Semester Final

CSE 4107

Chapter - 07 (Basic Types)

7.3 Character Types

Today's most popular Character Set is the ASCII (American Standard Code of Information Interchange) is a 7-bit code capable of representing 128 characters. A variable of character can be assigned any single character.

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[END OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

```
char ch;
ch = 'a' // lowercase character
ch = 'A' // uppercase character
ch = '0' // numeric character
ch = ' ' // escape character
```

The single characters are enclosed in single quotation marks.

Operation on Characters

C treats characters as small integers i.e. all data including characters are encoded in binary for storage.

```
printf("%c\n", 97); // Output : a
printf("%d\n", 'a'); // Output : 97
```

The characters are stored in the memory of the computer as the equivalent binary value of the ASCII value of each character. Where the ASCII equivalent of ‘a’ is 97.
The equivalent ASCII value of ‘A’ is 65. The equivalent ASCII value of ‘0’ is 48 and the equivalent ASCII value of ‘ ‘ is 32.

```
char ch;
int i;
i = 'a'; // i has value 97(dec), 01100001(binary)
ch = 97; // ch is set to the value ‘a’(character)
ch = ch + 1; // now ch is set to ‘b’
ch++; // now ch is set to ‘c’
printf("%c\n", i); // prints the character ‘a’
printf("%c\n", ch); // prints the character ‘c’
printf("%d\n", ch); // prints the equivalent ASCII of ‘c’ i.e. 99
```

Characters can be compared just like integers.

```
if(‘a’ <= ch && ch >= ‘z’)
   ch = ‘A’ + (ch - ‘a’);

for(char ch = ‘a’; ch <= ‘z’; ch++) // we can also run a loop
```

Modulo Operation :

When using arithmetic operators (including %), char values are promoted to int before the operation. The result of the modulo operation is an int.

```
char a = 25;
char b = 7;
int result = a % b;  // result will be 4 (25 % 7)
```

```
printf("%d\n", result);
// You can also store the result back in a char
char c = a % b;
printf("%c\n", c); // showing end of transmission
```

Signed And Unsigned Characters :

- Signed characters normally have values from -128 to 127
- Unsigned characters normally have values from 0 to 255

C standard does not define the type of char whether signed or unsigned. But if it matters then use signed char or unsigned char.

```
signed char sch;
unsigned char uch;
```

Arithmetic Types :

The integer types and floating types collectively are called arithmetic types.

1. Integral types
    - char
    - Signed integer types (signed char, short int, int, long int)
    - Unsigned integer types (unsigned char, unsigned short int, unsigned int, unsigned long int)
    - Enumerated types
2. Floating types (float, double, long double)

In C99 the hierarchy for arithmetic types :

1. Integer types
    - char
    - Signed integer types, both standard (signed char, short int, int, long int, long long int) and extended
    - Unsigned integer types, both standard (unsigned char, unsigned short int, unsigned int, unsigned long int, unsigned long long int, _Bool) and extended
    - Enumerated types
2. Floating types
    - Real floating types (float, double, long double)
    - Complex types (float_Complex, double_Complex, long double_Complex)

Escape Sequence :

Escape sequences are special character combinations that represent non-printable or special characters in strings and character constants. They begin with a backslash (\) followed by a specific character.

Common Escape Sequences in C:

Escape Sequence	Description	ASCII Value
`\a`	Alert (bell)	7
`\b`	Backspace	8
`\f`	Form feed (new page)	12
`\n`	Newline (line feed)	10
`\r`	Carriage return	13
`\t`	Horizontal tab	9
`\v`	Vertical tab	11
`\\`	Backslash	92
`\'`	Single quote	39
`\"`	Double quote	34
`\?`	Question mark	63
`\0`	Null character (terminates strings)	0

When we write ‘\012’ it denotes ‘\n’. Here ‘\012’ is considered the octal representation of the ASCII code for the new line sequence. If we write ‘\12’ it is also considered an octal representation. We can also use hexadecimal representation. For example: ‘\x0a’ also represents ‘\n’. We just cannot use the decimal equivalent of the ASCII code to represent the sequence. C does not support decimal escape sequences.

```
printf("Hello %c", '\n'); // New line sequence
printf("Hello %c", '\012'); // Octal representation of New line
printf("Hello %c", '\12'); // Also Octal representation of New line
printf("Hello %c", '\x0a'); // Hexadecimal representation of New line
```

Character Handling Functions :

In the directive #include<ctype.h> there are two functions to convert uppercase letters to lowercase and vice versa.

1. `toupper(ch)` :
   Function Implementation :

   ```
   ch = toupper(ch) ; // converts into uppercase letters
   ```

   Manual Implementation :

   ```
   if(ch >= 'a' && ch <= 'z') ch = 'A' + (ch - 'a');
   ```

2. `tolower(ch)` :
   Function Implementation :

   ```
   ch = tolower(ch); // converts into lowercase letters
   ```

   Manual Implementation :

   ```
   if(ch >= 'A' && ch <= 'Z') ch = 'a' + (ch - 'A');
   ```

Reading And Writing Characters using scanf and printf :

The %c conversion is used for scanf and printf for character type.

```
scanf("%c", &ch); // reads a single character
printf("%c", ch); // writes a single character
```

scanf does not skip white spaces. So in the previous example if the next unread character is a white space the variable ch will store the white space. That's why we will force scanf to skip white spaces.

```
scanf(" %c", &ch);
```

For taking a line as input using only char type :

```
do {
    scanf("%c", &ch);
} while (ch != '\n');
```

Reading And Writing Characters Using getchar() and putchar() :

- Each time getchar() is called it reads one character, which it returns. In order to save it we need to use assignment to store it.

  ```
  char ch = getchar();
  ```

  getchar() does not skip whitespaces as it reads the characters.
- getchar() and putchar() are implemented as macros for additional speed.

  ```
  putchar(ch)
  ```

Example :

```
while ((c = getchar()) != '\n') {
        putchar(c);
    }
```

Program : Determine the length of a Message

```
#include <stdio.h>
int main() {
    int len = 0;
    printf("Enter a message : ");
    char ch;
    while((ch = getchar()) != '\n'){
        len++;
    }
    printf("Your message was %d character(s) long.\n", len);
    return 0;
}
```

7.4 Type Conversion

In C programming, type conversion (also called type casting) can be performed implicitly by the compiler or explicitly by the programmer. Here's a detailed explanation:

Implicit Type Conversion (Automatic)

The C compiler automatically performs implicit conversions when:

- Different types are mixed in expressions
- Assigning a value of one type to a variable of another type
- Passing arguments to functions

Rules for Implicit Conversion:

1. **Integer Promotion**: Smaller integer types (`char`, `short`) are promoted to `int` or `unsigned int`
2. **Usual Arithmetic Conversion**: When operating on two different types:
    - If one operand is `long double`, convert the other to `long double`
    - Else if one is `double`, convert to `double`
    - Else if one is `float`, convert to `float`
    - Else perform integer promotion on both, then:
        - If one is `unsigned long int`, convert to `unsigned long int`
        - Else if one is `long int` and the other is `unsigned int`, convert both to `unsigned long int`
        - Else if one is `long int`, convert to `long int`
        - Else if one is `unsigned int`, convert to `unsigned int`
        - Else both become `int`

Explicit Type Conversion (Type Casting)

Syntax :

```
(type_name) expression
```

Example :

```
double x = 5.7;
    int y;
    y = (int)x;  // Explicit conversion (truncation)
    printf("%d\n", y);  // Output: 5
    int a = 10, b = 3;
    float div = (float)a / b;  // Convert a to float before division
    printf("%f\n", div);  // Output: 3.333333
```

The Usual Arithmetic Conversion :

- The type of either operand is a floating type :
  float => double => long double
- Neither operand type in a floating type :
  int => unsigned int => long int => unsigned long int

Example :

```
char c;
short int s;
int i;
unsigned int u;
long int l;
unsigned long int ul;
float f;
double d;
long double ld;
i = i + c; // c is converted into int
i = i + s; // s is converted into int
u = u + i; // i is converted into unsigned int
l = l + u; // u is converted into long
ul = ul + l; // l is converted into unsigned long int
f = f + ul; // ul is converted into float
d = d + f; // f is converted into double
ld = ld + d; // d is converted into long double
```

Conversion During Assignment :

In C, the expression on right is converted into the type of the expression on the left.

```
char c;
int i;
float f;
double d;
i = c; // char type converted into int
f = i; // int type converted into float
d = f; // float type converted into double
```

```
int i2;
i2 = 856.34; // now i2 is 856
i2 = -856.34; // now 12 is -856
```

Implicit Conversion in C99 :

Ranking Highest to Lowest :

1. long long int, unsigned long long int
2. long int, unsigned long int
3. Int, unsigned int
4. short int, unsigned short int
5. char, signed char, unsigned char
6. _Bool

Casting :

A cast expression :

```
(type_name) expression
```

type_name specifies the type to which the expression should be converted.

```
float f, frac_part;
frac_part = f - (int) f;
int i = (int) f; // f is converted into int;

float quotient;
int dividend, divisor;
quotient = (float) dividend / (float) divisor;

long i;
int j = 10000;
i = (long) j * j;
```

7.5 Type Definition

The typedef keyword in C allows you to create aliases (new names) for existing data types
This is extremely useful for:

1. Improving code readability
2. Creating platform-independent types
3. Simplifying complex type declarations
4. Making code easier to maintain

.

```
#define BOOL int
typedef int BOOL;
```

We can use typedef instead of define.

Advantages of Using typedef :

1. Code Readability and Abstraction
2. Portability Benefits
3. Maintenance Advantages

Type Definition and Portability :

1. Fixed-Size Integer Types
2. Floating-Point Consistency

7.6 The sizeof Operator :

The sizeof operator allows a program to determine how much memory is required to store values of a particular type.

```
sizeof (type-name)
```

```
printf("char: %zu bytes\n", sizeof(char));        // Always 1
printf("short: %zu bytes\n", sizeof(short));      // Typically 2
printf("int: %zu bytes\n", sizeof(int));          // Typically 4
printf("long: %zu bytes\n", sizeof(long));        // 4 or 8
printf("long long: %zu bytes\n", sizeof(long long)); // Typically 8
printf("float: %zu bytes\n", sizeof(float));      // Typically 4
printf("double: %zu bytes\n", sizeof(double));    // Typically 8
```

Code Examples :

1. **Write a program that translates an alphabetic phone number into a numeric form :**

```
#include <stdio.h>
int main()
{
    char ch;
    printf("Enter phone number : ");
    while ((ch = getchar()) != '\n')
    {
        switch (ch)
        {
        case 'A': case 'B': case 'C':
            printf("2");
            break;
        case 'D': case 'E': case 'F':
            printf("3");
            break;
        case 'G': case 'H': case 'I':
            printf("4");
            break;
        case 'J': case 'K': case 'L':
            printf("5");
            break;
        case 'M': case 'N': case 'O':
            printf("6");
            break;
        case 'P': case 'R': case 'S':
            printf("7");
            break;
        case 'T': case 'U': case 'V':
            printf("8");
            break;
        case 'W': case 'X': case 'Y':
            printf("9");
            break;
        default:
            printf("%c", ch);
        }
    }
```

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

2. Scribble Value

```
#include <stdio.h>
#include <ctype.h>
int main()
{
    char ch;
    int result = 0;
    printf("Enter a word : ");
    while ((ch = toupper(getchar())) != '\n')
    {
        switch (ch)
        {
        case 'A':case 'E':case 'I':case 'L':case 'N':case
'O':case 'R':case 'S':case 'T':
            result += 1;
            break;
        case 'D':case 'G':
            result += 2;
            break;
        case 'B':case 'C':case 'M':case 'P':
            result += 3;
            break;
        case 'F':case 'H':case 'V':case 'W':case 'Y':
            result += 4;
            break;
        case 'K':
            result += 5;
        case 'J':case 'X':
            result += 8;
            break;
        case 'Q':case 'Z':
            result += 10;
            break;
        }
```

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

3. **Write a program that asks the user a 12 hour format time and prints a 24 hour format time**

```
#include <stdio.h>
int main()
{
    int hh, mm;
    char ch;
    printf("Enter the time as 12 hour format : ");
    scanf("%d:%d %c", &hh, &mm, &ch);
    if (ch == 'P')
        hh += 12;
    printf("%d:%d\n", hh, mm);
    return 0;
}
```

4. **Count the vowels in the sentence**

```
#include <stdio.h>
#include <ctype.h>
int main()
{
    int count = 0;
    char ch;
    printf("Enter the sentence : ");
    while ((ch = tolower(getchar())) != '\n')
    {
        switch (ch)
        {
        case 'a':case 'e':case 'i':case 'o':case 'u':
            count++;
            break;
        }
```

```
    }
    printf("Vowel Count of the Sentence : %d\n", count);
    return 0;
}
```

5. Determine average length of words in a sentence

```
#include <stdio.h>
int main()
{
    double len = 0.0, wordCount = 0.0;
    printf("Enter the sentence : ");
    char ch;
    while ((ch = getchar()) != '\n')
    {
        if (('A' <= ch && ch <= 'Z') || ('a' <= ch && ch <= 'z'))
        {
            len++;
        }
        else if (ch == ' ')
        {
            wordCount++;
        }
    }
    wordCount++;
    printf("Average length of words : %.2lf\n", len / wordCount);
    return 0;
}
```

Chapter - 08 (Arrays)

Aggregate variables : Variables which can store collections of values. And there are two kinds of aggregates in C: arrays and structures.

8.1 One Dimensional Arrays

An array is a data structure containing a number of data values, all of which have the same type.
Each individual value is known as separated elements and can be individually selected by their position within the array.
The elements of a one dimensional array are conceptually arranged one after another in a single row.

Declaration of a one dimensional array :

```
data-type array_name[size]; // declaration of an array
int arr[5]; // the array a that has 10 elements of type int

#define N 10 // defining the value of N as 10
. . . .
int arr[N]; // the array a that has 10 elements of type int
```

If we need to change the size of the array later on we can just change the value of N.

Array Subscripting :

To access any particular element of an array, we write the array name followed by an integer value on the square brackets (subscripting/indexing).
Index starts from 0 to n-1.

```
for(int i=0; i<N; i++) // clears all elements
    arr[i] = 0;
for(int i=0; i<N; i++) // takes inputs in all indices
    scanf("%d", &arr[i]);
for(int i=0; i<N; i++) // sums up all the elements
    sum += arr[i];
```

Note : We must use & symbol when calling scanf to read an array element, just as we would with ordinary elements.

