 80) return 'B';
  else if (avg >= 70) return 'C';
  else if (avg >= 60) return 'D';
  else return 'F';
}

int main(void) {
  int count;
  double sum = 0, grade;

  // Ask the user to enter total grades between 1 to 5
  printf("How many grades (1 to 5)? ");
```

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

  // Validate that count is between 1 and 5
  if (count < 1 || count > 5) {
    printf("Invalid number. You must enter between 1 and 5
grades.\n");
    return 1;  // Exit
  }

  // Loop to collect each grade
  for (int i = 1; i <= count; i++) {
    printf("Enter grade %d: ", i);
    scanf("%lf", &grade);
    sum += grade;
  }

  // Calculate the average score
  double avg = sum / count;

  // Display numeric average
  printf("Average: %.2f\n", avg);

  // Display letter grade
  printf("Letter grade: %c\n", gradeFunction(avg));

  return 0;
}
```

Example Output:

```
How many grades (1 to 5)? 3
Enter grade 1: 85
Enter grade 2: 91
Enter grade 3: 78
Average: 84.67
Letter grade: B
```

Key Concepts Used: loops, functions, conditions, input handling, and basic logic.

Practice Challenge

Apna khud ka chhota project banao. For example, ek program likho jo:

- User se pooche ki wo 5 tak items kharidna chahta hai
- Items ko ek array mein store kare
- Puri shopping list print kare
- Count kare ki kitne items enter kiye gaye

Extra Challenge:
Ek feature add karo jisse user kisi item ko search kar sake aur program bataye ki wo list mein hai ya nahi.

Tip: CodeBlocks ya kisi bhi C IDE mein ye programs likho aur khud experiment karo!
Chhote se start karo. Ek feature ek baar mein add karo. Har step pe test karte raho.

Tip: Humne *Real Life Examples* page mein simple projects ka collection bhi add kiya hai.

C Reference Documentation

C Reference

Keywords

C language ke reserved words hote hain jo specific kaam ke liye use kiye jaate hain (jaise `int`, `if`, `return`, `for`, etc.).
Inhe aap variable names ke liye use nahi kar sakte.

Popular Libraries

<stdio.h>

(Standard Input/Output Header)
Isme input/output functions hote hain jaise:

- `printf()` – output print karne ke liye
- `scanf()` – input lene ke liye
- `fopen()`, `fclose()` – files ke saath kaam karne ke liye

<stdlib.h>

(Standard Library Header)
Memory management aur conversions ke liye functions deta hai:

- `malloc()`, `free()` – memory allocation aur release ke liye
- `atoi()`, `atof()` – strings ko numbers mein convert karne ke liye
- `rand()`, `srand()` – random number generation ke liye

<string.h>

(String Handling Header)
Strings (text) ke saath kaam karne ke liye functions:

- `strlen()` – string ki length nikalne ke liye
- `strcpy()`, `strcat()` – strings copy ya combine karne ke liye
- `strcmp()` – strings compare karne ke liye

<math.h>

(Mathematics Header)
Mathematical calculations ke liye functions:

- `sqrt()` – square root nikalne ke liye
- `pow()` – power calculate karne ke liye
- `sin()`, `cos()`, `tan()` – trigonometric functions ke liye

<ctype.h>

(Character Type Header)
Characters ke type check karne ya convert karne ke liye:

- `isalpha()` – check karta hai kya character alphabet hai
- `isdigit()` – check karta hai kya digit hai
- `tolower()`, `toupper()` – case convert karne ke liye

<time.h>

(Time Header)
Time aur date ke saath kaam karne ke liye:

- `time()` – current time nikalne ke liye
- `ctime()` – time ko readable string mein convert karne ke liye
- `difftime()` – do times ke beech difference nikalne ke liye

Ye sab headers aur keywords milkar C ko ek powerful aur flexible language banate hain.
Inhe achhe se samajhna aapko professional-level coding mein help karega.

C Keywords

C Keywords

Keyword	Description
break	Loop ya switch block se bahar nikalne ke liye use hota hai
case	switch statements mein code block ko mark karta hai
char	Ek data type jo single character store kar sakta hai
const	Variable ya parameter ko constant (unchangeable) define karta hai
continue	Loop ke next iteration par jump karta hai
default	switch statement mein default code block specify karta hai

