Constants

Constant A Constant is a value that cannot be altered during the execution of a program.

Defining Constant using preprocessor directive Constants are created using the #define preprocessor directive.

#define PI 3.14159 // Define constant

Defining Constant using const keyword const keyword is used to define constants with a specific type.

const int maxUsers = 100; // const keyword constant

Variables

Definition Variables are named storage locations in memory for data values.

int age; // Declares an integer variable named age.

Initialization Initialization assigns an initial value to a variable.

int age = 25; // Initializes age with value 25.

Data Types Data type of a variable determines the size and type of data it holds.

float salary; // Declares a variable of type float.

Scope Scope defines where a variable can be accessed within the code.

int main() { int local; } // local has scope within main.

Lifetime Lifetime refers to the duration a variable exists in memory.

static int count;// count has a lifetime of the program.

Types of C Variables

Global Variables Global variables are accessible from any function within the program.

int globalVar; // Declares a global variable.

Local Variables Local variables are accessible only within the function they are defined.

void func() { int localVar; } // localVar is local to func.

Automatic Variables Automatic variables are created when their block is entered and destroyed on exit.

void func() { auto int autoVar; } // autoVar is automatic.

External Variables External variables are defined outside any function and are accessible globally.

extern int extVar; // Declares an external variable.

Constant Variables Constant variables are variables whose value cannot be changed.

const int MAX = 100; // MAX is a constant variable.

Static Variables Static variables retain their value between function calls.

void function() { static int static_var = 0; } // Static variable

Register Variables Register variables suggest to the compiler to store them in a CPU register.

void function() { register int reg_var = 0; } // Register variable

Volatile Variables Volatile variables tell the compiler the variable can change unexpectedly.

volatile int volatile_var; // Volatile variable

Comments in a C Program

Comments are used to explain code and are ignored by the compiler.

Single-line Comments Single-line comments start with // and end at the line's end.

// This is a single-line comment

Multi-line Comments Multi-line comments start with /* and end with */.

/* This is a

   multi-line comment */

Nested Comments In C, nested comments are not allowed and will cause a compile error.

/* Outer comment /* Nested comment */ End */

printf()

printf() is a standard library function to output text to the console.

printf("Hello, World!\n"); // Prints 'Hello, World!' with a newline.

Format Specifiers Format specifiers define the type of data to be printed.

printf("%d", 10); // Prints an integer value: 10.

Multiple Arguments Handles multiple arguments according to the format specifiers.

printf("%s scored %d points.", "Alice", 90); // Prints 'Alice scored 90 points.'

Escape Sequences Escape sequences allow for formatting control, like newlines or tabs.

printf("Line1\nLine2"); // Prints 'Line1' followed by 'Line2' on a new line.

Width Specifiers Width specifiers set minimum field width for the output.

printf("%5d", 123); // Prints ' 123', with spaces for padding.

Precision Specifiers Precision specifiers set the number of digits after the decimal point.

printf("%.2f", 3.14159); // Prints '3.14'.

scanf()

scanf() is a function that reads formatted input from stdin.

scanf("%d", &number); // Reads an integer from the user.

It's used to take user input and store it in variables.

scanf("%f", &floatVar); // Stores a floating-point number.

Reading multiple values scanf can read multiple values in a single call.

scanf("%d %f", &intVar, &floatVar); // Reads an int and float.

Format specifiers Format specifiers define the type of data to be read.

scanf("%s", str); // Reads a string into variable str.

Skipping characters Use a space in format string to skip whitespace characters.

scanf(" %c", &charVar); // Skips leading whitespace.

Datatypes

Integer An Integer is a whole number without a fractional component,typically 16 bits in size.

int a = 100; // Declares an integer variable.

Short Short is a smaller integer type, typically 16 bits in size.

short b = 1000; // Declares a short integer variable.

Long Long is a larger integer type, typically 32 or 64 bits.

long c = 100000L; // Declares a long integer variable.

Signed Integers Signed integers can represent both positive and negative values. The range of signed integers is from $-2^{(n-1)}$ to $2^{(n-1)}-1$. n represents the number of bits used to store the signed integer.

int a = -10; // Signed integer with a negative value.

Unsigned Integers Unsigned integers can only represent non-negative values, including zero. The range of unsigned integers is from 0 to 2^n-1. n represents the number of bits used to store the Unsigned integer.

unsigned int b = 10; // Unsigned integer with a positive value.

char A char type stores a single character and uses 1 byte of memory.

char letter = 'A'; // Declares a char variable.