Array Index out of Bound :

This error occurs when a program tries to access an array element at an index that is outside its valid range.

```
int arr[6] = {10, 20, 30, 40, 50, 60};  // Indexes: 0 to 5
int x = arr[7];  // ERROR: Index 7 is out of bounds
```

- Valid indices: `0` to `5` (for an array of size `6`).
- Invalid access: `arr[6]` or `arr[7]` → Undefined Behavior (may crash, return garbage, or corrupt memory).

The compiler doesn't check.

- Accessing out-of-bounds memory does not always crash immediately.
- It might:
  - Return garbage values.
  - Corrupt other variables.
  - Cause a segmentation fault (if accessing restricted memory).

Expression in Array Subscript :

Array subscripts can have expressions as long as the expression evaluates to an integer number.

```
arr[i + j * 10] = 0;
```

This part is important for output prediction

```
i = 0;
while (i < n)
    arr[i++] = 0;
i = 0;
while (i < n)
    arr[i] = arrCpy[i++]; // copies elements to another array
```

Program : Reversing a Series of Numbers

```
#include<stdio.h>
#define N 10
int main(){
    int arr[N];
    printf("Enter the elements of the array : ");
    for(int i=N-1; i>=0; i--){
        scanf("%d", &arr[i]);
    }
```

```
    printf("The elements in reverse order : ");
    for(int i=0; i<N; i++){
        printf("%d ", arr[i]);
    }
    printf("\n");
    return 0;
}
```

Array Initialization :

```
int arr[5] = {1, 2, 3, 4, 5};
```

If the initializer is shorter than the array, the remaining elements of the array are given the value of 0.

```
int arr[5] = {1, 2, 3}; // here the elements in index 3 and 4 are set to 0
int arr[5] = {0}; // all the elements are set to 0
```

Note : It is illegal to not initialize the elements of an array ar all so we put a single 0 as initializer in the second example

Undesignated (Standard) Initialization

- Elements are assigned in order.
- Unspecified values are set to **zero** (for static arrays).

```
int arr[5] = {10, 20, 30};  // Rest are zeros
```

Designated Initialization (C99 and later)

- Explicitly assigns values to specific indices.
- Unspecified elements are **zero-initialized**.

```
int arr[5] = {[1] = 20, [3] = 40};  // Others are zero
```

Program : Checking A Number For Repeated Digits

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

```
#include <stdbool.h> // C99
int main()
{
    bool digit_seen[10] = {false}; // setting all elements to false
    int digit;
    long n;
    printf("Enter a number : ");
    scanf("%d", &n);    %ld instead of d
    while (n > 0)
    {
        digit = n % 10;
        if (digit_seen[digit])
            break;
        digit_seen[digit] = true;
        n /= 10;
    }
    if (n > 0)
        printf("Repeated digits.\n");
    else
        printf("No repeated digits.\n");
    return 0;
}
```

Using the sizeof operator with Arrays :

Using sizeof operator we can determine the array size.

```
int arr[10] = {0};
int arraySize = sizeof(arr) / sizeof(arr[0]);
```

The sizeof(arr) will give 10*4 = 40 bytes memory and the sizeof(arr[0]) will give 4 bytes. When the division is done arraySize will have 10 which is the exact size of the array.

Program : Computing Interest

// Will write this program afterwards

8.2 Multidimensional Arrays

Multidimensional arrays are arrays of arrays. They are used to represent matrices, grids, or tables. The most common type is the 2D array (like a table with rows and columns), but higher dimensions (3D, 4D, etc.) are also possible.

```
int mat[5][9];
```

This is a 2D array containing 5 rows and 9 columns.

0	1	2	3	4	5	6	7	8
1								
2								
3								
4								

To access the element in i-row and j-column of mat array we need to write mat[i][j]. All the elements are stored sequentially in row major order.

```
int m[3][3] = {1, 2, 3,
               4, 5, 6,
               7, 8, 9}
```

The given array is stored like this

1	2	3	4	5	6	7	8	9

```
#define N 10
   double arr[N][N];
   int row, col;
   for(row=0; row<N; row++){
       for(col=0; col<N; col++){
           if(row==col){
               arr[row][col]=1.0;
           }
           else{
               arr[row][col]=0.0;
```

```
            }
        }
    }
```

Initializing Multidimensional Array :

```
int mat[5][9] = {{1, 1, 1, 1, 1, 1, 0, 0, 1},
                 {0, 1, 0, 0, 1, 1, 1, 1, 0},
                 {1, 1, 1, 0, 0, 1, 0, 0, 0},
                 {0, 0, 0, 1, 1, 1, 1, 0, 1},
                 {0, 1, 1, 1, 0, 0, 1, 1, 1}};
```

- If an initializer isn't large enough to fill a multidimensional array, the remaining elements are given the value 0.
- If the inner list isn't long enough to fill a row, the remaining elements are given the value 0. For this the inner braces are a must.
- We can even omit the inner braces.

Designated Initialization of Multidimensional Array :

```
double ident[2][2] = {[0][0] = 1.0, [1][1] = 1.0}
```

Constant Arrays :

Whether one dimensional arrays or multidimensional arrays they can be made constant by starting its declaration with the word const.

```
const int arr[] = {1, 2, 3, 4, 5};
```

The elements of this array can not be modified afterwards.

Program : Dealing A Hand Of Cards

// Will write the Program Later on

8.3 Variable-Length Array

The program for reversing the elements of an array can also be done with VLA (Variable Length Array.

```
#include <stdio.h>
int main() {
    int n;
    printf("Enter the number of elements : ");
    scanf("%d", &n);
    int arr[n]; // variable length array
    printf("Enter the elements of the array : ");
    for(int i=0; i<n; i++){
        scanf("%d", &arr[i]);
    }
    printf("Reverse Order : ");
    for(int i=n-1; i>=0; i--){
        printf("%d ", arr[i]);
    }
    printf("\n");
    return 0;
}
```

Chapter - 09 (Functions)

A function in C is a self-contained block of code that performs a specific task. Functions are fundamental building blocks of C programs, allowing for modular programming and code reuse.

- A function is a series of statements that have been grouped together and given a name.
- Each function is a small program with its own declarations and statements.

```
if(condition){
   statement;
   statement;
}
```

Here a group of statements in the parentheses where the statements do a work.

Characteristics of Functions in C:

1. **Modularity**: Break down complex programs into smaller, manageable pieces
2. **Reusability**: Write once, use multiple times
3. **Abstraction**: Hide implementation details behind a simple interface

Scope of Variable :

The variable if declared in a curly braces, won't work outside the curly braces. Function is also like that, if you use variables outside the block, then it won't be valued really.

Procedure Vs Function :

After execution, the function returns something.

```
double avg(){
   -  - - - - -
   return avg;
}
```

Procedure does not return anything. void() function doesn't return anything. So, this can be called a procedure.

9.1 Defining and Calling Functions

Program : Computing Averages

```
#include<stdio.h>
double average(double a, double b)
{
    return (a + b) / 2.0;
}
int main()
{
    double a, b, c;
    printf("Enter three numbers : ");
    scanf("%lf %lf %lf", &a, &b, &c);
    printf("The average of a and b : %lf\n", average(a, b));
    printf("The average of b and c : %lf\n", average(b, c));
    printf("The average of c and a : %lf\n", average(c, a));
    return 0;
}
```

Program : Printing Countdown

```
#include<stdio.h>
void print_count(int n) // does not return a value
{
    printf("T minus %d and still counting.\n", n);
}
int main()
{
    int i;
    for(i = 10; i > 0; i--)
    {
        print_count(i);
    }
    return 0;
}
```

This function does not return a value rather executes the statements written inside the blocks.

Program : Printing a Pun

```
#include<stdio.h>
void print_pun(void)
{
    printf("To C,or not to C : this is the question.\n");
}
int main()
{
    print_pun();
    print_pun();
    return 0;
}
```

Function Definitions :

A **function definition** in C is a block of code that provides the complete implementation of a function.

```
return-type function-name(parameters)
{
  declarations;
  statements;
}
```

It specifies the-

1. The function's name (identifier used to call it).
2. Its return type (the data type of the value it returns, or void if nothing is returned).
3. Its parameters (input variables, if any) with their data types.
4. The function body (the actual code that executes when the function is called).
5. A return statement (if the function returns a value).

Note :

- The return type of a function is the type of value that the function returns.
- Rules governing the return type :
    - Function may not return arrays.

    - Specifying that the return type is void indicates that the function doesn't return a value.
- Function must use curly braces.

```
int sum(int a, int b) { - - - }
void printArray(int* arr, int n) { - - - }
```

```
double average (double a, double b)
{
    double sum; // declaration
    sum = a + b; // statement
    return sum / 2; // statement
}
```

Function Calls :

A function call consists of a function named followed by a list of arguments, enclosed in parentheses.

```
average (x, y);
print_count(i);
print_pun();
```

A call of a void function is always followed by a semicolon to turn it into a statement.

```
print_count(i);
print_pun();
```

A call of non-void function produces a value that can be stored in a variable, tested, printed or used in some other ways.

```
avg = average(x, y);
if(average(x, y) > 0)
   printf("The average is positive.\n");
printf("The average is %g.\n", average(x, y));
```

Ignoring the return value of average is an odd thing to do, but for some functions it makes sense.

```
num_chars = printf("Life is like a box of chocolates!\n"); // discards
return value
```

The printf function returns the number of characters that it prints. After the given call, num_chars will have the value 34. We will normally discard printfs return value.

Program : Testing Whether a Number is Prime

```
#include<stdio.h>
#include<stdbool.h>
bool isPrime(int n){
    int divisor;
    if(n <= 1)
       return false;
    for(divisor = 2; divisor * divisor <= n; divisor++)
    {
        if(n % divisor == 0)
           return false;
    }
    return true;
}
int main()
{
    int n;
    printf("Enter a number : ");
    scanf("%d", &n);
    if(isPrime(n))
       printf("The number %d is a Prime Number\n", n);
    else
       printf("The number %d is not a Prime Number\n", n);
    return 0;
}
```

Note : Function Name is the identifier.

Parameters :

- Each Parameter is preceded by a specification of its type.
- Separated by commas

- In void functions sometimes there are no parameters
- We cannot write it like void fun(int a, b, c, d). Rather, we need to write it like void fun(int a, int b, int c, int d).

Function Body :

- May include both declarations and statements.
- Variations of one function cannot be accessed by other functions.
- Might be empty if the return type is void.
- A Function body can have nothing.

9.2 Function Declaration

A function declaration (also called a function prototype) informs the compiler about a function's existence, its return type, and its parameters before the function is actually defined or used.

1. Explicit Function Declaration

An explicit declaration is a forward declaration that appears before the function is called.

```
return_type function_name(parameter_list);  // Note the semicolon
```

```
#include <stdio.h>
// Explicit declaration (prototype)
int add(int a, int b);
int main()
{
    int result = add(3, 5);  // Function call
    printf("Sum: %d\n", result);
    return 0;
}
// Function definition
int add(int a, int b)
{
    return a + b;
}
```

2. Implicit Function Declaration (Outdated & Risky)

If a function is called without any prior declaration, the compiler assumes an implicit declaration (defaulting to int return type and unknown parameters).

```
#include <stdio.h>
int main()
{
    // No declaration before call → Implicit "int add()" assumed
    int result = add(3, 5);
    printf("Sum: %d\n", result);
    return 0;
}
// Function definition
int add(int a, int b)
{
    return a + b;
}
```

Modern C Standards (C99 & Later)

- Implicit declarations are illegal (compilers like `gcc` throw errors).
- Explicit declarations are mandatory for type safety.

9.3 Arguments

In C programming, arguments are the values or expressions that are passed to a function when it is called. The values are assigned to the corresponding parameters in the function definition.

- Parameters are variables that are declared in function definition that receive the arguments.
- Arguments are values passed to a function during function call.

```
int power(int x, int n)
{
    int i, result = 1;
    for(i = 1; i <= n; i++)
    {
        result *= x;
```

```
    }
    return result;
}
```

Function Call : `power(2, 4);`

Syntax :

```
// Function definition with parameters
return_type function_name(parameter1, parameter2, ...) {
    // function body
}

// Function call with arguments
function_name(argument1, argument2, ...);
```

```
void decompose(double x, long int_part, double frac_part)
{
    int_part = (long) x;
    frac_part = x - int_part;
}

// Function Call
decompose(3.1416, i, d);
```

Passed by Value

When arguments are passed by value, a copy of the argument's value is made and passed to the function. The original variable remains unchanged.

```
void increment(int x) {
    x++;
}
int main() {
    int num = 5;
    increment(num);
    printf("%d", num); // Output: 5 (unchanged)
```

```
    return 0;
}
```

Advantages:

- Original data is protected from accidental modification
- Simple to understand and implement
- Works well for small data types

Disadvantages:

- Overhead of copying large data structures
- Cannot modify the original variable directly
- May consume more memory for large arguments

Passed by Reference

In C, this is simulated using pointers. The address of a variable is passed, allowing the function to modify the original value.

```
void increment(int *x) {
    (*x)++;
}
int main() {
    int num = 5;
    increment(&num);
    printf("%d", num); // Output: 6 (changed)
}
```

Advantages:

- Can modify the original variable
- Efficient for large data structures (only pointer is copied)
- Allows functions to return multiple values

Disadvantages:

- More complex syntax with pointers
- Risk of accidentally modifying data
- Potential for pointer-related errors (null pointers, dangling pointers)

Argument Conversions :

Type of argument might not match the type of the parameters. C performs implicit type conversions when arguments don't match parameter types.

- **Integer promotions:** Smaller integer types are promoted to int
- **Usual arithmetic conversions:** Operands are converted to a common type
- **Function prototype conversions:** If a prototype exists, arguments are converted to declared parameter types

```
void func(float f);

int main() {
    int i = 5;
    func(i); // i is converted to float
}
```

Array Arguments :

Arrays are often used as arguments. If the array in one dimensional then the length of the array can be left unspecified.

```
int f(int a[]) // no length specified
{
  ……..
}
```

Why sizeof doesn't work on array parameters in functions?

