

Course Material for

Subject Title: Computer Programming

Subject Code:
CS-214

Prepared by:
Dr. Preetpal Kaur Buttar
Assistant Professor (CSE)



Department of Computer Science & Engineering
Sant Longowal Institute of Engineering & Technology,
Longowal

Title of the course : **Computer Programming**

Subject Code : **CS-214**

Weekly load : 7 Hrs

Credit : 5 (Lecture 3, Practical 2)

LTP 3-0-4

Course Outcomes: At the end of the course, the student will be able to:

CO1	Use different data types & design programs involving decision structures, loops and functions.
CO2	Understand the concept of arrays, structures and union through which derived data types can be used, pointers dealing with memory management and file handling for permanent storage of data and record.
CO3	Understand the basic concepts of C programming which provides students with the means of writing efficient code compile & debug programs in C language.

CO/PO Mapping : (Strong(S)/Medium(M)/Weak(W) indicates strength of correlation)										
COs	Programme Outcomes (POs)									
	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10
CO1		S	M	S						S
CO2		S	S	M						S
CO3		S	M	S						S

Theory

Unit	Main Topics	Course outlines	Lecture(s)
Unit-1	1. Introduction	Steps in development of a program, Flow charts, Algorithm and Program Debugging.	06
	2. Program Structure	I/O statements, assign statements. Constants, variables and data types, Operators and Expressions, Standards and Formatted, Use of Header & Library files.	06
	3. Control Structures	Introduction, Decision making with IF – statement, IF – Else and Nested IF, While and do-while, for loop, Break and switch statements.	10
	4. Functions	Introduction to functions, Global and Local Variables, Function Declaration, Standard functions, Parameters and Parameter Passing, Call – by value/reference, Recursion.	06
Unit-2	5. Arrays	Introduction to Arrays, Array Declaration and Initialization, Single and Multidimensional Array. Arrays of characters.	06
	6. Structures and Unions	Declaration of structures, Accessing structure members, Structure Initialization, Arrays of structures, Unions.	04

	7. Pointers	Introduction to Pointers, Address operator and pointers, Declaring and Initializing pointers, Assignment through pointers, Pointers and Arrays.	06
	8. Files	Introduction, File reading/writing in different modes, File manipulation using standard function types	04

Total = 48

Recommended books:

1. Salaria RS, Application Programming in C, Khanna Book Publishing Co (P) Ltd. New Delhi
2. Schaum Series, Programming in C, McGraw Hills Publishers, New York
3. Yashwant Kanetkar, Exploring – BPB Publications, New Delhi

Table of Contents

Chapter 1 Introduction	1
1.1 Steps for problem solving	1
1.2 Flow of control.....	2
1.3 Representation of algorithms	4
1.4 Program debugging	5
1.5 Basic program structure	7
Chapter 2 Program Structure	11
2.1 Identifiers and their rules	11
2.2 Keywords	13
2.3 Constants.....	13
2.4 Variables	17
2.5 Comments	23
2.6 Data types.....	25
2.7 Operators.....	32
2.8 Expressions	46
2.9 Type conversion.....	47
2.10 Statements	50
2.11 Use of header and library files	53
Chapter 3 Control Structures	65
3.1 Introduction.....	65
3.2 Decision making with if , if...else , and nested ifs	65
3.3 The while loop	73
3.4 The do...while loop.....	75
3.5 The for loop.....	77
3.6 The break statement	80
3.7 The continue statement.....	81
3.8 The switch statement.....	82
Chapter 4 Functions	86
4.1 Introduction to functions	86
4.2 Types of functions.....	89
4.3 Passing parameters to functions	91
4.4 Global and local variables.....	93
4.5 Recursion	95

Chapter 5 Arrays	103
5.1 Introduction to arrays	103
5.2 C array declaration	103
5.3 C array initialization.....	104
5.4 Accessing array elements.....	105
5.5 Updating array elements	106
5.6 C array traversal	106
5.7 How to use arrays in C?	106
5.8 Types of arrays in C.....	107
5.9 Relationship between arrays and pointers.....	110
5.10 Passing an array to a function in C.....	111
5.11 Returning an array from a function in C	112
5.12 Properties of arrays in C.....	113
5.13 Examples of arrays in C	114
5.14 Advantages and disadvantages of arrays in C.....	116
Chapter 6 Structures and Unions	117
6.1 C structure declaration	117
6.2 C structure definition	117
6.3 Accessing structure members.....	117
6.4 Initializing structure members.....	118
6.5 Nested structures	120
6.6 Array of structures	122
6.7 Uses of structures in C	124
6.8 Limitations of C structures.....	124
6.9 Unions	125
6.10 Difference between C structure and C union	128
Chapter 7 Pointers	129
7.1 Syntax of C pointers.....	129
7.2 How to use pointers?.....	129
7.3 Types of pointers in C.....	131
7.4 Size of pointers in C.....	133
7.5 C pointers arithmetic	134
7.6 C pointers and arrays.....	135
7.7 Uses of pointers in C	137
7.8 Advantages and disadvantages of pointers	137

Chapter 8 Files	138
8.1 Need for file handling in C	138
8.2 Types of files.....	138
8.3 The FILE pointer (FILE*)	138
8.4 Opening (creating) a file	138
8.5 File opening modes	139
8.6 Writing to a text file	141
8.7 Reading from a text file	144
8.8 Binary read and write functions	146
8.9 Writing to a binary file.....	146
8.10 Reading from a binary file	147

Chapter 1

Introduction

1.1 Steps for problem solving

Introduction:

A program is a set of instructions that a computer can execute to perform a specific task. Developing a program involves several stages, including defining the problem, planning, designing, coding, testing, and maintenance. In this lecture, we will explore each of these stages in detail.