Float A float represents a single-precision floating point number.

float pi = 3.14f; // Single-precision floating point

Double A double represents a double-precision floating point number.

double e = 2.718281828459045; // Double-precision floating point

Type Conversions

Implicit Conversion Implicit conversion happens automatically when a value is assigned to a compatible type.

float num = 10; // Implicitly converts int to float.

Explicit Conversion Explicit conversion (casting) is done manually by the programmer.

int total = (int)3.14; // Explicitly casts float to int.

Promotion Promotion is an automatic conversion to a larger or more precise type.

double sum = 10 + 3.14; // int 10 promoted to double.

Demotion Demotion involves converting to a smaller or less precise type.

float balance = (float)3.14159; // Demotes double to float.

Integer to Floating Point Conversion from integer types to floating-point types.

int i = 42; float f = i; // Converts int to float.

Floating Point to Integer Conversion from floating-point types to integer types.

double d = 3.14; int i = (int)d; // Converts double to int.

Signed to Unsigned Conversion from signed to unsigned types.

signed int si = -10; unsigned int ui = si; // Converts signed to unsigned.

Unsigned to Signed Conversion from unsigned to signed types can cause unexpected behavior.

unsigned int ui = 10; int si = ui; // Converts unsigned to signed.

Operator Precedence

Parentheses Parentheses determine the primary grouping in an expression.

result = (a + b) * c; // (a + b) evaluated first

Multiplicative Operators Multiplication, division, and modulus operators are evaluated next.

result = a * (b / c); // Multiplication and division

Additive Operators Addition and subtraction operators are evaluated after multiplicative operators.

result = a + b - c; // Addition and subtraction

Relational Operators Relational operators compare values, evaluated after additive operators.

isGreater = a > b; // Greater than comparison

Equality Operators Equality operators check for equality and inequality.

isEqual = (a == b); // Equality check

Logical AND Logical AND is evaluated before logical OR.

result = (a > b) && (c > d); // Logical AND

Logical OR Logical OR is evaluated after logical AND.

result = (a > b) || (c > d); // Logical OR

Assignment Operators Assignment operators are evaluated last in an expression.

a += b; // Addition assignment

Associativity of Operators

Left-to-Right Associativity Operators with left-to-right associativity are evaluated from left to right.

int result = 100 / 10 * 2; // Evaluated as (100 / 10) * 2.

Right-to-Left Associativity Operators with right-to-left associativity are evaluated from right to left.

int x = 10; int y = 20; int z = x = y; // x = (x = y).

Unary Operators Unary operators like ++, --, and ! have right-to-left associativity.

int x = 10; int y = ++x; // First increment x, then assign to y.

Equality Operators Equality operators (==, !=) have left-to-right associativity.

int same = (5 == 5) == 1; // Evaluated as (5 == 5) == 1.

Assignment Operators Assignment operators (=, +=, -=, etc.) have right-to-left associativity.

int x, y; x = y = 4; // y is assigned 4, then x is assigned y.

Logical AND and OR Logical AND (&&) and OR (||) have left-to-right associativity.

int result = 1 || 0 && 0; // Evaluated as (1 || 0) && 0.

If Statement

The if statement is used to execute a block of code if a condition is true.

Syntax The syntax of an if statement includes the if keyword, condition, and code block.

```
if (condition) {

// code to be executed

}
```

Condition The condition is a boolean expression that evaluates to true or false.

```
if (x == 10) { // condition is true }
```

Code Block Execution If the condition is true, the code block within the if statement is executed.

```
if (temperature > 30) {

printf("It's hot outside!");

}
```

Single-line if Statement A single-line if statement doesn't require braces for a single statement.

```
if (score >= 50) printf("You passed.");
```

Nested if Statements Nested if statements allow for multiple levels of condition checking.

```
if (score > 50) {

if (score > 75) {

printf("Great score!");

 }
```

}

If-else Statement

The if-else statement is used to perform conditional execution of code

Syntax of if-else The if-else statement executes one block of code if a condition is true, another if false.

if (condition) {

// code if true

} else {

// code if false

}

Nested if-else Nested if-else statements allow for multiple conditions to be evaluated in a hierarchy.