Keyword	Description
do	while ke saath milkar do/while loop banata hai
double	Ek data type jo usually 64 bits ka hota hai aur fractional numbers store karta hai
else	Conditional statements mein use hota hai
enum	Ek enumerated type declare karta hai
float	Ek data type jo usually 32 bits ka hota hai aur fractional numbers store karta hai
for	Ek for loop create karta hai
goto	Ek label ke through code ke kisi specific line par jump karta hai
if	Conditional statement banata hai
int	Ek data type jo usually 32 bits ka hota hai aur whole numbers store karta hai
long	Ensure karta hai ki integer kam se kam 32 bits ka ho (64 bits ke liye long long use karein)
return	Function se value return karne ke liye use hota hai
short	Integer ka size 16 bits tak reduce karta hai
signed	Specify karta hai ki int ya char positive aur negative dono values represent kar sakta hai (int ke liye default hota hai, isliye keyword zaruri nahi hota)
sizeof	Ek operator jo variable ya data type ke memory size ko return karta hai
static	Specify karta hai ki function ke andar ka variable function ke khatam hone ke baad bhi apni value rakhe
struct	Ek structure define karta hai
switch	Multiple code blocks me se ek ko execute karta hai
typedef	Ek custom data type define karta hai
unsigned	Specify karta hai ki int ya char sirf positive values represent kare (isse number ka maximum range double ho jata hai)

Keyword	Description
void	Function ke liye use hota hai jo koi value return nahi karta, ya ek unspecified type pointer specify karta hai
while	Ek while loop create karta hai

C stdio (stdio.h) Library

C stdio Functions

Function	Description (Hinglish me)
fclose()	File ko close karta hai.
feof()	Jab file ka end aa jata hai, to true return karta hai.
ferror()	Agar file operation me koi error hua ho, to true return karta hai.
fgetc()	File se ek character padhta hai aur pointer ko aage badha deta hai.
fgets()	File se ek line padhta hai aur pointer ko aage move karta hai.
fopen()	File ko open karta hai aur ek file pointer return karta hai.
fprintf()	File me formatted string likhta hai (printf jaisa, lekin file ke liye).
fputc()	File me ek character likhta hai aur pointer ko aage badhata hai.
fputs()	File me ek string likhta hai aur pointer ko aage move karta hai.
fread()	File se data read karta hai aur memory block me store karta hai.
fscanf()	File se formatted data read karta hai aur variables me store karta hai.
fseek()	File pointer ko manually kisi position pe move karta hai.
ftell()	File pointer ki current position return karta hai.
fwrite()	Memory block se data file me likhta hai.

Function	Description (Hinglish me)
getc()	fgetc() jaisa hi hai — ek character read karta hai.
getchar()	User se ek character input leta hai aur uska ASCII value return karta hai.
printf()	Console (screen) pe formatted string print karta hai.
putc()	fputc() jaisa hi hai — ek character file me likhta hai.
putchar()	Console pe ek character print karta hai.
puts()	Console pe ek string print karta hai.
remove()	File ko delete karta hai.
rename()	File ka naam change karta hai.
rewind()	File pointer ko wapas file ke starting me le jata hai.
scanf()	User se formatted input leta hai aur variables me store karta hai.
snprintf()	Ek formatted string ko char array me likhta hai (safe version).
sprintf()	Ek formatted string ko char array me likhta hai.
sscanf()	Ek string se formatted data read karta hai aur variables me store karta hai.

C stdlib (stdlib.h) Library

C stdlib Functions

Function	Description (Hinglish me)
abs()	Kisi number ka absolute (positive) value return karta hai. Example: `abs(-5)` return karega `5`.
atof()	String (like `"3.14"`) ko `double` number me convert karta hai.
atoi()	String (like `"25"`) ko `int` me convert karta hai.

Function	Description (Hinglish me)
atol()	String ko `long int` me convert karta hai.
atoll()	String ko `long long int` me convert karta hai.
calloc()	Dynamic memory allocate karta hai aur usko zero se initialize karta hai.
div()	Do integers ka quotient aur remainder ek saath return karta hai.
exit()	Program ko immediately end (terminate) karta hai.
free()	Pehle se allocated memory ko release karta hai (memory leak se bachne ke liye).
malloc()	Dynamic memory allocate karta hai (lekin initialize nahi karta).
qsort()	Array ke elements ko sort karta hai (quick sort algorithm use karta hai).
rand()	Random integer generate karta hai.
realloc()	Pehle se allocated memory ka size change (resize) karta hai.
srand()	Random number generator ko initialize karta hai (seed set karta hai).

C string (string.h) Library

C string Functions

Function	Description (Hinglish me)
memchr()	Memory block me kisi specific value (character ya byte) ka **pehla occurrence** dhundhta hai aur uska pointer return karta hai.
memcmp()	Do memory blocks ko compare karta hai aur batata hai kaunsa bada hai (numeric value ke hisaab se).
memcpy()	Ek memory block ka data doosre memory block me **copy** karta hai.