1. Arrays turn into pointers when passed to functions (called "array decay")
2. sizeof then gives you:
    - Pointer size (usually 4 or 8 bytes) inside the function
    - Actual array size (like 40 bytes for int arr[10]) where the array is declared

```
int f(int a[])
{
    int len = sizeoff(a) / sizeoff(a[0]); // wrong
    ........
}
```

We may pass the length of the array as an argument. As the array passed will mark the pointer of the array.

```
int arrSum(int arr, int n)
{
    int sum = 0;
    for(int i =0; i < n; i++)
    {
        sum += arr[i];
    }
    return sum;
}
// Function call
arrSum(arr, n);
```

Variable Length Array Parameters :

VLAs let you use arrays where the size is determined at runtime (not compile time). When used as function parameters, they allow functions to work with arrays of any size.

```
int sumArray(int arr[], int n); // right
int sumArray(int n, int arr[n]); // right
int sumArray(int arr[n], int n); // wrong
int concentrate(int m, int n, int a[m], int b[n], int c[m + n]);
int sumMatrix(int n, int m, int a[n][m])
{
    int sum = 0;
    for(int i = 0; i < n; i++)
    {
        for(int j = 0; j < m; j++)
        {
            sum += a[i][j];
        }
    }
    return sum;
}
```

Using `static` in Array Parameters Declarations (C99)

C99 allows using static in array parameter declarations to indicate a minimum size.

```
void process(int arr[static 10]) {
    // Compiler can assume arr has at least 10 elements
}
```

Purpose:

- Allows compiler optimizations
- Documents function expectations
- Doesn't enforce the size, just hints at it

Compound Literals (C99)

Compound literals allow creating unnamed objects with specified values.

```
void printArray(int arr[], int size);

int main() {
    printArray((int[]){1, 2, 3, 4, 5}, 5);
}
```

Advantages:

- Convenient for passing temporary values
- Avoids needing to declare variables just for one-time use
- Can be used with both arrays and structures

Disadvantages:

- C99 feature, not available in older compilers
- Lifetime is the enclosing block (like regular variables)

Multidimensional Array :

Only the length of the first dimension may be omitted.

```
void printMatrix(int mat[][3], int rows) {
    for(int i = 0; i < rows; i++) {
        for(int j = 0; j < 3; j++) {
```

```
            printf("%d ", mat[i][j]);
        }
        printf("\n");
    }
}

int main() {
    int matrix[2][3] = {{1, 2, 3}, {4, 5, 6}};
    printMatrix(matrix, 2);
}
```

In C, when you pass a multidimensional array to a function, the compiler needs to know how to calculate memory offsets for accessing elements. Since arrays are stored in row-major order (all elements of a row are contiguous in memory), the compiler must know all dimensions except the first to compute the correct memory locations.

9.4 The return Statement

A non-void function must use the return statement to specify what value it will return.

```
return expression;
```

The expression is often just a constant or variable.

```
return 0;
return status;
return n >= 0 ? n : 0;
return; // return in a void function
```

```
void print_int(int i)
{
    if(i < 0)
       return;
    printf("%d ", i);
}

void print_pun()
{
    printf("Hola!");
```

```
    return; // Ok, but not needed
}
```

9.5 Program Termination

Since main is a function, it must have a return type. Mostly it returns an integer type.

The `exit()` Function

The exit() function is used to terminate a program immediately from any point in the code. It is declared in the <stdlib.h> header file.

Syntax:

```
#include <stdlib.h>
void exit(int status);
```

Key Characteristics:

- Terminates the program cleanly (flushes buffers, closes files)
- Returns control to the operating system
- Accepts an integer status code (convention: 0 = success, non-zero = error)

How `exit()` Differs from Regular Program Termination

Feature	**Normal Termination (`return` from `main()`)**	**`exit()`**
Where it works	Only from main() function	Anywhere
Cleanup	Yes	Yes
Order of operations	Runs all function returns	Immediate termination
Status code	Return value of main()	Argument to exit()

Basic return Examples

Example 1: Simple Function with Return

```c
#include <stdio.h>
// Function that adds two numbers and returns the result
int addNumbers(int a, int b) {
    int sum = a + b;
    return sum;  // Returns the sum back to the caller
}
int main() {
    int result = addNumbers(5, 3);
    printf("The sum is: %d\n", result);  // Output: The sum is: 8
    return 0;  // Indicates successful program execution
}
```

What happens:

1. main() calls addNumbers(5, 3)
2. addNumbers calculates 5 + 3 = 8
3. The return sum sends 8 back to main()
4. main() prints the result

Example 2: Early Return in a Function

```c
#include <stdio.h>
void checkAge(int age) {
    if (age < 0) {
        printf("Invalid age!\n");
        return;  // Exit function early if age is negative
    }

    if (age >= 18) {
        printf("You're an adult.\n");
    } else {
        printf("You're a minor.\n");
    }
}
int main() {
    checkAge(25);  // Output: You're an adult.
    checkAge(-5);  // Output: Invalid age!
    return 0;
}
```

Key points:

- The function exits immediately when it hits return
- For void functions, return doesn't need a value

Basic exit() Examples

Example 3: Simple Program Exit

```
#include <stdio.h>
#include <stdlib.h>  // Needed for exit()
int main() {
    printf("Program starting...\n");
    int number;
    printf("Enter a positive number: ");
    scanf("%d", &number);
    if (number <= 0) {
        printf("Error: Number must be positive!\n");
        exit(1);  // Exit program with error code 1
    }
    printf("You entered: %d\n", number);
    return 0;  // Normal exit
}
```

How it works:

- If user enters 0 or negative, program exits immediately
- The 1 in exit(1) is an error code (0 would mean success)
- No more code executes after exit()

Example 4: Exit from Inside a Function

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

void processFile(char* filename) {
    FILE *file = fopen(filename, "r");
    if (file == NULL) {
        printf("Error opening file!\n");
        exit(2);  // Exit entire program with code 2
    }

    // File processing code would go here
    fclose(file);
}
```

```
int main() {
    processFile("data.txt");
    printf("File processed successfully.\n");
    return 0;
}
```

What happens:

1. If "data.txt" doesn't exist, program exits immediately
2. The printf in main never runs if exit() is called
3. The error code 2 helps identify what went wrong

Practical Tips

1. **When to use return:**
    - When you want to send a value back from a function
    - For normal function completion
2. **When to use exit():**
    - When you encounter an unrecoverable error
    - When you need to stop the program immediately
    - In small programs where error handling would be overkill
3. **Exit codes:**
    - 0 means success
    - Non-zero means error (you can use different numbers for different errors)

Remember:

- return is for function control
- exit() is for program control

9.6 Recursion

A function is called recursive if it calls itself.
Recursion is the process of a function calling itself repeatedly till the given condition is satisfied.
A function that calls itself directly or indirectly is called a recursive function and such type/kind of function calls are called recursive calls.

```
int fact(int n)
{
```

```
    if(n <= 1)
        return 1;
    else
        return n * fact(n - 1);
}
```

Here's how the calls unfold when we compute fact(4):

1. **Initial Call: fact(4)**
    - 4 > 1 → return 4 * fact(3)
    - Must compute fact(3) first
2. **First Recursion: fact(3)**
    - 3 > 1 → return 3 * fact(2)
    - Must compute fact(2) first
3. **Second Recursion: fact(2)**
    - 2 > 1 → return 2 * fact(1)
    - Must compute fact(1) first
4. **Base Case: fact(1)**
    - 1 <= 1 → return 1 (recursion stops)
5. **Unwinding the Recursion:**
    - fact(2) gets 1 from fact(1) → returns 2 * 1 = 2
    - fact(3) gets 2 from fact(2) → returns 3 * 2 = 6
    - fact(4) gets 6 from fact(3) → returns 4 * 6 = 24

Call Stack Visualization:

Call Stack	Current n	Action
fact(4)	4	4 * fact(3)
fact(3)	3	3 * fact(2)
fact(2)	2	2 * fact(1)
fact(1)	1	returns 1
fact(2)	2	returns 2 * 1 = 2
fact(3)	3	returns 3 * 2 = 6
fact(4)	4	returns 4 * 6 = 24

Basic Structure of Recursive Functions :

```
type function_name (parameters)
{
  // function statements
  // base condition
  // recursion case (recursive call)
}
```

Example:

```
#include<stdio.h>
int nSum(int n)
{
    if (n == 0)
        return 0;
    int res = n + nSum(n - 1);
    return res;
}
int main()
{
    int n = 5;
    int sum = nSum(n);
    printf("Sum of first %d Natural Numbers : %d\n", n, sum);
    return 0;
}
```

Step-by-Step Calculation

1. **nSum(5)** calls 5 + nSum(4)
2. **nSum(4)** calls 4 + nSum(3)
3. **nSum(3)** calls 3 + nSum(2)
4. **nSum(2)** calls 2 + nSum(1)
5. **nSum(1)** calls 1 + nSum(0)
6. **nSum(0)** hits base case → returns 0
7. Now the stack "unwinds":
    - nSum(1) returns 1 + 0 = 1
    - nSum(2) returns 2 + 1 = 3

- nSum(3) returns 3 + 3 = 6
- nSum(4) returns 4 + 6 = 10
- nSum(5) returns 5 + 10 = 15

Key Characteristics

1. **Each recursive call** works on a smaller version of the problem (n-1)
2. **The call stack** grows until reaching the base case
3. **Results accumulate** as the stack unwinds
4. **Final result** is built from the combination of all partial results

Memory Visualization

Call Stack	n Value	Calculation	Result
nSum(5)	5	5 + nSum(4)	5 + 10 = 15
nSum(4)	4	4 + nSum(3)	4 + 6 = 10
nSum(3)	3	3 + nSum(2)	3 + 3 = 6
nSum(2)	2	2 + nSum(1)	2 + 1 = 3
nSum(1)	1	1 + nSum(0)	1 + 0 = 1
nSum(0)	0	base case	0

Memory Allocation for C recursive function :

- A stack frame is created on top of the existing stack frames each time a recursive call is encountered and data of each recursive copy of the function will be stored in their respective stack.
- Once, sum value is returned by the function, its stack frame will be destroyed.
- The compiler maintains an instruction pointer to store the address of the point where the control should return in the function after its progressive copy returns some value. This return point is the statement just after the recursive call.
- After all the recursive copy returns some value, we come back to the base function and then finally return the control to the caller function.

Example : Nth Fibonacci Sequence

```
#include<stdio.h>
int fibonacci(int n)
{
    if(n == 0)
       return 0;
    else if(n == 1)
       return 1;
    else
       return fibonacci(n - 1) + fibonacci(n - 2);
}
void printFibonacci(int n)
{
    for(int i = 0; i < n; i++)
    {
        printf("%d ", fibonacci(i));
    }
    printf("\n");
}
int main()
{
    int n;
    scanf("%d", &n);
    printFibonacci(n);
    return 0;
}
```

Chapter - 10 (Program Organization)

10.1 Local Variables

A variable declared in the body of a function is said to be local in that function.

```
int sumDigits(int n)
{
    int sum = 0; // sum is a local variable
    while(n > 0)
    {
        sum += n % 10;
        n /= 10;
    }
    return sum;
}
```

Properties of Local Variables in C

1. **Automatic Storage Duration**
    - Created when the function/block is entered
    - Destroyed when the function/block exits
    - Memory is managed automatically (on the stack)
    - Values are lost after the block ends
2. **Block Scope**
    - Only accessible within the block ({}) where declared
    - Cannot be accessed outside the block
    - Variables with the same name in outer blocks are **hidden** inside the inner block

```
void example() {
    int x = 10;  // Automatic storage, block scope (function-level)

    if (x > 5) {
        int y = 20;  // Only exists inside this block
        printf("%d", x + y);  // Valid
    }
    // y is NOT accessible here
```

```
}
```

Here the variable is declared in the function. So, it can be accessed with in the curly braces only.

Static Local Variables

static word before the variable declaration causes it to have static storage.

- For the static local variable a separated memory location is created.
- Has permanent storage throughout the execution of the program.
- If we know the memory location we can access it in the main function. (Pointers). Otherwise no because it is a local variable.

```
#include <stdio.h>
void counter() {
    static int count = 0;  // Static local variable
    count++;
    printf("Count: %d\n", count);
}
int main() {
    counter();  // Output: Count: 1
    counter();  // Output: Count: 2 (retains value)
    return 0;
}
```

Parameters in C Functions

Parameters (also called **formal parameters**) are variables declared in a function definition that receive values from **arguments** (actual values passed during a function call).

- Exist only within the function
- Behave like local variables

10.2 External Variables

External Variables are variables that are declared outside the body of any function. They are also called global variables.

Properties of External Variables

- Static Storage Duration
- File Scope

```
#include <stdio.h>

int globalCount = 0; // External variable (static storage, file scope)

void increment() {
    globalCount++; // Modifies the global variable
}

int main() {
    printf("Before: %d\n", globalCount); // 0
    increment();
    printf("After: %d\n", globalCount);  // 1
    return 0;
}
```

Pros

- **Persistent** – Retain values for entire program
- **Accessible** – Shared across functions/files
- **Convenient** – Avoids passing data repeatedly

Cons

- **Unsafe** – Any function can modify them
- **Less Modular** – Creates hidden dependencies
- **Hard to Debug** – Side effects in large programs

Best Practice: Use sparingly, prefer local variables and parameters.

Program : Guess The Number

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

#define MAX_NUMBER 100
```

```
int secret_number;

void initialize_number_generator();
void choose_new_secret_number();
void read_guesses();

int main()
{
    char command;
    printf("Guess the secret number between 1 and %d: ", MAX_NUMBER);
    initialize_number_generator();
    do
    {
        choose_new_secret_number();
        printf("A new number has been chosen.\n");
        read_guesses();
        printf("Play again? (Y/N): ");
        scanf(" %c", &command);
        printf("\n");
    } while (command == 'y' || command == 'Y');
    return 0;
}
void initialize_number_generator()
{
    srand((unsigned)time(NULL));
}
void choose_new_secret_number()
{
    secret_number = rand() % MAX_NUMBER + 1;
}
void read_guesses()
{
    int guess, num_guesses = 0;
    for (;;)
    {
        num_guesses++;
        printf("Enter guess : ");
        scanf("%d", &guess);
```

```
        if (guess == secret_number)
        {
            printf("You won in %d guesses! \n", num_guesses);
            return;
        }
        else if (guess < secret_number)
            printf("Too low; try again. \n");
        else
            printf("Too high; try again. \n");
    }
}
```

10.3 Blocks

A block (or compound statement) is a group of statements enclosed in curly braces {}. It defines a scope where variables can be declared and used.