if (condition1) {

// code if condition1 is true

} else if (condition2) {

// code if condition2 is true

} else {

// code if both are false

}

if-else Ladder An if-else ladder is a series of if-else statements to check multiple conditions.

if (condition1) {

// code for condition1

} else if (condition2) {

// code for condition2

} else {

// code if none are true

}

Ternary Operator The ternary operator is a shorthand for if-else, returning one of two values.

int max = (a > b) ? a : b; // max is the greater of a or b

The else if Clause

else if The else if clause is used to chain multiple conditions in an if statement.

if (x > 0) {

```
 // code
} else if (x < 0) {
 // code
}
```

Loops

While Loop A while loop repeatedly executes a target statement as long as a given condition is true.

```
int i = 0;
while (i < 5) {
 printf("%d\n", i);
 i++;
}
//prints 0,1,2,3,4 in separate lines
```

Do-While Loop A Do-While Loop executes at least once, then checks the condition.

```
int i = 0; do { printf("%d\n", i); i++; } while(i < 5);
//prints 0,1,2,3,4 in separate lines
```

For Loop

```
for (int i = 0; i < 10; i++) { printf("%d", i); } // Loop body
//prints from 0 to 9
```

Nested Loop Nested Loops are loops inside another loop, often used for multi-dimensional arrays.

```
for(int i = 0; i < 3; i++) { for(int j = 0; j < 3; j++) { printf("%d ", i*j); } printf("\n"); }

/*
Output:
0 0 0
0 1 2
0 2 4
*/
```

Infinite Loop An Infinite Loop runs without any termination condition, often by mistake.

```
while(1) { printf("This loop will run forever.\n"); }
```

Loop Control Statements Control statements like break and continue alter the flow of loops.

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

//prints 0,1,2,3,4 in separate lines

The break Statement The break statement terminates the nearest enclosing loop or switch statement.

```
for (int i = 0; i < 10; i++) {
 if (i == 5) break;// Output: Stops the loop when i equals 5
 // Other code
}
```

The continue Statement The continue statement skips the current iteration of a loop.

```
for (int i = 0; i < 10; i++) {
 if (i == 5) continue;// Skips printing when i equals 5
 printf("%d ", i);// Output: 0 1 2 3 4 6 7 8 9
}
```

Switch

- **Purpose:** The switch statement provides a convenient way to handle multiple cases of a single variable without resorting to multiple if-else statements.

Switch syntax Switch syntax includes switch keyword, expression, and case/default blocks. Case labels specify the code to execute for specific values in a switch The default label is executed if no case matches the switch expression.

```
switch (expression) {
 case x:
 // code
 break;
 default:
 // code
}
```

Strings

Definition of String A string in C is an array of characters terminated by a null character '\0'.

char str[] = "Hello, World!"; // Declares a string.

String Initialization Strings can be initialized using string literals or character arrays.

char str[14] = "Hello, World!"; // Initializes a string.

strlen() Function The strlen() function calculates the length of a string, excluding the null terminator.

size_t len = strlen("Hello"); // len is 5.

strcpy() Function strcpy() is used to copy one string to another in C.

char dest[20];

strcpy(dest, "Hello World"); // Copies string.

String Concatenation The strcat() function appends one string to the end of another.

char str1[20] = "Hello, ";

char str2[] = "World!";

strcat(str1, str2); // str1 is now "Hello, World!"

String Comparison The strcmp() function compares two strings lexicographically.

int result = strcmp("hello", "world"); // Compares two strings.

strcmp() returns an integer value.

- If string1 is lexicographically less than string2, it returns a negative value.
- If string1 is lexicographically greater than string2, it returns a positive value.
- If both strings are equal, it returns 0.

Arrays

Definition Arrays are contiguous memory locations holding elements of the same data type.

int arr[10]; // Declares an array of 10 integers.

Initialization Arrays can be initialized with a list of values at declaration.

int arr[3] = {1, 2, 3}; // Initializes an array with values.

Accessing Elements Elements are accessed using an index, starting from zero.

arr[0] = 10; // Sets the first element of arr to 10.

Multi-dimensional Multi-dimensional arrays are arrays of arrays, like a matrix.

int matrix[3][3]; // Declares a 3x3 two-dimensional array.

Size The size of an array is fixed at compile time and can't be changed.

sizeof(arr); // Returns the total size in bytes of arr.

Passing Array to Functions Arrays are passed to functions by reference, not by value in C language only.

void func(int myArr[]); // Function taking an array as an argument.