Function	Description (Hinglish me)
memmove()	`memcpy()` jaisa hi hai, lekin ye overlapping memory blocks ke case me safely copy karta hai.
memset()	Memory block ke har byte ko ek hi value se fill karta hai. (Example: sabko zero se set karna)
strcat()	Do strings ko jodta hai — doosri string ko pehli string ke end me add karta hai.
strchr()	String me kisi character ka **pehla occurrence** dhundhta hai aur uska pointer return karta hai.
strcmp()	Do strings ke ASCII values compare karta hai aur batata hai kaunsa bada hai.
strcoll()	Do strings ko compare karta hai lekin **locale-based (language-specific)** rules follow karta hai.
strcpy()	Ek string ko doosri string me **copy** karta hai.
strcspn()	String ki length return karta hai jab tak koi specified character nahi milta.
strerror()	Error code ka meaning batane wala **error message string** return karta hai.
strlen()	String ki length (characters ki count) return karta hai.
strncat()	Do strings ko jodta hai, lekin **limited characters** tak (specified number).
strncmp()	Do strings ke **n characters tak** compare karta hai.
strncpy()	Ek string ke **n characters tak** doosri string me copy karta hai.
strpbrk()	String me pehli jagah dhundhta hai jahan koi specified character milta hai.
strrchr()	String me kisi character ka **last occurrence** dhundhta hai.
strspn()	String ki length return karta hai jab tak usme sirf specified characters hi milte hain.
strstr()	Ek string ke andar doosri string ka **pehla occurrence** dhundhta hai.

Function	Description (Hinglish me)
strtok()	Ek string ko **parts (tokens)** me todta hai using delimiters (like space, comma, etc.).
strxfrm()	String ke characters ko **locale-specific encoding** me convert karta hai (comparison ke liye useful).

C math (math.h) Library

C Math Functions

Function	Description (Hinglish me)
acos(x)	x ka arccosine (inverse cosine) return karta hai, radians me.
acosh(x)	x ka hyperbolic arccosine return karta hai.
asin(x)	x ka arcsine (inverse sine) return karta hai, radians me.
asinh(x)	x ka hyperbolic arcsine return karta hai.
atan(x)	x ka arctangent (inverse tangent) return karta hai, -π/2 se π/2 ke beech.
atan2(y, x)	(x, y) coordinates se polar angle (theta) calculate karta hai.
atanh(x)	x ka hyperbolic arctangent return karta hai.
cbrt(x)	x ka cube root return karta hai.
ceil(x)	x ko upar ke nearest integer tak round karta hai.
copysign(x, y)	x ki value return karta hai lekin sign y ka hota hai.
cos(x)	x (radians me) ka cosine value return karta hai.

Function	Description (Hinglish me)
cosh(x)	`x` ka hyperbolic cosine return karta hai.
exp(x)	`e` (Euler's number) ki power `x` return karta hai — i.e., `e^x`.
exp2(x)	`2` ki power `x` return karta hai — i.e., `2^x`.
expm1(x)	`e^x - 1` return karta hai (precision ke liye useful).
erf(x)	`x` ka error function value return karta hai.
erfc(x)	Complementary error function value return karta hai.
fabs(x)	`x` ka absolute (positive) value return karta hai.
fdim(x, y)	Agar `x > y`, to `x - y` return karta hai, warna `0`.
floor(x)	`x` ko neeche ke nearest integer tak round karta hai.
fma(x, y, z)	`(x * y) + z` calculate karta hai bina precision lose kiye.
fmax(x, y)	`x` aur `y` me se bada value return karta hai.
fmin(x, y)	`x` aur `y` me se chhota value return karta hai.
fmod(x, y)	`x / y` ka floating-point remainder return karta hai.
frexp(x, y)	`x` ko `m * 2^n` ke form me todta hai — `m` return karta hai aur `n` ko `y` me store karta hai.
hypot(x, y)	$\sqrt{(x^2 + y^2)}$ return karta hai (overflow/underflow se safe).
ilogb(x)	`x` ka logarithm base ke integer part ko return karta hai.
ldexp(x, y)	`x * 2^y` return karta hai.
lgamma(x)	Gamma function ke absolute value ka logarithm return karta hai.
llrint(x)	`x` ko nearest integer tak round karta hai aur long long int me return karta hai.
llround(x)	`x` ko round karke long long int return karta hai.
log(x)	`x` ka natural logarithm (base e) return karta hai.
log10(x)	`x` ka base 10 logarithm return karta hai.
log1p(x)	`log(x + 1)` return karta hai (precision ke liye).