Creates a New Scope

- Variables declared inside a block are **local** to it.
- They **cease to exist** when the block ends.

Used with Control Structures

- `if`, `for`, `while`, `switch`, etc., use blocks to group statements.

Variable Hiding

- Inner blocks can declare variables with the **same name** as outer ones (the inner variable **hides** the outer one).

```
int x = 5;
{
    int x = 10;  // Hides the outer 'x'
    printf("%d", x);  // Prints 10
}
printf("%d", x);  // Prints 5 (outer 'x' is unaffected)
```

Nested Blocks

- Blocks can contain other blocks, creating **nested scopes**.

10.4 Scope

Block Scope (Local Scope)

- Variables declared inside a **block** `{}` (e.g., inside functions, loops, or `if` statements).
- **Lifetime:** Exists only while the block executes.

Function Scope

- Applies only to **labels** (used with `goto`).

File Scope (Global Scope)

- Variables declared **outside all functions** (global variables).
- **Lifetime:** Entire program execution.

Function Prototype Scope

- Applies to **parameters in function declarations** (not definitions).

```
#include <stdio.h>
int global = 100;  // File scope
int main()
{
    int x = 10;    // Function scope (main)
    if (x > 5)
    {
        int y = 20;  // Block scope (inside if)
        printf("%d %d", x, y);  // Valid (x from outer scope)
    }
    // y is NOT accessible here
    printf("%d", global);  // Valid (global variable)
}
```

10.5 Organizing A Program

Preprocessing Directives such as #include #define
Type Definitions
Declaration of External Variables
Prototypes for functions other than main

Definition of main
Definition of other functions

Chapter - 11 (Pointers)

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

- It contains a memory address, not the actual data.
- The type of data it points to is specified when the pointer is declared.
- Pointers are powerful and enable features like dynamic memory allocation, arrays, and function pointers.

11.1 Pointer Variables

A pointer variable is specifically designed to store the memory address of another variable.

100	104	108	112	116
112	-1	107852331	100	108
a	b	c	d	e

Here 100, 104, 108, 112 and 116 are the memory address for the variables a, b, c, d and e respectively.
The declaration of the values are as follows :

```
int a = 112, b = -1;
float c = 3.14;
int *d = &a;
float *e = &c;
```

Declaring Pointer Variable

To declare a pointer variable, we need to specify the type of data it will point to, followed by an asterisk (*) and the pointer's name.
Syntax :

```
type *pointer_name;
```

type : The data type of the variable the pointer will point to. (reference type)
(*) : Indicates that the variable is of pointer type.
pointer_name : The name of the pointer variable.
Example :

```
int *ptr; // Declares a pointer to an integer
float *fptr; // Declares a pointer to a float
```

```
char *cptr; // Declares a pointer to a character
```

- int *ptr : The pointer ptr is designed to store the address of an integer variable
- float *fptr : The pointer fptr is designed to store the address of a float variable
- char *cptr : The pointer cptr is designed to store the address of a char variable

11.2 The Address and Indirection Operators :

To find the address of a variable we use the address of (&) operator. And to gain access to the object to which a pointer is pointing to we use the indirection/dereference operator.

The Address Operator (&)

The address of operator (&) is used to obtain the memory address of a variable.
It is used to retrieve the memory address of a variable.
Syntax :

```
pointerVariabe = &variable
// variable is the variable whose address is needed
// pointerVariable is a pointer that can store the address of the variable
```

Example :

```
int x, *p;
x = 10;
p = &x;
// &x gives the memory address of x
// p holds the address of x
```

The Indirection/Dereference Operator (*)

The indirection operator (*) is used to access or modify the value at the memory address stored in a pointer.
It accesses the value stored at the memory address a pointer holds.
Syntax :

```
value = *pointerVariable
// pointerVariable is a pointer storing the address of a variable
// *pointerVariable access the value stored at the address
```

Example :

```
int x = 10;
int *p = &x; // holds the address of x
printf("%d\n", *p); // Output : 10
```

Here *p accesses the value stored at the address p points to which is 10.

Example :

```
#include<stdio.h>
int main()
{
    int x = 42; // Regular Variable
    // Pointer 'ptr' stores the address of 'x'
    int *ptr = &x;
    // Using '&' to get the address of 'x'
    printf("Address of x : %p\n", &x);
    // Address stored in 'ptr'
    printf("Address stored in ptr : %p\n", ptr);
    // Dereferencing 'ptr'
    printf("Value of x using *ptr : %d\n", *ptr);
    // Changing the value of x using (*) via ptr
    *ptr = 100;
    printf("New value od x : %d\n", x);
    return 0;
}
```

Output :

```
Address of x : 0x7ffe151d00a4
Address stored in ptr : 0x7ffe151d00a4
Value of x using *ptr : 42
New value od x : 100
```

11.3 Pointer Assignment

Pointer assignment refers to assigning a value to a pointer variable.

1. Assigning the address of the variable to a pointer.
2. Assigning one pointer to another.
3. Assigning the result of dynamic memory allocation.

```
int x, y;
```

When we declare x, y integer type, they take memory space in the memory

2005	
2001	

```
int *p, *q;
```

When we declare *p, *q pointer type variable, they take memory space in the memory as well.

2013	
2009	
2005	
2001	

```
x = 7, y = 10;
p = &x, q = &y;
*p = 5;
```

2013	2005
2009	2001
2005	00001010
2001	00000111

- When 7 and 10 are assigned, binary forms of them are stored in the address.
- Here p will contain 2001 i.e the address of the first one.
- How many bits/bytes of RAM memory will be allocated to the pointer variable depends on the computer system.
- When we write x = 5; 5 is written in the memory of x in binary form.
- If we write &x; It says the address of the variable x.
- So, p = &x; stores the address of x in the memory of p i.e. 2001.
- Similarly for q = &y; stores the address of y in the memory of q i.e. 2005.
- Here p points to x and q points to y.
- If we write *p = 6; It indirects to the variable x. So *p will help to access the memory of x and the value of x will be updated.

Program : Swap numbers

```
#include<stdio.h>
int main()
{
    int a = 5, b = 6;
    int *aptr = &a, *bptr = &b;
    printf("Before swapping a = %d, b = %d\n", a, b);
    int temp = *aptr;
    *aptr = *bptr;
    *bptr = temp;
    printf("After swapping a = %d, b = %d\n", a, b);
    return 0;
}
```

Output :

```
Before swapping a = 5, b = 6
After swapping a = 6, b = 5
```

Undefined Behaviors

```
int *r;
printf("%d", *r);
```

When r is declared the memory is blocked and will contain some garbage value.
Some garbage value that is there if it points to an actual address then will print the value stored in the address, otherwise if it does not exist then runtime error. (Program crashes).

```
*r = 1; // will show undefined behaviour
```

We can define a pointer variable. But if we do not initialize the pointer variable to a valid memory address then accessing the address may lead to an undefined behaviour.

Assignments :

```
int x, y;
int *p, *q;
x = 7, y = 10;
p = &x, q = &u;
q = p; // now both are pointing to x
```

Here both the pointer q and p are now pointing to the same memory location i.e. the memory address of the variable x. And by using the indirection operator we will get the same value that is stored in the memory for x for both the pointers.

```
int x, y;
int *p, *q;
x = 7;
p = &x, q = &y;
*q = *p; // 7 is stored in y now
```

The line *q = *p; copies the value 7 from x (accessed via *p) into y (accessed via *q). This uses pointer dereferencing to indirectly assign y = x without directly referencing the variables.

Example : This is for output prediction

```
#include<stdio.h>
int main()
{
    int u = 3;
    int v;
    int *pu; // Pointer to an integer
    int *pv; // Pointer to an integer
    pu = &u; // Assign address of u to pu
    v = *pu; // Assign value of u to v
    pv = &v; // Assign address of v to pv
    printf("u = %d &u = %p pu = %p *pu = %d\n", u, &u, pu, *pu);
    printf("v = %d &v = %p pv = %p *pv = %d\n", v, &v, pv, *pv);
    return 0;
}
```

Output :

```
u = 3 &u = 0x7ffde48dcdfc pu = 0x7ffde48dcdfc *pu = 3
v = 3 &v = 0x7ffde48dcdf8 pv = 0x7ffde48dcdf8 *pv = 3
```

Example :

```
#include<stdio.h>
int main()
{
    int u1, u2;
    int v = 3;
```

```
    int *pv; // pv points to v
    u1 = 2 *(v + 5); // ordinary expression
    pv = & v;
    u2 = 2 * (*pv + 5); // equivalent expression
    printf("u1 = %d u2 = %d\n", u1, u2);
    return 0;
}
```

Output :

```
u1 = 16 u2 = 16
```

Example :

```
#include<stdio.h>
int main()
{
    int v = 3;
    int *pv;
    pv = &v; // pv points to v
    printf("*pv = %d  v = %d\n", *pv, v);
    *pv = 0; // reset v indirectly
    printf("*pv = %d  v = %d\n", *pv, v);
    return 0;
}
```

Output :

```
*pv = 3  v = 3
*pv = 0  v = 0
```

11.4 Pointers as Arguments

Pointers as arguments are used to pass the memory address of a variable to a function. This is commonly known as call by reference.

1. **To modify the original variable :** Since the function works with the actual memory address, changes made in the function are reflected in the original variable.
2. **To improve efficiency :** Instead of copying large structures or arrays, passing a pointer avoids overhead and improves performance.
3. **Dynamic memory handling :** Allows the function to allocate and manage memory dynamically.

Example :

```
#include<stdio.h>
void decompose(double val, int *real, double *fract)
{
    *real = (int)val;
    *fract = val - *real;
}
int main()
{
    double pi = 3.1416;
    int r;
    double f;
    decompose(pi, &r, &f);
    printf("%d\n", r);
    printf("%lf\n", f);
    return 0;
}
```

Output :

```
3
0.141600
```

Example :

```
#include<stdio.h>
void updateValue(int *ptr)
{
    *ptr = 100; // Modify the value at the address
}
int main()
{
   int num = 10;
   printf("Before : %d\n", num);
   updateValue(&num); // Pass the address of num
   printf("After : %d\n", num);
   return 0;
}
```

Output :

```
Before : 10
After : 100
```

Example :

```
#include<stdio.h>
void function1(int u, int v); // function prototype
void function2(int *pu, int *pv); // function protptype
int main()
{
    int u = 1;
    int v = 3;
    printf("Before calling function1 : u = %d  v = %d\n", u, v);
    function1(u, v);
    printf("After calling function1 : u = %d  v = %d\n", u, v);
    printf("Before calling function2 : u = %d  v = %d\n", u, v);
    function2(&u, &v);
    printf("After calling function2 : u = %d  v = %d\n", u, v);
    return 0;
}
void function1(int u, int v)
{
    u = 0;
    v = 0;
    printf("Within function1 : u = %d  v = %d\n", u, v);
}
void function2(int *pu, int *pv)
{
    *pu = 0;
    *pv = 0;
    printf("Within function2 : *pu = %d  *pv = %d\n", *pu, *pv);
}
```

Output :

```
Before calling function1 : u = 1  v = 3
Within function1 : u = 0  v = 0
After calling function1 : u = 1  v = 3
Before calling function2 : u = 1  v = 3
Within function2 : *pu = 0  *pv = 0
After calling function2 : u = 0  v = 0
```

Program : Finding the largest and smallest elements in an Array

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

```
#define N 10
void max_min(int a[], int n, int *max, int *min);
int main()
{
    int a[N], max, min;
    printf("Enter %d numbers : ", N);
    for(int i = 0; i < N; i++)
    {
        scanf("%d", &a[i]);
    }
    max_min(a, N, &max, &min);
    printf("Largest : %d\n", max);
    printf("Smallest : %d\n", min);
    return 0;
}
void max_min(int a[], int n, int *max, int *min)
{
    *max = *min = a[0];
    for(int i =0; i < n; i++)
    {
        if(a[i] > *max)
           *max = a[i];
        else if(a[i] < *min)
           *min = a[i];
    }
}
```

Using const in Pointers :

The use of const in pointers can control whether the pointer itself, the data it points to, or both are protected from modification.

- **Pointer to constant data :**
  `const int *ptr = &y;`
  The data being pointed to by ptr cannot be modified through the pointer. However, the pointer ptr itself can be reassigned to point to another memory address.
  `ptr = &x; // allowed`
  `*ptr = 5; // error`
- **Constant Pointer :**
  `int *const ptr;`

The pointer ptr itself cannot be reassigned to point to another memory address. However, the data being pointed to by ptr can be modified through the pointer.

```
ptr = &x; // error
*ptr = 6; // allowed
```

- **Constant Pointer to Constant Data :**

```
const int *cont ptr;
```

The pointer ptr itself cannot be reassigned to point to another memory address. The data being pointed to by ptr cannot be modified through the pointer.