Pointers

A pointer is a variable that stores the memory address of another variable.

int *ptr; // Declares a pointer to an integer.

Pointer Initialization Pointers can be initialized to the address of a variable using the & operator.

int var = 10; int *ptr = &var; // Initializes pointer with address of var.

Dereferencing Pointers Dereferencing a pointer means accessing the value at the address the pointer points to.

int value = *ptr; // Dereferences ptr to get the value of var.

Pointer to Pointer A pointer to a pointer is a variable that stores the address of another pointer.

int **pptr = &ptr; // Declares a pointer to a pointer.

Pointer Arithmetic Pointer arithmetic allows pointers to navigate through arrays by incrementing or decrementing.

ptr++; // Moves to the next integer position in memory.

Null Pointer A null pointer points to no valid memory location and is often used for error checking.

int *ptr = NULL; // Initializes ptr as a null pointer.

Dynamic Memory Allocation Pointers are used with dynamic memory allocation functions like malloc and free.

int *ptr = (int *)malloc(sizeof(int)); // Allocates dynamic memory for an integer.

Free Free() in C is used to dynamically de-allocate the memory.

free(ptr);

Calloc() calloc() in C is used to dynamically allocate the specified number of blocks of memory of the specified type.

ptr = (int*) calloc(25, sizeof(int)); //This statement allocates contiguous space in memory for 25 elements each with the size of the int.

Function

A function is a block of code that performs a specific task.

Declaration Function declaration tells the compiler about a function's name, return type, and parameters.

int add(int, int); // Function declaration.

Definition Function definition provides the actual body of the function.

int add(int a, int b) { return a + b; } // Function definition.

Function Call A function call is an expression that passes control and arguments to a function.

int result = add(5, 3); // Calling the function.

Parameters Parameters are variables that accept values passed to the function.

int add(int a, int b) { return a + b; } // 'a' and 'b' are parameters.

Arguments Arguments are the actual values passed to the function when it is called.

int result = add(5, 3); // 5 and 3 are arguments.

Return Type The return type specifies the data type of the value the function returns.

int add(int a, int b) { return a + b; } // Return type is 'int'.

Void Void functions do not return a value.

void printMessage() { printf("Hello, World!\n"); } // Void function.

Parameter Passing Functions can take parameters, allowing for customization of tasks.

void drawCircle(int radius) {

// Drawing logic

}

Pass By Value Passing by value copies the actual value of an argument into the formal parameter of the function.

void function(int a) { /* ... */ }

int main() { int x = 5; function(x); }

Pass By Reference Passing by reference passes the address of the argument, allowing the function to modify the original variable.

void function(int *a) { /* ... */ }

int main() { int x = 5; function(&x); }

Pass By Pointer Similar to by reference, passing by pointer allows a function to modify the variable to which the pointer points.

void function(int *a) { *a = 10; }

int main() { int x = 5; function(&x); }

Scope Rule of Functions

Local Scope Local scope refers to variables defined within a function.

void func() { int local_var = 5; // Local scope variable }

Global Scope Global scope refers to variables defined outside of all functions.

int global_var = 10; // Global scope variable

Block Scope Block scope is limited to the block where variables are defined.

if (true) { int block_var = 20; // Block scope variable }

Function Scope Function scope refers to the entire body of the function.

void anotherFunction() { // Function scope starts here }

Structure

A structure in C is a composite data type that groups variables under a single name.

struct Person {

char name[50];

int age;

float salary;

};

Structure Initialization Structures can be initialized with values for their members.

struct Person john = {"John Doe", 30, 50000.0};

Pointer to Structure Pointers can be used to access and manipulate structure data.

struct Person *ptr = &john;

ptr->age = 31;

Array of Structures An array of structures creates a sequence of structure instances.

struct Person employees[5];

Accessing Members Structure elements are accessed using the dot (.) operator.

struct Point {int x; int y;};

struct Point p1 = {10, 20};

int x_val = p1.x; // Accessing structure member

Pointer to Structure Use the arrow (->) operator to access members via a pointer.

struct Point *ptr = &p1;

int y_val = ptr->y; // Accessing via pointer

Nested Structures Nested structures are accessed using a combination of dot (.) operators.

struct Line {struct Point start; struct Point end;};

int start_x = line.start.x; // Nested access

Union

A union is a user-defined data type that can hold different data types in the same memory location.