Defining the Problem:

The first step in developing a program is to define the problem that the program is intended to solve. This involves identifying the requirements of the program, such as its inputs, outputs, and desired behavior. The problem definition helps to determine the scope of the project and provides a clear understanding of what the program should accomplish.

Planning:

Once the problem has been defined, the next step is to create a plan for how to solve it. This involves breaking down the problem into smaller, manageable tasks, determining the resources required, and creating a timeline for completion. The plan provides a roadmap for the development process and helps to ensure that the program is developed in an efficient and effective manner.

Designing:

The design phase is where the program's architecture and structure are created. This involves creating a detailed plan for how the program will be organized, including the algorithms that will be used to perform the necessary tasks. The design should take into account the requirements of the problem and the constraints of the hardware and software platforms on which the program will run.

Coding:

The coding phase is where the program is actually written. During this phase, the code is written in a programming language, such as Python, Java, or C++. The code should be written in a way that is easy to understand and maintain, and should be organized in a logical and modular fashion. It is also important to write and document the code in such a way that it can be easily understood and modified by others.

Testing:

Once the program has been written, it is time to test it to ensure that it works as intended. This involves running the program with various inputs and comparing the results to the expected outputs. If the program does not produce the desired results, the code may need to be modified and tested again. The testing phase also helps to identify any bugs or errors in the code and ensure that the program is functioning correctly.

Maintenance:

Once the program has been tested and is working as intended, it is ready to be released. However, the development process does not end there. The program may need to be updated and modified in the future to accommodate changes in the hardware or software platforms on which it runs, or to add new features.

The maintenance phase involves updating the program to ensure that it continues to work as intended and addressing any issues that may arise.

Conclusion:

The development of a program involves several stages, including defining the problem, planning, designing, coding, testing, and maintenance. Each of these stages is important in ensuring that the program is developed in an efficient and effective manner and that it functions correctly. Understanding the steps in the development of a program is critical for anyone who is interested in creating software applications.

1.2 Flow of control

Introduction:

A flowchart is a diagram that represents a process or algorithm in a graphical and sequential manner. Flowcharts are used to visualize and document complex processes, making it easier to understand and analyze them. They are also useful for communicating ideas and designs to others, as well as for testing and debugging programs. In this lecture, we will explore flowcharts in more detail and use an example to demonstrate their use.

Components of a Flowchart:

A flowchart consists of several basic components, including:

- **Start/Stop symbols:** These symbols indicate the start and end of the process or algorithm being represented by the flowchart.

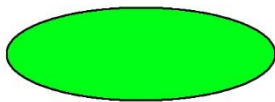


Fig. 1.1: A start/stop symbol

- **Input/Output:** A parallelogram denotes any function of input/output type. Program instructions that take input from input devices and display output on output devices are indicated with parallelogram in a flowchart.



Fig. 1.2: An input/output symbol

- **Process symbols:** These symbols represent a step in the process or algorithm, such as a calculation or decision.



Fig. 1.3: A process symbol

- **Decision symbols:** These symbols represent a decision point in the process or algorithm, where the flow of control is determined based on the outcome of a condition.

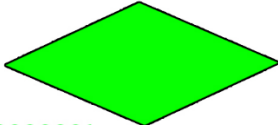


Fig. 1.4: A decision symbol

- **Connector symbols:** These symbols represent the flow of control between steps in the process or algorithm, showing the order in which the steps are executed.

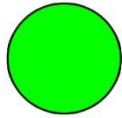


Fig. 1.5: A connector symbol

- **Flow lines:** Flow lines indicate the exact sequence in which instructions are executed. Arrows represent the direction of flow of control and relationship among different symbols of flowchart.

Rules For Creating Flowchart:

A flowchart is a graphical representation of an algorithm. It should follow some rules while creating a flowchart:

Rule 1: Flowchart opening statement must be 'start' keyword.

Rule 2: Flowchart ending statement must be 'end' keyword.

Rule 3: All symbols in the flowchart must be connected with an arrow line.

Rule 4: The decision symbol in the flowchart is associated with the arrow line.

Advantages of Flowchart:

- Flowcharts are a better way of communicating the logic of the system.
- Flowcharts act as a guide for blueprint during program designed.
- Flowcharts help in debugging process.
- With the help of flowcharts programs can be easily analyzed.
- It provides better documentation.
- Flowcharts serve as a good proper documentation.
- Easy to trace errors in the software.
- Easy to understand.
- The flowchart can be reused for inconvenience in the future.
- It helps to provide correct logic.