Example :

```
#include <stdio.h>
int main() {
    int x = 10, y = 20;

    // 1. Pointer to constant data
    const int *ptr1 = &y;
    ptr1 = &x;    // Allowed - pointer can change
    // *ptr1 = 5;  // Error - data cannot be modified

    // 2. Constant pointer
    int *const ptr2 = &x;
    *ptr2 = 15;    // Allowed - data can be modified
    // ptr2 = &y;  // Error - pointer cannot change

    // 3. Constant pointer to constant data
    const int *const ptr3 = &x;
    // ptr3 = &y;  // Error - pointer cannot change
    // *ptr3 = 20; // Error - data cannot be modified

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

Using const to Protect Argument :

In C programming, we can use the const keyword to protect function arguments by making them read-only. This ensures that the function cannot modify the value of the argument.

It is useful when you want to pass a value by references but ensures that the original value is not altered.

1. Prevent accidental changes

2. Self Documentation
3. Optimizations

Example :

```
#include <stdio.h>
// 1. Protect value from modification (pass by value)
void printValue(const int num) {
    // num = 5;  // Error - cannot modify const argument
    printf("Value: %d\n", num);
}
// 2. Protect pointer data from modification (pass by reference)
void printString(const char *str) {
    // str[0] = 'X';  // Error - cannot modify const data
    printf("String: %s\n", str);
}
// 3. Protect pointer itself from modification
void printArray(const int *const arr, int size) {
    // arr = NULL;    // Error - cannot modify const pointer
    // arr[0] = 99;   // Error - cannot modify const data
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
int main() {
    int x = 10;
    char msg[] = "Hello";
    int nums[] = {1, 2, 3};
    printValue(x);      // Value won't be modified
    printString(msg);   // String won't be modified
    printArray(nums, 3); // Array won't be modified
    return 0;
}
```

11.5 Pointers as Return Values :

Syntax :

```
type *function_name(parameters)
{
  return pointer_variable;
```

}

- **Local variables :** Local variables go out of scope after the function returns, making their address invalid. So, avoid returning local variables.
- **Global variables :** Safe to use, but they can lead to unintended side effects if modified elsewhere.
- **Dynamic Memory :** Offers flexibility but requires proper memory management.
- **Static variables :** Can be used if you want to return variables with a lifetime that extends beyond the function.

Example :

```
#include <stdio.h>
// 1. Returning pointer to global variable (safe but has side effects)
int global_var = 10;
int* get_global()
{
    return &global_var; // Safe - global exists forever
}
// 2. Returning pointer to static local variable (safe)
int* get_static()
{
    static int static_var = 20; // Persists between calls
    return &static_var;
}
// 3. Dangerous: Returning pointer to local variable (WRONG)
int* dangerous()
 {
    int local = 30;
    return &local; // COMPILER WARNING - address becomes invalid
}
// 4. Safe: Returning pointer to parameter (if parameter persists)
int* process_value(int* input)
{
    *input += 5;
    return input; // Safe if input points to valid memory
}
int main()
{
    int x = 100;
    // Safe uses:
```

```
    int *p1 = get_global();
    int *p2 = get_static();
    int *p3 = process_value(&x);
    printf("Global: %d\n", *p1);  // 10
    printf("Static: %d\n", *p2);  // 20
    printf("Param:  %d\n", *p3);  // 105
    // Dangerous:
    int *p4 = dangerous(); // Undefined behavior!
    // printf("%d", *p4);  // Might crash or show garbage
    return 0;
}
```

Chapter - 12 (Pointers & Arrays)

12.1 Pointer Arithmetic

Refers to operations performed on pointers in programming language. There operations allow pointers to traverse and manipulate memory by leveraging their ability to store addresses.
C supports three forms of pointer arithmetic :

- Adding an integer to a pointer
- Subtracting an integer from a pointer
- Subtracting one pointer from another
- Comparing Pointers

Adding an integer to a pointer

Adds an integer n to a pointer, moving it forward by n elements.

```
int a[] = {1, 2, 5, 9, 4, 6}
```

1021	6
1017	4
1013	9
1009	5
1005	2
1001	1

```
int *p = &a[0]; // stores 1001
p = p + 1; // jumps to the next memory address i.e. 1005
int x = *p; // stores 2 now
```

When we add 1 to p it jumps to the address of the next memory. At first p = 1001 adding 1 p becomes 1005. Now writing p = p -1 will again take me to 1001 memory addresses. By adding 5 we will get p = 1021 now.
For an int *ptr, ptr+1 moves the pointer forward by sizeof(int) bytes.

Subtracting an Integer from a Pointer

Subtracts an integer n from a pointer, moving it backward by n elements.

For an int *ptr, ptr-1 moves the pointer backward by sizeof(int) bytes.

Subtracting one pointer from another

Calculates the number of elements between two pointers pointing to the same array or memory block.

```
#include <stdio.h>
int main() {
    int numbers[] = {10, 20, 30, 40, 50};
    int *ptr1 = &numbers[1];  // Points to 20 (2nd element)
    int *ptr2 = &numbers[4];  // Points to 50 (5th element)
    // Calculate number of elements between pointers
    ptrdiff_t diff = ptr2 - ptr1;
    printf("Pointer difference: %td elements\n", diff);
    printf("This means there are %td elements between %d and %d\n",
           diff - 1, *ptr1, *ptr2);
    return 0;
}
```

If ptr2 points to arr[4] and pt1 points to arr[1], ptr2 - ptr1 yields 3.

Comparing Pointers :

Compare two pointers to determine their relative position in memory, assuming they point to the same array or memory block.
p < q, p <= q, p >= q, p >q, p != q, p == q can be done.

Note : Array name also works as a pointer so pointer arithmetic can be done to array name.
The array name is a constant pointer.

```
int *const p;
*(a + 2); // accessing element of index - 2
a = a + 1; // wrong as a is a constant pointer
a++; // wrong as a is a constant pointer
```

In pointer ++, - -, +=, -= can also be used but this cannot be done in case of pointer array i.e. array name as pointer as it is constant (compilation error).

Compound Literals

Pointers to compound literals refer to pointers that point to compound literal objects, which are temporary objects created using compound literals.
A compound literal is a way to create unnamed objects (such as arrays or structures) that can be accessed immediately.

Syntax :

```
(type){initializer_list};
```

Example :

```
#include<stdio.h>
int main()
{
    // compound literal of an array
    int *ptr = (int[]){1, 2, 3, 4, 5};
    printf("%d\n", ptr[0]); // output : 1
    printf("%d\n", ptr[2]); // output : 3
    return 0;
}
```

(int) is typecasting

Example :

```
#include<stdio.h>
void printArray(int *arr, int size)
{
    for(int i = 0; i < size; i++)
    {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
int main()
{
    printArray((int[]){1, 2, 3, 4, 5}, 5);
    return 0;
}
```

12.2 Using Pointer for Array Processing

Array & Pointer :

```
int a[] = {1, 2, 5, 9, 4, 6};
int *p = a;
```

For array we can omit the address operator because ‘a’ in here is the name of a constant pointer. And p points to the first element of the array.

```
int arr[10];
for(int *p = a; p < a + 10; p++)
{
    scanf("%d", p);
}
```

Here p is the address of the array so no & needed as &arr[0] means p. And here by p++ it keeps updating and it jumps places.

- In earlier days using pointers was more efficient as less memory required.
- Using pointers we can process arrays.
- We can also use the array name as a pointer but the array name is a constant pointer.

Pointer arithmetic allows us to visit the element of an array by repeatedly incrementing a pointer variable

Example :

```
#include<stdio.h>
#define N 10
int main()
{
    int a[N], sum, *p;
    a[N] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    sum = 0;
    for(p = &a[0]; p < &a[N]; p++)
    {
        sum += *p;
    }
    printf("%d", sum);
}
```

Combining the * and ++ operators :

The postfix version of ++ takes precedence over *, the compiler sees it as *(p++).

- *p++ or, *(p++) = value of expression *p before increment; increment p later
- (*p)++ = value of expression is *p before increment; increment p later
- *++p or, *(++p) = Increment p first; value of expression is *p after increment
- ++*p pr, ++(*p) = Increment *p first; value of expression is *p after increment

Example :

```
#include <stdio.h>
int main() {
    int nums[] = {10, 20, 30, 40};
    int *p = nums;
    printf("Initial array: ");
    for (int i = 0; i < 4; i++) {
        printf("%d ", nums[i]);
    }
    // 1. *p++ (post-increment pointer)
    printf("\n*p++ : %d", *p++); // Prints 10 (nums[0]), then moves to
nums[1]
    printf("\nNow *p : %d", *p);  // Now shows 20 (nums[1])
    // 2. (*p)++ (post-increment value)
    printf("\n(*p)++ : %d", (*p)++); // Prints 20, then increments
nums[1] to 21
    printf("\nNow *p : %d", *p);     // Now shows 21
    // 3. *++p (pre-increment pointer)
    printf("\n*++p : %d", *++p); // Moves to nums[2] first, then
prints 30
    printf("\nNow *p : %d", *p); // Still shows 30 (nums[2])
    // 4. ++*p (pre-increment value)
    printf("\n++*p : %d", ++*p); // Increments nums[2] to 31, then
prints 31
    printf("\nNow *p : %d", *p); // Now shows 31
    printf("\nFinal array: ");
    for (int i = 0; i < 4; i++) {
        printf("%d ", nums[i]); // Shows modified values
    }
    return 0;
}
```

12.3 Using an Array Name as Pointer

The name of an array can be used as a pointer to thr first element in the array.

```
int a[10];
*a = 7; // stores 7 in a[0]
*(a + 1) = 12; // stores 12 in a[1]
```

```
while(N--)
{
    printf("%d", *(arr + N));
}
```

Array name works as a constant pointer.
As arr is a constant pointer we cannot update the pointer ass but change the value within the pointer.

```
arr += 1; // error
```

Program : Reversing a Series of Numbers

```
#include<stdio.h>
#define N 10
int main()
{
    int a[N], *p;
    printf("Enter %d numbers : ", N);
    for(p = a; p < a + N; p++)
    {
        scanf("%d", p);
    }
    printf("In reverse order : ");
    for(p = a + N - 1; p >= a; p--)
    {
        printf("%d ", *p);
    }
    printf("\n");
    return 0;
}
```

Array Arguments (Revisited)

When passed to a function, an array name is always treated as a pointer.

```
int find_largest(int a[], int n)
{
    int i, max;
    max = a[0];
    for(i = 1; i < n; i++)
    {
        if(max < a[i])
           max = a[i];
    }
    return max;
}
```

Suppose we call find_largest:

```
largest = find_largest(b, N);
```

Array Decay to Pointer

- When passing an array to a function, C **automatically converts it to a pointer** to the first element.
- Inside find_largest(), int a[] is treated as int *a.

Equivalent Function Declarations

```
int find_largest(int a[], int n);  // Array-style (recommended for clarity)
int find_largest(int *a, int n);   // Pointer-style (same behavior)
```

Both forms work identically.

Behavior in find_largest(b, N)

- b decays to &b[0] (address of first element).
- The function accesses b's elements via pointer arithmetic (a[i] ≡ *(a + i)).

Using a Pointer as an Array Name

A pointer can be used as if it were an array name because pointers and arrays share some similarities in how they manage memory. When a pointer points to a block of memory (such as an array), the pointer can be used to access elements in that memory block using the array subscript notation (ptr[index])

Example :

```
#include<stdio.h>
int main()
{
    int arr[] = {10, 20, 30, 40, 50};
    int *ptr = arr;
    // pointer points to the first element of arr
    printf("%d\n", ptr[0]); // output : 10
    printf("%d\n", ptr[1]); // output : 20
    printf("%d\n", ptr[2]); // output : 30
    ptr[1] = 25;
    printf("%d\n", ptr[1]);
    return 0;
}
```

12.4 Pointers And Multidimensional Arrays

In C, multidimensional arrays are stored in row-major order (contiguous memory), and pointers can be used to navigate them efficiently. However, there are key differences between array notation and pointer arithmetic.

```
int arr[rows][cols];
int *p = &arr[0][0];
for(; p < arr[0] + (rows * cols); p++)
{
    printf("%d ", *p);
}
```

Here &arr[0][0] = arr[0]. The elements in an array (also in a multidimensional array) are stored in a contiguous manner. So all memory addresses will be contiguous.

0 0	0 1	0 2	1 0	1 1	1 2	2 0	2 1	2 2

- arr[0][0] is the first number and it is a double pointer
- arr[0] is a pointer to the first row and is a pointer to an array
- Arr[1] is a pointer to the second row and is a pointer to an array

```
find_largest(arr[0], cols); // processes a single row
```

```
find_largest(&arr[0][0], rows*cols); // processes the whole matrix
```

arr -> &arr[0]
(arr + 1) -> &arr[1]
*(arr[0] + 1) -> accessing 2nd element of the first row
*(arr[1] + 1) -> accessing 2nd elements of the second row

Processing All Elements (Element-Wise)

```
int arr[3][4] = {{1,2,3,4}, {5,6,7,8}, {9,10,11,12}};

// Method 1: Nested loops (recommended)
for(int i=0; i<3; i++) {
    for(int j=0; j<4; j++) {
        printf("%d ", arr[i][j]); // Process each element
    }
}

// Method 2: Single pointer (linear access)
int *ptr = &arr[0][0];
for(int k=0; k<3*4; k++) {
    printf("%d ", *(ptr + k)); // Treat as 1D array
}
```

Processing Row by Row

```
// Print each row's sum
for(int i=0; i<3; i++) {
    int row_sum = 0;
    for(int j=0; j<4; j++) {
        row_sum += arr[i][j];
    }
    printf("Row %d sum: %d\n", i, row_sum);
}

// Using pointer to row
int (*row_ptr)[4] = arr; // Pointer to first row
for(int i=0; i<3; i++) {
    printf("Row %d: ", i);
    for(int j=0; j<4; j++) {
```

```
        printf("%d ", row_ptr[i][j]); // Or *(*(row_ptr+i)+j)
    }
    printf("\n");
}
```

Processing Column by Column

```
// Print each column's sum
for(int j=0; j<4; j++) {
    int col_sum = 0;
    for(int i=0; i<3; i++) {
        col_sum += arr[i][j]; // Note: i and j swapped
    }
    printf("Column %d sum: %d\n", j, col_sum);
}

// Using pointer arithmetic (column-wise)
for(int j=0; j<4; j++) {
    printf("Column %d: ", j);
    for(int i=0; i<3; i++) {
        printf("%d ", *(arr[i] + j)); // arr[i][j] alternative
    }
    printf("\n");
}
```

2D Array :

For int arr[8][6];
Type(arr) : int(*)[6] array of 6 integer
Type(arr[0]) : int * points to the first element of the first row
arr : arr[0], arr points to arr[0]
(arr + 1) : arr[1]
(arr + 2) : arr[2]
(arr[0] + 1) : arr[0][1], points to the 2nd int of 1st row
To access through dereference of arr we need double dereference as it is double pointers.

3D Array :

Int b[3][4][5];

Type(b) : int (*) [4][5]; points to a 2 dimensional array
Type(b[0]) : int(*)[5]; points to an array
Type(b[0][0]) : int(*)
For (b[0]) dereferencing **b[0]
For (b) dereferencing ***b
Type(b[0][0][0]) : int ; pointing to the first element
(b + 1) : b[1]
(*(b + 1) + 1) : b[1][1] ; Dereferencing to the second block and adding 1 will dereference to the second row.
fun(int a[][4][5]) ; pass a pointer to a 2D array i.e. we can pass b in the function as argument.
fun(int ***a) ; for 3D array
fun(int **a); for 2D array
fun(int *a); for 1D array

*p[5] : array of pointer
(*p)[5] : pointer to an array