union Data { int i;

float f;

char str[20]; }; // Union declaration

Memory Sharing Unions share the same memory space for all its members.

union Data data;

data.i = 4; // Memory shared with f and str

Size of Union The size of a union is the size of its largest member.

printf("Size of union: %lu", sizeof(data)); // Outputs size of largest member

Accessing Members Members of a union are accessed using the dot operator.

data.f = 3.14; // Accessing the float member

Union with Structures Unions can be members of structures to save space.

```
struct Item {
char type;
union {
 int i;
float f;
}
data; }; // Struct with union
```

File

fopen fopen is a function used to open a file and create a file stream.

FILE *fp = fopen("file.txt", "r"); // Opens a file for reading.

fopen modes fopen modes define how a file is to be accessed (read, write, etc.).

FILE *fp = fopen("file.txt", "w"); // Opens a file for writing.

fread() fread() reads data from the given stream into the array pointed to by ptr.

```
#include <stdio.h>

int main() {
  FILE *file;
  char buffer[100]; // Buffer to store data read from the file

  // Open the file in binary mode for reading
  file = fopen("example.txt", "rb");

  if (file == NULL) {
    printf("Error opening file");
    return 1;
```

```c
    }

    // Read data from the file into the buffer
    fread(buffer, sizeof(char), 100, file);

    // Output what was read from the file
    printf("Content read from file: %s\n", buffer);

    // Close the file
    fclose(file);

    return 0;
}
```

fwrite() The fwrite function writes binary data to a file.

```c
int numbers[] = {1, 2, 3};
FILE *fp = fopen("data.bin", "wb");
if (fp) {
 fwrite(numbers, sizeof(int), 3, fp);
 fclose(fp);
}
```

fprintf() The fprintf function writes formatted text to a file.

```c
FILE *fp = fopen("log.txt", "a");
if (fp) {
 fprintf(fp, "Error: %s\n", errorMsg);
 fclose(fp);
}
```

putc() putc is used to write a single character to a file.

```c
FILE *fp = fopen("example.txt", "w");
if (fp) {
 putc('A', fp);
fclose(fp);
```

}

fclose() fclose function closes an open file pointed to by a file pointer.

fclose(filePtr); // Closes the file associated with filePtr.

Macro

Macro Definitions Macros are a piece of code in a program which is given some name.

#define PI 3.14 // Defines a macro named PI.

Header Files File inclusion in C allows the inclusion of header files in a source file.

#include <stdio.h> // Includes the standard input/output header file.

Conditional Compilation Conditional compilation includes or excludes code based on certain conditions.

#ifdef DEBUG

// Code for debugging

#endif // Ends the conditional.

#if Directive The #if directive checks if a condition is true for compilation.

#if defined(DEBUG)

// Debug code here

#endif

#elif Directive The #elif directive is an 'else if' for conditional compilation.

#if defined(WIN32)

// Windows code

#elif defined(LINUX)

// Linux code

#endif

Macros with Arguments Macros with arguments allow for code reuse and can take parameters.

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

int result = SQUARE(5); // result is 25

Enumeration

An enumerated type is a user-defined data type in C. It consists of integral constants assigned to unique names.

enum color { RED, GREEN, BLUE }; // Defines an enumeration.

Assigning Values By default, enumeration constants are assigned integer values starting from 0.

enum color { RED, GREEN, BLUE }; // RED=0, GREEN=1, BLUE=2

Custom Values You can assign custom integer values to enumeration constants.

enum color { RED = 1, GREEN, BLUE }; // GREEN=2, BLUE=3

typedef

typedef creates an alias for a data type, simplifying code readability and maintenance.

typedef unsigned long ulong; // ulong now represents unsigned long.

typedef for structs typedef is often used with structs to avoid using the struct keyword repeatedly.

typedef struct { int x; int y; } Point; // Point can now be used to declare struct variables.

typedef for pointers typedef can simplify pointer type definitions, making them easier to understand.

typedef char *string; // string can now be used as an alias for char pointers.

typedef for function pointers typedef simplifies the syntax for declaring function pointers.

typedef void (*FunctionPtr)(); // FunctionPtr can now be used for function pointers.

typedef for complex types typedef can rename complex data types, making them easier to work with.

typedef struct { int num; char *name; } *PersonPtr; // PersonPtr is a pointer to such a struct.