Disadvantages of Flowchart:

- It is difficult to draw flowcharts for large and complex programs.
- There is no standard to determine the amount of detail.
- Difficult to reproduce the flowcharts.
- It is very difficult to modify the Flowchart.
- Making a flowchart is costly.
- Some developer thinks that it is waste of time.
- It makes software processes low.
- If changes are done in software, then the flowchart must be redrawn.

Example:

To demonstrate the use of flowcharts, let's consider a simple example of a program that draws a flowchart to input two numbers from the user and display the largest of two numbers. The flowchart for this program would look something like this:

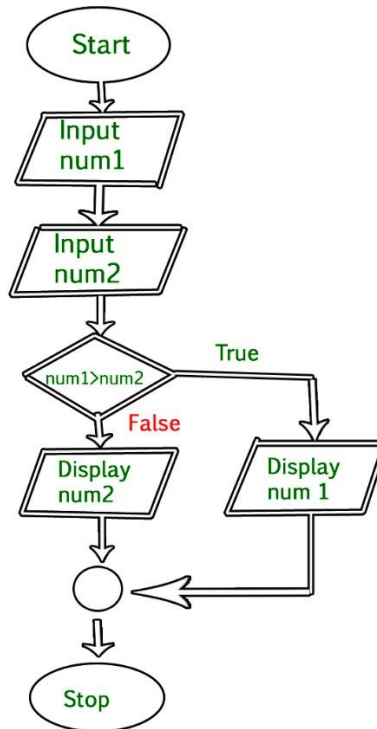


Fig. 1.6: A flowchart to input two numbers from the user and display the largest of two numbers

Conclusion:

Flowcharts are a useful tool for visualizing and documenting complex processes and algorithms. They help to communicate ideas and designs to others, as well as for testing and debugging programs. The components of a flowchart, including start/stop symbols, process symbols, decision symbols, and connector symbols, are used to represent the steps involved in a process or algorithm in a graphical and sequential manner. By using an example, we have demonstrated how flowcharts can be used to represent a simple program for calculating the total cost of a purchase.

1.3 Representation of algorithms

Introduction:

An algorithm is a set of well-defined steps used to solve a problem or accomplish a task. An algorithm can be thought of as a recipe or a set of instructions that can be followed to achieve a desired outcome. Algorithms are used in computer programming to solve complex problems and accomplish tasks in a

systematic and efficient manner. In this lecture, we will explore algorithms in more detail and use an example to demonstrate their use.

Characteristics of a Good Algorithm:

A good algorithm should have the following characteristics:

- **Well-defined:** The steps in the algorithm should be well-defined and clear, with no ambiguity or vagueness.
- **Feasible:** The algorithm should be able to be executed with the resources available, such as time, memory, and processing power.
- **Efficient:** The algorithm should be able to solve the problem in an acceptable amount of time, with a reasonable use of resources.
- **Optimal:** The algorithm should provide the best possible solution for the problem, given the constraints and available resources.
- **Verifiable:** The algorithm should have a clear way of verifying its solution, with a clear and well-defined output.

Example:

To demonstrate the use of algorithms, let's consider a simple example of a program that finds the maximum value in a list of numbers. The algorithm for this program would look something like this:

1. Start
2. Input a list of numbers
3. Set the maximum value to the first number in the list
4. For each number in the list, starting from the second number:
 - a) If the current number is greater than the maximum value, set the maximum value to the current number
5. Output the maximum value
6. Stop

This algorithm provides a set of well-defined steps for finding the maximum value in a list of numbers. The flow of control starts with the start step and moves to step 2, where the list of numbers is input. In step 3, the maximum value is set to the first number in the list. Step 4 is a loop that iterates over each number in the list, starting from the second number. In each iteration of the loop, the algorithm checks if the current number is greater than the maximum value, and if so, sets the maximum value to the current number. Finally, in step 5, the maximum value is output and the algorithm stops.

Conclusion:

Algorithms are a fundamental tool for solving problems and accomplishing tasks in computer programming. A good algorithm should be well-defined, feasible, efficient, optimal, and verifiable. By using an example, we have demonstrated how algorithms can be used to represent a simple program for finding the maximum value in a list of numbers. The use of algorithms is essential for designing and implementing efficient and effective solutions to complex problems in computer science and beyond.

1.4 Program debugging

Introduction:

Debugging is the process of finding and fixing errors in a program. Debugging is an important part of the software development process and is crucial for ensuring the program runs correctly and meets the desired requirements. In this lecture, we will explore the basics of debugging and the steps involved in debugging a program.

Types of Errors:

There are two main types of errors in a program: syntax errors and logic errors.

- **Syntax Errors:** These are errors in the way the code is written and include mistakes such as misspelled keywords, missing punctuation, or incorrect indentation. Syntax errors prevent the program from running, and they are usually caught by the compiler or interpreter.
- **Logic Errors:** These are errors in the way the code is written that do not prevent the program from running but produce incorrect results. Logic errors are more difficult to detect than syntax errors and require a careful examination of the code and the output.