```
for(p = &arr[0][2]; p <= &arr[3][2]; p++) {
    printf("%d", **p);
}
```

1. **Pointer Setup:**
    - p is a pointer to an array of 5 integers
    - Starts at address of arr[0][2] (3rd element of first row)
2. **Loop Logic:**
    - Moves through rows (p++ jumps by entire rows)
    - Stops when reaching arr[3][2] (3rd element of last row)
3. **Output:**
    - **p dereferences to get the first element of each pointed row
    - Prints: arr[0][2], arr[1][2], arr[2][2], arr[3][2]

```
for(p = &arr[0][0]; p <= &arr[3][2]; p++) {
    printf("%d", (*p)[2]);
}
```

1. **Pointer Setup**:
    - Starts at `arr[0][0]` (first element)
2. **Loop Logic**:
    - Still moves by rows (`p++` jumps 5 elements)

- Stops at `arr[3][2]`

3. **Output**:
   - `(*p)[2]` accesses the 3rd element of each row
   - Prints: `arr[0][2]`, `arr[1][2]`, `arr[2][2]`, `arr[3][2]`

Example :

```
#include <stdio.h>
int main() {
    // Initialize a 4x5 array with sequential values
    int arr[4][5] = {
        {1, 2, 3, 4, 5},
        {6, 7, 8, 9, 10},
        {11, 12, 13, 14, 15},
        {16, 17, 18, 19, 20}
    };
    int (*p)[5]; // Pointer to array of 5 ints
    printf("First loop output (using **p):\n");
    for(p = &arr[0][2]; p <= &arr[3][2]; p++) {
        printf("%d ", **p);
    }
    printf("\n\nSecond loop output (using (*p)[2]):\n");
    for(p = &arr[0][0]; p <= &arr[3][2]; p++) {
        printf("%d ", (*p)[2]);
    }
    return 0;
}
```

Output :

```
First loop output (using **p):
3 8 13 18

Second loop output (using (*p)[2]):
3 8 13 18
```

12.5 Pointers And VLA

A pointer to a VLA maintains information about the array's dimensions, enabling proper pointer arithmetic for multidimensional arrays.

Syntax :

```
int rows = 3, cols = 4;
int (*vla_ptr)[cols];  // Pointer to VLA with 'cols' columns
```

Example :

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

void process_VLA(int rows, int cols, int (*arr)[cols]) {
    // Access elements using array notation
    for(int i = 0; i < rows; i++) {
        for(int j = 0; j < cols; j++) {
            printf("%d ", arr[i][j]);
        }
        printf("\n");
    }

    // Pointer arithmetic works correctly
    printf("\nThird element of second row: %d\n", *(*(arr + 1) + 2));
}

int main() {
    int rows, cols;
    printf("Enter rows and columns: ");
    scanf("%d %d", &rows, &cols);

    // Create VLA
    int vla[rows][cols];

    // Initialize with sample values
    for(int i = 0; i < rows; i++) {
        for(int j = 0; j < cols; j++) {
            vla[i][j] = i * cols + j + 1;
        }
    }

    // Create pointer to VLA
    int (*vla_ptr)[cols] = vla;

    // Pass to function
```

```
    process_VLA(rows, cols, vla_ptr);

    return 0;
}
```

Output :

Enter rows and columns: 3 4
1 2 3 4
5 6 7 8
9 10 11 12

Third element of second row: 7

Chapter - 13 (Strings)

In C, a string is an array of characters terminated by a null character.

13.1 String Literals

A string literal is a sequence of characters enclosed within a double quotation.

```
"String Literal"
```

S	t	r	i	n	g		L	i	t	e	r	a	l	\0

In memory (RAM) the string is stored contiguously and at the end a null termination is added.
Null character = \0
The null character marks the end of the string and it stops there.
If there is no null character then it is considered as a character array.

We cannot modify a string literal :

```
char *p = "abc";  // p points to a string literal "abc" (stored in
read-only memory)
p = "def";        // p now points to a different string literal "def"
```

Escape Sequence in String Literal :

String literals may contain the same sequences as character constants.

```
"This is my\ntake on the strings\nin C Program\n"
```

Output :

```
This is my
take on the strings
in C Program
```

Although octal and hexadecimal character sequences are also legal string literals, they are not commonly used.

Continuing a String Literal :

String Splicing Rule in C

1. Adjacent string literals are automatically joined at compile time.
2. Whitespace is not automatically inserted between them.
3. Works across multiple lines, making long strings more readable.

In C, when you write:

```
printf("This is line one " "This is line two");
```

The compiler automatically concatenates adjacent string literals, so this is equivalent to:

```
printf("This is line one This is line two");
```

How string literals are stored

C treats strings as a character array. When a C compiler encounters a string character of N number of characters, it sets aside N+1 bytes of memory space for the string. This area of memory will contain the characters in string, plus an extra character the null character / terminator to mark the end of the string.
Null Character : The null character is a byte whos bits are all 0. So, it is represented by \0 in the escape sequence.

Operations on String Literals :

C allows a char pointer to store a string.

```
#include <stdio.h>
int main() {
    char *str = "Hello, World!";
    printf("%s\n", str);  // %s is used for strings
    return 0;
}
```

%s when we pass a variable 'a' it directs to its address. As a result we do not need to use & operator.

This does not copy the character “Hello, World!”; it merely makes p point to the first character of the string.

String Literals vs Character Constants :

A string literal containing a single character isn't the same as a character constant.
The string literal "a" is represented by a pointer to a memory location that stores the character a followed by a null character. The character constant 'a' is represented as an integer i.e. the numerical code for the character.

```
char ch1[] = "H";
char ch2 = 'H';
```

They are stores in memory like:

Address	Value (hex)	Value (ASCII)	Purpose
0x1000	0x48	'H'	First character
0x1001	0x00	'\0'	Null terminator (end of string)

Address	Value (hex)	Value (ASCII)	Purpose
0x1002	0x48	'H'	Single character (ASCII value)

13.2 String Variables

In C, a string variable is a variable that holds a sequence of characters (a string). Any one dimensional array of characters can be used to store a string, with the understanding that a string is terminated by a null terminator / null character.
If we want to store a string of N characters in a string variable then we can declare an array of N+1 characters as an extra space for the null terminator. Otherwise it will simply be treated as a character array.

Initializing a string variable :

```
char str1[10] = "String Var"; // treated as a char array
char str2[10+1] = "String Var"; // treated as a string
```

Here in str1 a null character isn't stored. So, it is treated as a character array. So, for considering it as a string the array size needed to be set with 1 extra space, which was done in str2. So, str2 is treated as a string.
If the array is initialized with more than N+1 values than all the other memory spaces will be filled with null characters.

```
char str1[10] = "String Var";
char str2[11] = "String Var";
char str3[13] = "String Var";
```

Draw the diagram when this appears in the exam

str1

S	t	r	i	n	g		V	a	r

str2

S	t	r	i	n	g		V	a	r	\0

str3

S	t	r	i	n	g		V	a	r	\0	\0	\0

In the declaration of a string the variable length may be omitted in that case the computer computes as a string and sets N+1 memory spaces for the string variable when initialized.

```
char str[] = "String Var";
```

The compiler treats this as a string and sets 11 memory spaces for the string (characters + null character).

Character Array Vs Character Pointer :

```
char date[] = "June 14";
char *date = "June 14";
```

- In the array version the character stored in date can be modified, like the elements in an array.
  In the pointer version, date points to a string literal and a string literal cannot be modified.
- In the array version date is an array name.
  In the pointer version, date is a variable that can be made to be pointed to any other string during program execution.

Pointer of a constant variable

```
char str[STR_LEN + 1], *p;
p = str;
```

Before we can use p it must point to an array of characters. Here p points to the first element of the array.

13.3 Reading and Writing Strings

Writing Strings

We can write strings using printf() and puts().

Using printf()

The printf() function is used to display formatted output. It can print strings using the %s format specifier.

Syntax:

```
printf("format string", arguments);
```

Example:

```
#include <stdio.h>
int main() {
    char str[] = "Hello, World!";
    printf("%s\n", str); // Output: Hello, World!
    return 0;
}
```

Explanation:

- %s is used to print a string.
- \n is used to move to a new line.

Using puts()

The puts() function prints a string followed by a newline.

Syntax:

```
puts(string);
```

Example:

```c
#include <stdio.h>
int main() {
    char str[] = "Hello, World!";
    puts(str); // Output: Hello, World!
    return 0;
}
```

Differences between printf() and puts()

Feature	printf("%s")	puts()
Newline	Not added automatically	Adds newline automatically
Return Value	Number of characters printed	Non-negative value on success
Usage	General-purpose formatted output	Simple string output

Reading Strings

Strings can be read using scanf() and gets().

Using scanf()

The scanf() function reads a string using the %s format specifier.

Syntax:

```c
scanf("%s", string);
```

Example:

```c
#include <stdio.h>

int main() {
    char name[50];
    printf("Enter your name: ");
    scanf("%s", name);
    printf("Hello, %s\n", name);
    return 0;
}
```

Limitations:

- scanf() stops reading input at the first whitespace (space, tab, newline).
- It may cause buffer overflow if input exceeds the allocated memory.

Solution: Use scanf("%49s", name); (for a 50-character array) to prevent overflow.

Using gets()

The gets() function reads a full line of input, including spaces.

Syntax:

```
gets(string);
```

Example:

```
#include <stdio.h>
int main() {
    char name[50];
    printf("Enter your full name: ");
    gets(name);
    printf("Hello, %s\n", name);
    return 0;
}
```

Warning: gets() is dangerous because it doesn't check buffer size, leading to buffer overflow. It has been removed from C11.

Alternative: Use fgets() instead:

```
fgets(name, sizeof(name), stdin);
```

Reading Strings Character by Character

Instead of reading a full string, we can read it character by character using getchar() or a loop.

Using getchar()

The getchar() function reads a single character from input.

Example:

```c
#include <stdio.h>

int main() {
    char ch;
    printf("Enter a character: ");
    ch = getchar(); // Reads one character
    printf("You entered: %c\n", ch);
    return 0;
}
```

Using a Loop (getchar() with while)

We can use a loop to read an entire string character by character.

Example:

```c
#include <stdio.h>

int main() {
    char str[100];
    char ch;
    int i = 0;

    printf("Enter a string: ");
    while ((ch = getchar()) != '\n' && i < 99) {
        str[i++] = ch;
    }
    str[i] = '\0'; // Null-terminate the string

    printf("You entered: %s\n", str);
    return 0;
}
```

Explanation:

- getchar() reads one character at a time.
- The loop continues until a newline (\n) is encountered.
- We manually add \0 to terminate the string.

13.4 Accessing Characters in a String in C

Accessing Characters Using Array Indexing

Since a string is an array of characters, we can access its characters using their index.

Example:

```
#include <stdio.h>
int main() {
    char str[] = "Hello, World!";
    printf("First character: %c\n", str[0]); // H
    printf("Fifth character: %c\n", str[4]); // o
    printf("Last character: %c\n", str[12]); // d
    return 0;
}
```

Explanation:

- str[0] accesses 'H'
- str[4] accesses 'o'
- str[12] accesses 'd' (Indexing starts from 0)
- The string ends with a null character (\0), which isn't printed.

Accessing Characters Using Pointer Arithmetic

We can also access characters using pointers.

Example:

```
#include <stdio.h>

int main() {

    char str[] = "Hello";

    char *ptr = str; // Pointer to first character

    printf("First character: %c\n", *ptr);      // H

    printf("Second character: %c\n", *(ptr+1)); // e
```

```
    printf("Third character: %c\n", *(ptr+2));  // l

    return 0;

}
```

Explanation:

- ptr points to str[0] ('H').
- *(ptr + 1) gives str[1] ('e').
- *(ptr + 2) gives str[2] ('l').

Counting Spaces in a String

We can count spaces by iterating through the string and checking for ' '.

Example:

```
#include <stdio.h>

int main() {

    char str[] = "Hello World, how are you?";

    int i, count = 0;

    for (i = 0; str[i] != '\0'; i++) {

        if (str[i] == ' ') {

            count++;

        }

    }

    printf("Number of spaces: %d\n", count);

    return 0;

}
```

Explanation:

- The loop runs until '\0' (end of the string).
- If str[i] is a space (' '), count is incremented.
- Finally, the number of spaces is printed.

13.5 String Operations in C

The C String Library (`#include <string.h>`) provides several functions for manipulating strings. Here, we will discuss copying, comparing, concatenating, and selecting a substring, along with their manual implementations.

Copying a String (`strcpy()`)

The `strcpy()` function copies the contents of one string to another.

Using Library Function

```
#include <stdio.h>

#include <string.h>

int main() {

    char source[] = "Hello, World!";

    char destination[50];

    strcpy(destination, source); // Copy source to destination

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

    return 0;

}
```

Manual Implementation (`my_strcpy()`)

```
#include <stdio.h>

void my_strcpy(char *dest, const char *src) {

    while (*src) {  // Copy until null terminator
```

```
        *dest = *src;
        dest++;
        src++;
    }
    *dest = '\0';  // Add null terminator
}
int main() {
    char source[] = "Hello, World!";
    char destination[50];
    my_strcpy(destination, source);
    printf("Copied String: %s\n", destination);
    return 0;
}
```

Explanation:

- The function loops through `src`, copying characters one by one to `dest`.
- A null terminator (`'\0'`) is added at the end.

Comparing Strings (`strcmp()`)

The `strcmp()` function compares two strings lexicographically.