Function	Description (Hinglish me)
log2(x)	x ka base 2 logarithm return karta hai.
logb(x)	x ka floating-point base logarithm return karta hai.
lrint(x)	x ko nearest integer tak round karta hai aur long int return karta hai.
lround(x)	x ko nearest integer tak round karta hai aur long int return karta hai.
modf(x, y)	x ka fractional part return karta hai aur integer part y me store karta hai.
nan(s)	"Not a Number" (NaN) value return karta hai.
nearbyint(x)	x ko nearest integer tak round karta hai (without changing floating point mode).
nextafter(x, y)	x ke baad wala nearest floating point number return karta hai y ki direction me.
nexttoward(x, y)	nextafter() jaisa hi, lekin y long double ho sakta hai.
pow(x, y)	x^y (x raised to the power y) return karta hai.
remainder(x, y)	x / y ka remainder return karta hai (nearest integer tak round karke).
remquo(x, y, z)	x / y ka remainder aur quotient dono calculate karta hai; quotient z me store karta hai.
rint(x)	x ko nearby integer tak round karta hai (current rounding mode ke hisaab se).
round(x)	x ko nearest integer tak round karta hai.
scalbln(x, y)	x * R^y (usually R = 2) return karta hai.
scalbn(x, y)	x * R^y (usually R = 2) return karta hai.
sin(x)	x ka sine value (radians me) return karta hai.
sinh(x)	x ka hyperbolic sine return karta hai.

Function	Description (Hinglish me)
sqrt(x)	`x` ka square root return karta hai.
tan(x)	`x` ka tangent value (radians me) return karta hai.
tanh(x)	`x` ka hyperbolic tangent return karta hai.
tgamma(x)	`x` ka gamma function value return karta hai (factorial ka extension).
trunc(x)	`x` ka integer part return karta hai (fractional part hata deta hai).

C ctype (ctype.h) Library

C ctype Functions

Function	Description (Hinglish me)
isalnum()	Check karta hai ki character **letter ya digit** hai (A–Z, a–z, 0–9).
isalpha()	Check karta hai ki character **sirf letter** hai (A–Z ya a–z).
isblank()	Check karta hai ki character **space ya tab** hai.
iscntrl()	Check karta hai ki character **control character** hai (like newline `\n`, tab `\t`, etc.).
isdigit()	Check karta hai ki character **digit (0–9)** hai.
isgraph()	Check karta hai ki character ka **visible graphical representation** hai (space ke alawa sab printable characters).
islower()	Check karta hai ki character **lowercase letter** hai (a–z).
isprint()	Check karta hai ki character **printable** hai (including space).
ispunct()	Check karta hai ki character **punctuation mark** hai (like !, ?, ., ,, etc.).
isspace()	Check karta hai ki character **whitespace** hai (space, tab, newline, etc.).

Function	Description (Hinglish me)
isupper()	Check karta hai ki character **uppercase letter** hai (A–Z).
isxdigit()	Check karta hai ki character **hexadecimal digit** hai (0–9, A–F, a–f).
tolower()	Ek **uppercase** character ko **lowercase** me convert karta hai.
toupper()	Ek **lowercase** character ko **uppercase** me convert karta hai.

C time (time.h) Library

C time Functions

Function	Description (Hinglish me)
time()	Current calendar time return karta hai as `time_t` value (seconds since 1 Jan 1970).
localtime()	`time_t` value ko **local time** me convert karta hai aur `struct tm` ka pointer return karta hai.
gmtime()	`time_t` value ko **UTC (GMT) time** me convert karta hai, `struct tm` ke form me.
ctime()	`time_t` value ko readable string me convert karta hai (example: Thu Jun 26 10:30:00 2025).
asctime()	`struct tm` ko standard date/time string me convert karta hai.
strftime()	`struct tm` ko **custom format** me date/time string me convert karta hai.
difftime()	Do `time_t` values ke beech ka difference (seconds me) calculate karta hai.
mktime()	Fully filled `struct tm` ko `time_t` value me convert karta hai.

Function	Description (Hinglish me)
clock()	Program ke processor clock ticks return karta hai (execution time measure karne ke liye use hota hai).

---

struct tm kya hai?

`struct tm` ek special structure hai jo **date aur time ke individual parts** hold karta hai, jaise:

- `tm_year` → Year (1900 ke baad ka difference)
- `tm_mon` → Month (0–11)
- `tm_mday` → Day of the month (1–31)
- `tm_hour` → Hour (0–23)
- `tm_min` → Minutes (0–59)
- `tm_sec` → Seconds (0–60, leap second included)
- `tm_wday` → Weekday (0 = Sunday)
- `tm_yday` → Day of the year (0–365)
- `tm_isdst` → Daylight saving flag

Is structure ko use karke aap **date aur time ke har component ko individually access aur format kar sakte ho**.