Debugging Techniques:

There are several debugging techniques that can be used to find and fix errors in a program, including:

- **Print Statements:** Adding print statements to the code is a simple way to observe the behavior of the program and check the values of variables as the program runs.
- **Debugging Tools:** Many programming environments provide debugging tools that allow you to step through the code, set breakpoints, and examine the values of variables.
- **Test Cases:** Writing test cases is a good way to verify the behavior of the program and can help identify logic errors.
- **Code Reviews:** Having another person review your code can be a valuable way to find errors and improve the overall quality of the code.
- **Rubber Duck Debugging:** This technique involves explaining the code to a rubber duck or another object as if you were explaining it to another person. The process of explaining the code can help identify errors and improve the understanding of the code.

Debugging Process:

The debugging process typically involves the following steps:

- **Reproduce the error:** This step involves creating a test case that consistently produces the error.
- **Identify the source of the error:** This step involves examining the code and output to determine the source of the error.
- **Plan a solution:** This step involves determining a plan for fixing the error, such as correcting the code or changing the approach.
- **Implement the solution:** This step involves making the necessary changes to the code.
- **Test the solution:** This step involves testing the program with the test case to verify that the error has been fixed.
- **Repeat the process:** This step may be necessary if the solution does not solve the problem or if the solution creates new errors.

Conclusion:

Debugging is an essential part of the software development process and is crucial for ensuring the program runs correctly and meets the desired requirements. There are several debugging techniques that can be used

to find and fix errors in a program, including print statements, debugging tools, test cases, code reviews, and rubber duck debugging. The debugging process typically involves reproducing the error, identifying the source of the error, planning a solution, implementing the solution, testing the solution, and repeating the process as necessary. By following these steps, you can find and fix errors in your code and produce a high-quality program.

1.5 Basic program structure

The basic structure of a C program is divided into 6 parts which makes it easy to read, modify, document, and understand in a particular format. C program must follow the below-mentioned outline in order to successfully compile and execute. Debugging is easier in a well-structured C program.

Sections of the C Program:

There are 6 basic sections responsible for the proper execution of a program. Sections are mentioned below:

1. Documentation
2. Preprocessor Section
3. Definition
4. Global Declaration
5. `main()` Function
6. Sub Programs

1. Documentation

This section consists of the description of the program, the name of the program, and the creation date and time of the program. It is specified at the start of the program in the form of comments. Documentation can be represented as:

```
// description, name of the program, programmer name, date, time etc.
```

Or

```
/*  
description, name of the program, programmer name, date, time etc.  
*/
```

Anything written as comments will be treated as documentation of the program and this will not interfere with the given code. Basically, it gives an overview to the reader of the program.

2. Preprocessor Section

All the header files of the program will be declared in the preprocessor section of the program. Header files help us to access other's improved code into our code. A copy of these multiple files is inserted into our program before the process of compilation.

Example:

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

3. Definition

Preprocessors are the programs that process our source code before the process of compilation. There are multiple steps which are involved in the writing and execution of the program. Preprocessor directives start with the '#' symbol. The `#define` preprocessor is used to create a constant throughout the program. Whenever this name is encountered by the compiler, it is replaced by the actual piece of defined code.

Example:

```
#define long long ll
```

4. Global Declaration

The global declaration section contains global variables, function declaration, and static variables. Variables and functions which are declared in this scope can be used anywhere in the program.

Example:

```
int num = 18;
```

5. `main()` Function

Every C program must have a main function. The `main()` function of the program is written in this section. Operations like declaration and execution are performed inside the curly braces of the main program. The return type of the `main()` function can be int as well as void too. `void()` main tells the compiler that the program will not return any value. The `int main()` tells the compiler that the program will return an integer value.

Example:

```
void main()
```

or

```
int main()
```

6. Sub Programs

User-defined functions are called in this section of the program. The control of the program is shifted to the called function whenever they are called from the main or outside the `main()` function. These are specified as per the requirements of the programmer.

Example:

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

Structure of C Program with example

Example: Below C program to find the sum of 2 numbers:

```
// Documentation
/**
 * file: sum.c
```

```

* author: you
* description: program to find sum.
*/

// Link
#include <stdio.h>

// Definition
#define X 20

// Global Declaration
int sum(int y);

// Main() Function
int main(void)
{
    int y = 55;
    printf("Sum: %d", sum(y));
    return 0;
}

// Subprogram
int sum(int y)
{
    return y + X;
}

```

Output

Sum: 75

Explanation of the above Program

Below is the explanation of the above program. With a description explaining the program's meaning and use.