Using Library Function

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

```
    char str1[] = "Hello";
    char str2[] = "World";
    int result = strcmp(str1, str2);
    if (result == 0)
        printf("Strings are equal\n");
    else if (result < 0)
        printf("str1 is less than str2\n");
    else
        printf("str1 is greater than str2\n");
    return 0;
}
```

Manual Implementation (`my_strcmp()`)

```
#include <stdio.h>
int my_strcmp(const char *str1, const char *str2) {
    while (*str1 && (*str1 == *str2)) { // Compare character by
character
        str1++;
        str2++;
    }
    return *(unsigned char *)str1 - *(unsigned char *)str2;
}
int main() {
    char str1[] = "Hello";
```

```
    char str2[] = "World";

    int result = my_strcmp(str1, str2);

    if (result == 0)

        printf("Strings are equal\n");

    else if (result < 0)

        printf("str1 is less than str2\n");

    else

        printf("str1 is greater than str2\n");

    return 0;

}
```

Explanation:

- The function compares characters until a difference is found or a null terminator is reached.
- It returns:
    - `0` if the strings are equal.
    - A negative value if `str1` is smaller.
    - A positive value if `str1` is larger.

Concatenating Strings (`strcat()`)

The `strcat()` function appends one string to another.

Using Library Function

```
#include <stdio.h>

#include <string.h>

int main() {

    char str1[50] = "Hello, ";
```

```
    char str2[] = "World!";
    strcat(str1, str2); // Append str2 to str1
    printf("Concatenated String: %s\n", str1);
    return 0;
}
```

Manual Implementation (`my_strcat()`)

```
#include <stdio.h>
void my_strcat(char *dest, const char *src) {
    while (*dest) dest++; // Move to end of dest
    while (*src) {  // Copy src to dest
        *dest = *src;
        dest++;
        src++;
    }
    *dest = '\0';  // Null-terminate the string
}
int main() {
    char str1[50] = "Hello, ";
    char str2[] = "World!";
    my_strcat(str1, str2);

    printf("Concatenated String: %s\n", str1);
```

```
    return 0;
}
```

Explanation:

- The function moves `dest` to its null terminator.
- It then copies `src` to `dest` and adds a null terminator.

Selecting a Substring (`strncpy()` or Custom Function)

The `strncpy()` function copies a specified number of characters.

Using Library Function

```
#include <stdio.h>
#include <string.h>
int main() {
    char source[] = "Hello, World!";
    char substring[6];
    strncpy(substring, source, 5); // Copy first 5 characters
    substring[5] = '\0';  // Ensure null termination
    printf("Substring: %s\n", substring);
    return 0;
}
```

Manual Implementation (`my_substr()`)

```
#include <stdio.h>
void my_substr(char *dest, const char *src, int start, int length) {
    int i;
    for (i = 0; i < length && src[start + i] != '\0'; i++) {
```

```
        dest[i] = src[start + i];
    }
    dest[i] = '\0'; // Null-terminate the substring
}
int main() {
    char source[] = "Hello, World!";
    char substring[6];
    my_substr(substring, source, 0, 5); // Extract first 5 characters
    printf("Substring: %s\n", substring);
    return 0;
}
```

Explanation:

- The function starts at index `start` and copies `length` characters.
- It ensures null termination to prevent garbage values.

Program : Printing A One-Month Reminder List

```
#include <stdio.h>
#include <string.h>
#define MAX_REMINDERS 50
#define MAX_LENGTH 100
int main()
{
    int dates[MAX_REMINDERS];                 // Array to store dates
    char messages[MAX_REMINDERS][MAX_LENGTH]; // 2D Array to store
messages
    int count = 0;
    printf("Enter your reminders (Format: Date Reminder_Message)\n");
    printf("Enter '0 0' to stop.\n");
    while (1)
```

```
{
    int day;
    char msg[MAX_LENGTH];
    printf("Enter reminder (day first, then message): ");
    scanf("%d", &day); // Read the date
    if (day == 0)
        break; // Stop input when '0 0' is entered
    getchar(); // Consume the newline left by scanf
    gets(msg); // ⚠ Unsafe! Vulnerable to buffer overflow.
    // Store date and reminder message
    dates[count] = day;
    strcpy(messages[count], msg);
    count++;
    if (count >= MAX_REMINDERS)
    {
        printf("Reminder list is full!\n");
        break;
    }
}
// Sort reminders by date using Bubble Sort
for (int i = 0; i < count - 1; i++)
{
    for (int j = i + 1; j < count; j++)
    {
        if (dates[i] > dates[j])
        {
            // Swap dates
            int temp = dates[i];
            dates[i] = dates[j];
            dates[j] = temp;
            // Swap messages
            char tempMsg[MAX_LENGTH];
            strcpy(tempMsg, messages[i]);
            strcpy(messages[i], messages[j]);
            strcpy(messages[j], tempMsg);
        }
    }
}
```

```
    // Display the reminders
    printf("\nYour Reminders for the Month:\n");
    for (int i = 0; i < count; i++)
    {
        printf("Date: %d - %s\n", dates[i], messages[i]);
    }

    return 0;
}
```

13.6 String Idioms

String idioms refer to common patterns or techniques used to manipulate strings efficiently. In C, strings are character arrays, and many operations require manual handling with functions from <string.h>.

Copying a String

Idiom: Using strcpy()

The standard strcpy() function is used to copy one string into another.

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

int main() {
    char source[] = "Hello, World!";
    char destination[50];

    strcpy(destination, source);  // Copying source into destination
    printf("Copied String: %s\n", destination);

    return 0;
}
```

Comparing Strings

Idiom: Using strcmp()

The strcmp() function compares two strings lexicographically.

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

int main() {
    char str1[] = "apple";
    char str2[] = "banana";

    if (strcmp(str1, str2) < 0)
        printf("%s comes before %s\n", str1, str2);
    else if (strcmp(str1, str2) > 0)
        printf("%s comes after %s\n", str1, str2);
    else
        printf("Strings are equal\n");

    return 0;
}
```

Concatenating Strings

Idiom: Using strcat()

strcat() appends one string to another.

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

int main() {
    char str1[50] = "Hello, ";
    char str2[] = "World!";

    strcat(str1, str2);  // Concatenating str2 to str1
    printf("Concatenated String: %s\n", str1);

    return 0;
}
```

Finding a Substring

Idiom: Using strstr()

strstr() searches for a substring within a string.

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

int main() {
    char str[] = "C programming is powerful!";
    char *sub = strstr(str, "programming");

    if (sub)
        printf("Substring found: %s\n", sub);
    else
        printf("Substring not found.\n");

    return 0;
}
```

Finding String Length

Idiom: Using strlen()

strlen() returns the number of characters before the null terminator (\0).

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

int main() {
    char str[] = "Hello, World!";
    printf("Length: %lu\n", strlen(str));

    return 0;
}
```

13.7 Arrays of Strings

An array of strings in C is essentially a 2D array, where each element of the array is a string (a character array). Strings in C are terminated by a null character ('\0'), so an array of strings is just a collection of character arrays.

How Array of Strings is Stored in Memory

- An array of strings is a 2D character array.
- Each string is a sequence of characters terminated by `'\0'`, and each string is stored as a row (or 1D array) in the 2D array.
- The outer array holds pointers (or references) to each individual string (character array), and each string is terminated with a null character (`'\0'`).

Example :

```
#include <stdio.h>
int main() {
    // Array of strings (2D array of characters)
    char names[3][20] = {
        "Alice",   // 1st string
        "Bob",     // 2nd string
        "Charlie"  // 3rd string
    };
    // Printing the strings
    for (int i = 0; i < 3; i++) {
        printf("Name %d: %s\n", i + 1, names[i]);
    }
    return 0;
}
```

Accessing Elements in an Array of Strings

- You can access each string in the array using the array indexing, e.g., names[0], names[1], etc.
- To access individual characters of a string, you can use a second index, e.g., names[i][j], where i is the string index and j is the character index.

Example :

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

```
    char names[3][20] = {
        "Alice",
        "Bob",
        "Charlie"
    };
    // Accessing individual characters
    for (int i = 0; i < 3; i++) {
        for (int j = 0; names[i][j] != '\0'; j++) {
            printf("Character at names[%d][%d]: %c\n", i, j,
names[i][j]);
        }
    }
    return 0;
}
```

Array of Pointers to Strings

In this approach, we have an array of pointers, where each pointer points to a string (a character array). The strings themselves can be stored in string literals, or we can dynamically allocate memory for them.

Example of Array of Pointers to Strings

```
#include <stdio.h>
int main() {
    // Array of pointers to strings
    char *names[] = {"Alice", "Bob", "Charlie"};
    // Printing the strings using array of pointers
    for (int i = 0; i < 3; i++) {
        printf("Name %d: %s\n", i + 1, names[i]);
    }
    return 0;
}
```

Explanation:

1. **`char *names[]`**:
    - This is an array of **pointers**. Each element of `names[]` is a pointer that points to the beginning of a string.

    - The strings `"Alice"`, `"Bob"`, and `"Charlie"` are stored as string literals in memory, and each pointer in `names[]` points to the respective string.
2. **Accessing Strings**:
    - `names[i]` gives the **i-th string**, where `names[i]` is a **pointer to the first character** of the string.
    - `printf("Name %d: %s\n", i + 1, names[i]);` prints each string stored in the `names` array.

Chapter - 16 (Structures, Unions and Enumerators)

A structure in C is a user-defined data type that allows grouping variables of different types under one name.

16.1 Structure Variables

A structure variable is a variable that is declared based on a structure in C (or similar programming languages). A structure (`struct`) is a user-defined data type that allows grouping different types of variables under a single name.

Declaring Structure Variables

A structure in C allows you to group variables of different types under one name. After defining a structure, you can declare variables of that structure type. These are called structure variables.

Syntax:

```
struct StructureName {

    dataType member1;

    dataType member2;

    ...

};
```

Example:

```
struct Student {
    int id;
    char name[50];
    float gpa;
};
```

Here, `Student` is a structure type, but it doesn't create any variables yet. To use it, you must declare structure variables:

```
struct Student s1, s2;
```

Now s1 and s2 are structure variables of type `struct Student`. Each variable will have its own `id`, `name`, and `gpa`.

Initializing Structure Variables

You can assign values to a structure variable at the time of declaration using positional initialization, where you provide values in the same order as the structure members are declared.

Example:

```
struct Student s1 = {101, "Alice", 3.85};
```

This assigns:

- `id = 101`
- `name = "Alice"`
- `gpa = 3.85`

If you skip some values, only the provided ones are initialized, and the rest are set to default values (e.g., 0 for numbers, empty for arrays).

Designated Initializers

Introduced in the C99 standard, designated initializers allow you to specify values for specific structure members by name. This gives more control and clarity and allows you to initialize members in any order.

Example:

```
struct Student s2 = {
    .gpa = 3.75,
    .id = 102,
    .name = "Bob"
};
```

Here:

- `gpa` is set to 3.75
- `id` is set to 102

- `name` is set to "Bob"

Even though `id` is declared first in the structure, we initialize `gpa` first. This is valid because we're using designated initializers.

Accessing Structure Members

Use the dot (`.`) operator to access or modify members of a structure variable.

Syntax:

```
variableName.memberName
```

Example:

```
s1.id = 101;
strcpy(s1.name, "Alice");
s1.gpa = 3.85;
```

Copying Structure Variables

You can assign one structure variable to another if they are of the same type. This copies all members.

Example:

```
struct Student s2;
s2 = s1; // All members of s1 are copied to s2
```

Note: This is a **shallow copy**, not a deep copy. For nested pointers or dynamically allocated memory inside structures, this does not duplicate the actual data.

Passing Structures to Functions

There are three ways to pass structures to functions:

a) Pass by Value

Make a copy of the structure. Changes inside the function do not affect the original.

```
void printStudent(struct Student s) {
```

```
    printf("%d %s %.2f\n", s.id, s.name, s.gpa);
}
```

Call it with:

```
printStudent(s1);
```

b) Pass by Reference (using Pointers)

Allows modification of the original structure.

```
void updateGPA(struct Student *s, float newGPA) {
    s->gpa = newGPA;
}
```

Call it with:

```
updateGPA(&s1, 4.00);
```

Use `->` to access members via pointer.

Returning a Structure from a Function

Functions can return structure variables.

Example:

```
struct Student createStudent(int id, char name[], float gpa) {
    struct Student s;
    s.id = id;
    strcpy(s.name, name);
    s.gpa = gpa;
    return s;
}
```

Call it like:

```
struct Student s3 = createStudent(103, "Charlie", 3.9);
```

16.2 Structure Types

Declaring a structure with the same member types again and again is a hectic job.

```
struct
{
    int number;
    char name[NAME_LEN + 1];
    int on_hand
} part1;

struct
{
    int number;
    char name[NAME_LEN + 1];
    int on_hand
} part2;
```

Instead of repeating the code again and again we can just use typedef instead.

Using `typedef` in C

The `typedef` keyword in C allows you to create an **alias for data types**, which:

- Improves code readability
- Reduces redundancy
- Makes complex structures easier to manage

Using `typedef` with Structures

Normally, when declaring a structure, you must use the `struct` keyword:

```
struct Student {
    char name[50];
    int age;
    float marks;
};
```

```
struct Student s1;  // Must use "struct Student"
```

With `typedef`, you can define an alias:

```
typedef struct {
    char name[50];
    int age;
    float marks;
} Student;

Student s1;  // Now "struct" keyword is not required
```

Using typedef with Nested Structures

You can also use typedef with nested structures for better readability:

```
typedef struct {
    char city[50];
    int zip;
} Address;

typedef struct {
    char name[50];
    int age;
    float marks;
    Address address;  // No need for "struct Address"
} Student;
```

Example Program Using `typedef`

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

typedef struct {
    char city[50];
    int zip;
} Address;

typedef struct {
```

```
    char name[50];
    int age;
    float marks;
    Address address;
} Student;

void printStudent(Student *s) {
    printf("Name: %s\n", s->name);
    printf("Age: %d\n", s->age);
    printf("Marks: %.2f\n", s->marks);
    printf("City: %s\n", s->address.city);
    printf("Zip Code: %d\n", s->address.zip);
}

int main() {
    Student s1 = {"John Doe", 20, 85.5, {"New York", 10001}};
    Student *ptr = &s1;
    printStudent(ptr);
    return 0;
}
```

Structure Arguments and Return Types

Passing Structures as Arguments to Functions

Structures can be passed to functions by value, which means a copy of the structure is made. Modifications in the function do not affect the original structure (unless passed by pointer).

Example: Passing by Value

```
void printStudentInfo(struct Student s) {
    printf("Name: %s\n", s.name);
    printf("Age: %d\n", s.age);
    printf("Grade: %.2f\n", s.grade);
}
```

Here, s is a local copy of the `Student` structure passed to `printStudentInfo`.