Sections	Description
<pre> /** *file: sum.c *author: you *description: program to find sum. */ </pre>	<p>It is the comment section and is part of the description section of the code.</p>
<pre> #include<stdio.h> </pre>	<p>Header file which is used for standard input-output. This is the preprocessor section.</p>

Sections	Description
<code>#define X 20</code>	This is the definition section. It allows the use of constant <code>X</code> in the code.
<code>int sum(int y)</code>	This is the global declaration section includes the function declaration that can be used anywhere in the program.
<code>int main()</code>	<code>main()</code> is the first function that is executed in the C program.
<code>{...}</code>	These curly braces mark the beginning and end of the main function.
<code>printf("Sum: %d", sum(y));</code>	<code>printf()</code> function is used to print the sum on the screen.
<code>return 0;</code>	We have used <code>int</code> as the return type so we have to return 0 which states that the given program is free from the error and it can be exited successfully.
<code>int sum(int y) { return y + X; }</code>	This is the subprogram section. It includes the user-defined functions that are called in the <code>main()</code> function.

Steps involved in the Compilation and execution of a C program:

- Program Creation
- Compilation of the program
- Execution of the program
- The output of the program

Chapter 2

Program Structure

2.1 Identifiers and their rules:

In C programming language, identifiers are the building blocks of a program. Identifiers are unique names that are assigned to variables, structs, functions, and other entities. They are used to uniquely identify the entity within the program. In the below example “section” is an identifier assigned to the string type value.

```
char section = 'A';
```

For the naming of identifiers, we have a set of rules in C to be followed for valid identifier names.

Rules to Name an Identifier in C

A programmer has to follow certain rules while naming variables. For the valid identifier, we must follow the given below set of rules.

1. An identifier can include letters (a-z or A-Z), and digits (0-9).
2. An identifier cannot include special characters except the ‘_’ underscore.
3. Spaces are not allowed while naming an identifier.
4. An identifier can only begin with an underscore or letters.
5. We cannot name identifiers the same as keywords because they are reserved words to perform a specific task. For example, printf, scanf, int, char, struct, etc. If we use a keyword’s name as an identifier the compiler will throw an error.
6. The identifier must be **unique** in its namespace.
7. C language is case-sensitive so, ‘name’ and ‘NAME’ are different identifiers.

The below image shows some **valid** and **invalid** identifiers in C language.

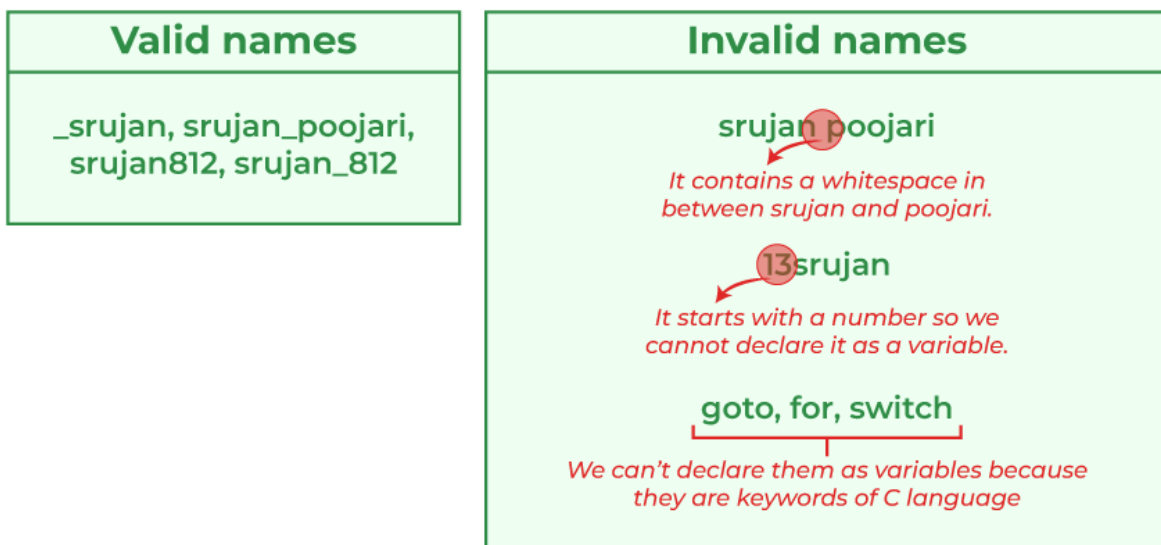


Fig. 2.1: Valid and Invalid Identifiers in C

Examples of Identifiers in C

In the below code, identifiers are named by following all the rules for valid identifiers. We use identifiers to name a structure, function, character data type, double data type, etc. If you are not aware of structures and functions in C then don't worry you will learn them soon. The below code ran successfully without throwing any errors as we have followed all the rules.

```
// C program to illustrate the identifiers
#include <stdio.h>
// here student identifier is used to refer the below structure
struct _student {
    int id;
    int class;
    char section;
};

// isEven identifier is used to call the below function
void isEven(int num)
{
    if(num%2==0){
        printf("It is an Even Number");
    }
    else{
        printf("It is not an Even Number");
    }
}

void main()
{
    // identifiers used as variable names.
    int studentAge = 20;
    double Marks = 349.50;

    // Calling isEven function.
    isEven(453);
}
```

Output

```
It is not an Even Number
```

What happens if we use a keyword as an Identifier in C?

In the below code, we have used const as an identifier which is a keyword in C. This will result in an error in the output.

```
#include <stdio.h>

int main() {
```

```

// used keyword as an identifier
int const = 90;

return 0;
}

```

Output

```

./Solution.c: In function 'main':
./Solution.c:5:14: error: expected identifier or '(' before '=' token
    int const = 90;
                ^

```

2.2 Keywords

In C Programming language, there are many rules to avoid different types of errors. One of such rules is not able to declare variable names with `auto`, `long`, etc. This is all because these are keywords. Let us check all keywords in C language.

Keywords are predefined or reserved words that have special meanings to the compiler. These are part of the syntax and cannot be used as identifiers in the program. A list of keywords in C or reserved words in the C programming language are mentioned below:

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

2.3 Constants

The constants in C are the read-only variables whose values cannot be modified once they are declared in the C program. The type of constant can be an integer constant, a floating pointer constant, a string constant, or a character constant. In C language, the `const` keyword is used to define the constants.

What is a constant in C?

As the name suggests, a constant in C is a variable that cannot be modified once it is declared in the program. We cannot make any change in the value of the constant variables after they are defined.

How to Define Constant in C?

We define a constant in C language using the `const` keyword. Also known as a `const` type qualifier, the `const` keyword is placed at the start of the variable declaration to declare that variable as a constant.

Syntax to Define Constant

```
const data_type var_name = value;
```

Example of Constants in C

```
// C program to illustrate constant variable definition
#include <stdio.h>
int main()
{
    // defining integer constant using const keyword
    const int int_const = 25;

    // defining character constant using const keyword
    const char char_const = 'A';

    // defining float constant using const keyword
    const float float_const = 15.66;

    printf("Printing value of Integer Constant: %d\n",
           int_const);
    printf("Printing value of Character Constant: %c\n",
           char_const);
    printf("Printing value of Float Constant: %f",
           float_const);

    return 0;
}
```

Output

```
Printing value of Integer Constant: 25
Printing value of Character Constant: A
Printing value of Float Constant: 15.660000
```

One thing to note here is that we must **initialize the constant variables at declaration**. Otherwise, the variable will store some garbage value, and we won't be able to change it. The following image describes examples of incorrect and correct variable definitions.

How to Declare Constants

```
const int var;      ✗
const int var;
var=5               ✗
Const int var = 5;  ✓
```

Fig. 2.2 How to declare C constants

Types of Constants in C

The type of the constant is the same as the data type of the variables. Following is the list of the types of constants

- Integer Constant
- Character Constant
- Floating Point Constant
- Double Precision Floating Point Constant
- Array Constant
- Structure Constant

We just have to add the const keyword at the start of the variable declaration.

Properties of Constant in C

The important properties of constant variables in C defined using the const keyword are as follows:

1. Initialization with Declaration

We can only initialize the constant variable in C at the time of its declaration. Otherwise, it will store the garbage value.

2. Immutability

The constant variables in c are immutable after its definition, i.e., they can be initialized only once in the whole program. After that, we cannot modify the value stored inside that variable.

```
// C Program to demonstrate the behaviour of constant variable
#include <stdio.h>
int main()
{
    // declaring a constant variable
    const int var;
    // initializing constant variable var after declaration
    var = 20;

    printf("Value of var: %d", var);
    return 0;
}
```

Output

```
In function 'main':
10:9: error: assignment of read-only variable 'var'
10 |     var = 20;
    |         ^
```

Difference Between Constants and Literals

The constant and literals are often confused as the same. But in C language, they are different entities and have different semantics. The following table lists the differences between the constants and literals in C:

Constant	Literals
Constants are variables that cannot be modified once declared.	Literals are the fixed values that define themselves.
Constants are defined by using the const keyword in C. They store literal values in themselves.	They themselves are the values that are assigned to the variables or constants.
We can determine the address of constants.	We cannot determine the address of a literal except string literal.
They are lvalues.	They are rvalues.
Example: <code>const int c = 20.</code>	Example: <code>24,15.5, 'a', "Geeks",</code> etc.

Defining Constant using `#define` Preprocessor

We can also define a constant in C using `#define` preprocessor. The constants defined using `#define` are macros that behave like a constant. These constants are not handled by the compiler, they are handled by the preprocessor and are replaced by their value before compilation.

```
#define const_name value
```

Example of Constant Macro

```
// C Program to define a constant using #define
#include <stdio.h>
#define pi 3.14
int main()
{
    printf("The value of pi: %.2f", pi);
    return 0;
}
```

Output

```
The value of pi: 3.14
```

Note: This method for defining constant is not preferred as it may introduce bugs and make the code difficult to maintain.

2.4 Variables

A **variable in C language** is the name associated with some memory location to store data of different types. There are many types of variables in C depending on the scope, storage class, lifetime, type of data they store, etc. A variable is the basic building block of a C program that can be used in expressions as a substitute in place of the value it stores.

What is a variable in C?