Returning Structures from Functions

Functions in C can return structures. When a structure is returned, it is copied back to the calling function.

Example: Returning a Modified Structure

```
struct Student updateGrade(struct Student s, float newGrade) {
    s.grade = newGrade;  // Modify the grade
    return s;            // Return the updated structure
}
```

Full Example Program: Passing and Returning Structures

```
#include <stdio.h>

struct Student {
    char name[50];
    int age;
    float grade;
};

// Function to update the grade of a student
struct Student updateGrade(struct Student s, float newGrade) {
    s.grade = newGrade;  // Modify the grade
    return s;            // Return the updated structure
}

int main() {
    struct Student student1 = {"Alice", 20, 85.5};

    // Print original grade
    printf("Original grade: %.2f\n", student1.grade);

    // Update grade using the function
    student1 = updateGrade(student1, 90.0);

    // Print updated grade
    printf("Updated grade: %.2f\n", student1.grade);

    return 0;
```

```
}
```

Explanation

- A `Student` structure is defined with `name`, `age`, and `grade`.
- In `main()`, `student1` is created and initialized.
- `updateGrade()` is called to return a new `Student` with the updated grade.
- The updated structure is reassigned to `student1`.
- The original and updated grades are printed using `printf()`.

16.3 Nested Arrays and Structures in C

Nested Structures

A nested structure is a structure that contains another structure as a member. This allows for a logical grouping of related sub-entities under a main entity. For example, an address can be a part of a student structure.

Syntax

```
struct Address {
    char city[50];
    int zip;
};

struct Student {
    char name[50];
    int age;
    float grade;
    struct Address address;  // Nested structure
};
```

Explanation

- `struct Address` defines a separate structure for storing address-related data.
- `struct Student` contains an `Address` variable as a member, thus nesting one structure within another.

- Access to nested members uses the dot (`.`) operator: `student.address.city`.

Access Example

```
struct Student student1;
student1.age = 20;
student1.grade = 88.5;
student1.address.zip = 12345;
```

Arrays of Structures

An array of structures is used to store multiple structure variables of the same type in a sequential manner. This is useful when we need to manage a list of similar entities like multiple students, employees, etc.

Syntax:

```
struct Student {
    char name[50];
    int age;
    float grade;
};

struct Student students[10];  // Array of 10 student structures
```

Explanation

- Each element in the array (`students[0]`, `students[1]`, etc.) is a full structure variable with its own set of fields.
- This allows us to manage multiple sets of related data (e.g., multiple student records) efficiently.

Access Example

```
students[0].age = 18;
students[1].grade = 92.5;
```

Initializing an Array of Structures

C allows arrays of structures to be initialized at the time of declaration using a list of values. This method is concise and ensures that each structure variable starts with defined values.

Syntax: Without Nested Structures

```
struct Student {
    char name[50];
    int age;
    float grade;
};

struct Student students[3] = {
    {"Alice", 20, 85.5},
    {"Bob", 21, 90.0},
    {"Charlie", 19, 78.0}
};
```

Syntax: With Nested Structures

```
struct Address {
    char city[50];
    int zip;
};

struct Student {
    char name[50];
    int age;
    float grade;
    struct Address address;
};

struct Student students[2] = {
    {"Alice", 20, 85.5, {"New York", 10001}},
    {"Bob", 21, 90.0, {"Los Angeles", 90001}}
};
```

Explanation

- Each structure is initialized with values for all its fields, including nested structure members.
- For the nested structure, a sub-initializer is used (e.g., `{"New York", 10001}`).

Complete Example: Nested Structures with Array of Structures

```
#include <stdio.h>

struct Address {

    char city[50];

    int zip;

};

struct Student {

    char name[50];

    int age;

    float grade;

    struct Address address;

};

void displayStudents(struct Student s[], int count) {

    int i;

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

        printf("Student %d:\n", i + 1);

        printf("Name: %s\n", s[i].name);

        printf("Age: %d\n", s[i].age);

        printf("Grade: %.2f\n", s[i].grade);
```

```
        printf("City: %s\n", s[i].address.city);
        printf("Zip: %d\n\n", s[i].address.zip);
    }
}
int main() {
    struct Student students[2] = {
        {"Alice", 20, 85.5, {"New York", 10001}},
        {"Bob", 21, 90.0, {"Los Angeles", 90001}}
    };
    displayStudents(students, 2);
    return 0;
}
```

Program Explanation

- Two structures are defined: `Address` and `Student`.
- Each student contains a nested `Address` structure.
- The `students` array holds multiple student records.
- The `displayStudents` function takes the array as input and prints all student information using a loop.

Program : Maintains a part database

```
#include <stdio.h>
#include <string.h>
#define NAME_LEN 25
#define MAX_PARTS 100

struct part
{
    int number;
```

```
    char name[NAME_LEN + 1];
    int on_hand;
} inventory[MAX_PARTS];

int num_parts = 0;

int find_part(int number);
void insert(void);
void search(void);
void update(void);
void print(void);

int main()
{
    char code;
    for (;;)
    {
        printf("Enter operation code: ");
        scanf(" %c", &code);  // fix: add & before code
        while (getchar() != '\n') ;  // discard the rest of the line

        switch (code)
        {
            case 'i': insert();
                      break;
            case 's': search();
                      break;
            case 'u': update();
                      break;
            case 'p': print();
                      break;
            case 'q': return 0;
            default:  printf("Illegal code\n");
        }
        printf("\n");
    }
}
```

```
int find_part(int number)
{
    for (int i = 0; i < num_parts; i++)
    {
        if (inventory[i].number == number)
            return i;
    }
    return -1;
}

void insert(void)
{
    int part_number;

    if (num_parts == MAX_PARTS)
    {
        printf("Database is full; cannot add more parts.\n");
        return;
    }

    printf("Enter part number: ");
    scanf("%d", &part_number);

    if (find_part(part_number) >= 0)
    {
        printf("Part already exists.\n");
        return;
    }

    inventory[num_parts].number = part_number;
    printf("Enter part name: ");
    scanf(" %[^\n]", inventory[num_parts].name);  // Read string with
spaces
    printf("Enter quantity on hand: ");
    scanf("%d", &inventory[num_parts].on_hand);
    num_parts++;
}
```

```
void search(void)
 {
    int number, i;
    printf("Enter part number: ");
    scanf("%d", &number);
    i = find_part(number);
    if (i >= 0)
    {
        printf("Part name: %s\n", inventory[i].name);
        printf("Quantity on hand: %d\n", inventory[i].on_hand);
    }
    else
    {
        printf("Part not found.\n");
    }
}

void update(void)
{
    int number, change, i;
    printf("Enter part number: ");
    scanf("%d", &number);
    i = find_part(number);
    if (i >= 0)
    {
        printf("Enter change in quantity on hand: ");
        scanf("%d", &change);
        inventory[i].on_hand += change;
    }
    else
    {
        printf("Part not found.\n");
    }
}

void print(void)
{
```

```
    printf("Part Number   Part Name                  Quantity on
Hand\n");
    for (int i = 0; i < num_parts; i++)
    {
        printf("%7d       %-25s%11d\n", inventory[i].number,
               inventory[i].name, inventory[i].on_hand);
    }
}
```

16.4 Unions

A union in the C programming language is a user-defined data type that is similar to a structure (`struct`). Like structures, unions allow the grouping of variables of different types under a single name. However, the key difference lies in memory allocation: while a structure allocates separate memory for each of its members, a union allocates a single shared memory location that is used by all its members.

This means that at any given time, a union can hold a value for only one of its members, although all members are accessible by name.

Declaration and Syntax

The syntax for declaring a union is as follows:

```
union union_name {
    data_type1 member1;
    data_type2 member2;
    ...
    data_typeN memberN;
};
```

Example:

```
union Data {
    int i;
    float f;
    char str[20];
};
```

In the example above, `Data` is a union with three members: an integer, a float, and a character array. All three share the same memory space.

Memory Allocation in Unions

The memory allocated for a union is equal to the size of its largest member. This is in contrast with structures, where the memory allocated is the sum of the sizes of all members (plus possible padding).

In the previous example:

- `int i` requires 4 bytes (typically),
- `float f` requires 4 bytes,
- `char str[20]` requires 20 bytes,

Thus, the total memory allocated for the union is 20 bytes, not 28 bytes as it would be in a structure.

This shared memory model enables efficient use of memory, especially in scenarios where a variable may hold different data types at different times.

Accessing Union Members

Union members are accessed in the same way as structure members, using the dot (`.`) operator.

Example:

```
union Data data;
data.i = 10;
printf("%d\n", data.i);

data.f = 3.14;
printf("%f\n", data.f);

// Accessing data.i now results in undefined behavior.
```

Here, after assigning `data.f`, the previous value stored in `data.i` is no longer valid because the memory location has been overwritten.

Use Cases and Advantages of Unions

1. Saving Space in Memory

Unions are often used when memory usage is a concern. Since all members share the same memory, unions are memory-efficient in scenarios where:

- Only one of several data members will be used at a time.
- Systems have limited memory (e.g., embedded systems, microcontrollers).

Example:

```
union SensorData {
    int intReading;
    float floatReading;
    char textReading[50];
};
```

Instead of storing all types of readings in separate variables or a structure (which would require more memory), this union stores only one type of reading at any time, reducing memory consumption.

2. Mixed Data Structures

Unions can be used in combination with structures to build complex and flexible data structures that can handle multiple data types.

Example:

```
struct Variant {
    int type;  // tag field: 1 = int, 2 = float, 3 = string
    union {
        int i;
        float f;
        char str[20];
    } data;
};
```

This structure (`Variant`) includes a union and an additional field called a "tag" that specifies the type of data currently stored in the union. This allows the structure to handle different types of data in a controlled and safe way.

3. Tagged Unions (Discriminated Unions)

Tagged unions are a common technique used to associate a "type indicator" with a union. Since a union can only store one member's value at a time, it is crucial to keep track of which member is currently being used. This is done using a tag field, often implemented as an enumeration or integer.

Example:

```
#define INT_TYPE 1
#define FLOAT_TYPE 2
#define STRING_TYPE 3

struct TaggedUnion {
    int tag;  // indicates the type of data stored
    union {
        int i;
        float f;
        char str[20];
    } data;
};
```

Usage:

```
struct TaggedUnion value;

value.tag = FLOAT_TYPE;
value.data.f = 3.14;

if (value.tag == FLOAT_TYPE) {
    printf("Float: %.2f\n", value.data.f);
}
```

This approach is widely used in:

- Interpreters and compilers (to represent expressions of various types),
- Abstract syntax trees,

- Event systems,
- Configuration file parsers.

16.5 Enumeration in C

An enumeration (`enum`) in C is a user-defined data type that consists of named integer constants. It allows the programmer to define variables that can only take one out of a small set of meaningful values, which improves code readability and maintainability.

Syntax:

```
enum EnumName {
    ENUM_CONSTANT1,
    ENUM_CONSTANT2,
    ...
};
```

Example:

```
enum Weekday { MON, TUE, WED, THU, FRI, SAT, SUN };
```

- By default, the values are assigned as: `MON = 0`, `TUE = 1`, ..., `SUN = 6`.

Enumeration Tags and Type Names in C

Enumeration Tag

An enumeration tag is the name given to an enumeration type. It allows defining multiple variables of that type.

```
enum Color { RED, GREEN, BLUE };
enum Color shirt, cap;
```

- Here, `Color` is the **tag** for the enumeration.
- It can be used to declare variables like `enum Color shirt`.

Type Name with `typedef`

A type name is created using `typedef` to give the enumeration a simpler alias.

```
typedef enum Color { RED, GREEN, BLUE } Color;

Color bag, pen;
```

- `Color` becomes a **type name**; we no longer need to use the `enum` keyword when declaring variables.

Anonymous Enumeration (no tag or type name)

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

- This is a one-time-use enumeration.
- You cannot reuse the type elsewhere because it has no name.

Best Practice:

Use both tag and type name:

```
typedef enum State { IDLE, RUNNING, STOPPED } State;
```

- `State` is both the tag and the type alias.
- Preferred for reusability and clarity.

Enumeration as Integers

Enumerators in C are integers by default. They are internally treated as constants of type `int`, unless specified differently.

Default Value Assignment:

```
enum Status { OFF, ON };
```

- `OFF = 0`
- `ON = 1`

Custom Value Assignment:

```
enum HttpCode { OK = 200, NOT_FOUND = 404, SERVER_ERROR = 500 };
```

Operations Allowed:

Because enumerators are integers:

- They can be compared: `if (code == OK)`
- They can be used in arithmetic: `code + 100`
- They can be used in switch statements

However, C does not restrict assigning any arbitrary integer to an `enum` variable, so additional checks may be required.

Using Enumeration to Declare a Tag Field

Enumerations can be used to declare a tag field inside a `struct` when building mixed-type or tagged unions (common in systems programming and compilers).

What is a Tag Field?

A tag field is a field in a structure that indicates which variant of data is currently in use, especially useful when combined with `union`.

Example: Mixed-Type Structure Using Enumeration as a Tag Field

```
#include <stdio.h>

typedef enum DataType { INT_TYPE, FLOAT_TYPE, CHAR_TYPE } DataType;

typedef struct {
    DataType tag;  // This is the tag field
    union {
        int i;
        float f;
        char c;
    } data;
} Variant;

void print(Variant v) {
    switch (v.tag) {
        case INT_TYPE:
            printf("Integer: %d\n", v.data.i);
            break;
        case FLOAT_TYPE:
            printf("Float: %.2f\n", v.data.f);
```

```
            break;
        case CHAR_TYPE:
            printf("Char: %c\n", v.data.c);
            break;
    }
}

int main() {
    Variant v1 = { INT_TYPE, .data.i = 42 };
    Variant v2 = { FLOAT_TYPE, .data.f = 3.14f };
    Variant v3 = { CHAR_TYPE, .data.c = 'A' };

    print(v1);
    print(v2);
    print(v3);

    return 0;
}
```

How It Works:

- `tag` (of type `DataType`) holds the enum value indicating which field in the `union` is active.
- The `union` saves memory by sharing space among different types.
- The `enum` ensures only the correct type is accessed.

Advantages:

- Memory efficiency (using `union`)
- Type safety and clarity (using `enum` as tag)
- Structured way to manage variant data types