*A **variable in C** is a memory location with some name that helps store some form of data and retrieves it when required. We can store different types of data in the variable and reuse the same variable for storing some other data any number of times.*

They can be viewed as the names given to the memory location so that we can refer to it without having to memorize the memory address. The size of the variable depends upon the data type it stores.

C Variable Syntax

The syntax to declare a variable in C specifies the name and the type of the variable.

```
data_type variable_name = value; // defining single variable
```

or

```
data_type variable_name1, variable_name2; // defining multiple variables
```

Here,

- **data_type:** Type of data that a variable can store.
- **variable_name:** Name of the variable given by the user.
- **value:** value assigned to the variable by the user.

Example

```
int var; // integer variable
```

```
char a; // character variable
```

```
float fff; // float variables
```

Note: C is a strongly typed language so all the variables types must be specified before using them.

There are 3 aspects of defining a variable:

1. Variable Declaration
2. Variable Definition
3. Variable Initialization

1. C Variable Declaration

Variable declaration in C tells the compiler about the existence of the variable with the given name and data type. When the variable is declared, an entry in symbol table is created and memory will be allocated at the time of initialization of the variable.

2. C Variable Definition

In the definition of a C variable, the compiler allocates some memory and some value to it. A defined variable will contain some random garbage value till it is not initialized.

Example

```
int var;
```

```
char var2;
```

Note: Most of the modern C compilers declare and define the variable in single step. Although we can declare a variable in C by using extern keyword, it is not required in most of the cases.

3. C Variable Initialization

Initialization of a variable is the process where the user assigns some meaningful value to the variable when creating the variable.

Example

```
int var = 10;    // variable declaration and definition
                //(i.e. Variable Initialization)
```

Difference between Variable Initialization and Assignment

Initialization occurs when a variable is first declared and assigned an initial value. This usually happens during the declaration of the variable. On the other hand, assignment involves setting or updating the value of an already declared variable, and this can happen multiple times after the initial initialization.

Example

```
int a=10;       //Variable initialization
a=10;          //assignment
```

How to use variables in C?

The below example demonstrates how we can use variables in C language.

```
// C program to demonstrate the declaration, definition and
// initialization
#include <stdio.h>
int main()
{
    // declaration with definition
    int defined_var;

    printf("Defined_var: %d\n", defined_var);

    // assignment
    defined_var = 12;

    // declaration + definition + initialization
    int ini_var = 25;

    printf("Value of defined_var after assignment: %d\n", defined_var);
```



```
printf("Value of ini_var: %d", ini_var);  
  
return 0;  
}
```

Output

```
Defined_var: 0  
Value of defined_var after assignment: 12  
Value of ini_var: 25
```

C Variable Types

The C variables can be classified into the following types:

1. **Local Variables**
2. **Global Variables**
3. **Static Variables**
4. **Automatic Variables**
5. **Extern Variables**
6. **Register Variables**

1. Local Variables in C

A **Local variable in C** is a variable that is declared inside a function or a block of code. Its scope is limited to the block or function in which it is declared.

Example of Local Variable in C

```
// C program to declare and print local variable inside a function.  
#include <stdio.h>  
void function()  
{  
    int x = 10; // local variable  
    printf("%d", x);  
}  
  
int main() { function(); }
```

Output

```
10
```

In the above code, **x** can be used only in the scope of **function()**. Using it in the **main** function will give an error.

2. Global Variables in C

A **Global variable in C** is a variable that is declared outside the function or a block of code. Its scope is the whole program, i.e., we can access the global variable anywhere in the C program after it is declared.

Example of Global Variable in C

```
// C program to demonstrate use of global variable
```

```

#include <stdio.h>
int x = 20; // global variable

void function1() { printf("Function 1: %d\n", x); }

void function2() { printf("Function 2: %d\n", x); }

int main()
{
    function1();
    function2();
    return 0;
}

```

Output

```

Function 1: 20
Function 2: 20

```

In the above code, both functions can use the global variable as global variables are accessible by all the functions.

Note: When we have same name for local and global variable, local variable will be given preference over the global variable by the compiler.

3. Static Variables in C

A **static variable in C** is a variable that is defined using the **static** keyword. It can be defined only once in a C program and its scope depends upon the region where it is declared (can be **global or local**). The **default value** of static variables is **zero**.

Syntax of Static Variable in C

```

static data_type variable_name = initial_value;

```

As its lifetime is till the end of the program, it can retain its value for multiple function calls as shown in the example.

Example of Static Variable in C

```

// C program to demonstrate use of static variable
#include <stdio.h>

void function()
{
    int x = 20; // local variable
    static int y = 30; // static variable
    x = x + 10;
    y = y + 10;
    printf("\tLocal: %d\n\tStatic: %d\n", x, y);
}

```

```

int main()
{
    printf("First Call\n");
    function();
    printf("Second Call\n");
    function();
    printf("Third Call\n");
    function();
    return 0;
}

```

Output

```

First Call
    Local: 30
    Static: 40
Second Call
    Local: 30
    Static: 50
Third Call
    Local: 30
    Static: 60

```

In the above example, we can see that the local variable will always print the same value whenever the function will be called whereas the static variable will print the incremented value in each function call.

4. Automatic Variable in C

All the **local** variables are **automatic** variables **by default**. They are also known as auto variables. Their scope is **local** and their lifetime is till the end of the **block**. If we need, we can use the **auto** keyword to define the auto variables. The default value of the auto variables is a garbage value.

Syntax of Auto Variable in C

```

auto data_type variable_name;

```

or

```

data_type variable_name; // in local scope

```

Example of auto Variable in C

```

// C program to demonstrate use of automatic variable
#include <stdio.h>
void function()
{
    int x = 10; // local variable (also automatic)
    auto int y = 20; // automatic variable
    printf("Auto Variable: %d", y);
}
int main()

```

```
{
    function();
    return 0;
}
```

Output

```
Auto Variable: 20
```

In the above example, both **x** and **y** are automatic variables. The only difference is that variable **y** is explicitly declared with the **auto** keyword.

5. External Variables in C

External variables in C can be **shared** between **multiple C files**. We can declare an external variable using the **extern** keyword. Their scope is **global** and they exist between multiple C files.

Syntax of Extern Variables in C

```
extern data_type variable_name;
```

Example of Extern Variable in C

```
-----myfile.h-----
extern int x=10; //external variable (also global)

-----program1.c-----
#include "myfile.h"
#include <stdio.h>
void printValue(){
printf("Global variable: %d", x);
}
```

In the above example, **x** is an external variable that is used in multiple C files.

6. Register Variables in C

Register variables in C are those variables that are stored in the **CPU register** instead of the conventional storage place like RAM. Their scope is **local** and exists till the **end of the block** or a function. These variables are declared using the **register** keyword. The default value of register variables is a **garbage value**.

Syntax of Register Variables in C

```
register data_type variable_name = initial_value;
```

Example of Register Variables in C

```
// C program to demonstrate the definition of register variable
#include <stdio.h>
int main()
{
```

```

//    register variable
register int var = 22;

printf("Value of Register Variable: %d\n", var);
return 0;
}

```

Output

```
Value of Register Variable: 22
```

NOTE: We cannot get the address of the register variable using `addressof (&)` operator because they are stored in the CPU register. The compiler will throw an error if we try to get the address of register variable.

2.5 Comments

The **comments in C** are human-readable explanations or notes in the source code of a C program. A comment makes the program easier to read and understand. These are the statements that are not executed by the compiler or an interpreter.

It is considered to be a good practice to document our code using comments.

When and Why to use Comments in C programming?

- A person reading a large code will be bemused if comments are not provided about details of the program.
- C Comments are a way to make a code more readable by providing more descriptions.
- C Comments can include a description of an algorithm to make code understandable.
- C Comments can be used to prevent the execution of some parts of the code.

Types of comments in C

In C there are two types of comments in C language:

- **Single-line comment**
- **Multi-line comment**

Types of Comments in C

1. Single-line Comment in C

A single-line comment in C starts with (//) double forward slash. It extends till the end of the line and we don't need to specify its end.

Syntax of Single Line C Comment

```
// This is a single line comment
```

Example 1: C Program to illustrate single-line comment

```
// C program to illustrate
// use of single-line comment
```

```

#include <stdio.h>
int main(void)
{
    // This is a single-line comment
    printf("Welcome");
    return 0;
}

```

Output:

Welcome

Comment at End of Code Line

We can also create a comment that displays at the end of a line of code using a single-line comment. But generally, it's better to practice putting the comment before the line of code.

```

// C program to demonstrate commenting after line of code
#include <stdio.h>
int main() {
    // single line comment here

    printf("Welcome"); // comment here
    return 0;
}

```

Output

Welcome

2. Multi-line Comment in C

The Multi-line comment in C starts with a forward slash and asterisk (/*) and ends with an asterisk and forward slash (*/). Any text between /* and */ is treated as a comment and is ignored by the compiler.

It can apply comments to multiple lines in the program.

Syntax of Multi-Line C Comment

```

/*Comment starts
    continues
    continues
    .
    .
    .
Comment ends*/

```

Example 2: C Program to illustrate the multi-line comment

```

/* C program to illustrate
use of
multi-line comment */
#include <stdio.h>

```

```

int main(void)
{
    /*
    This is a
    multi-line comment
    */

    /*
    This comment contains some code which
    will not be executed.
    printf("Code enclosed in Comment");
    */
    printf("Welcome");
    return 0;
}

```

Output:

Welcome

2.6 Data Types

Each variable in C has an associated data type. It specifies the type of data that the variable can store like integer, character, floating, double, etc. Each data type requires different amounts of memory and has some specific operations which can be performed over it. The data type is a collection of data with values having fixed values, meaning as well as its characteristics.

The data types in C can be classified as follows:

Types	Description
Primitive Data Types	Primitive data types are the most basic data types that are used for representing simple values such as integers, float, characters, etc.
User Defined Data Types	The user-defined data types are defined by the user himself.
Derived Types	The data types that are derived from the primitive or built-in datatypes are referred to as Derived Data Types.

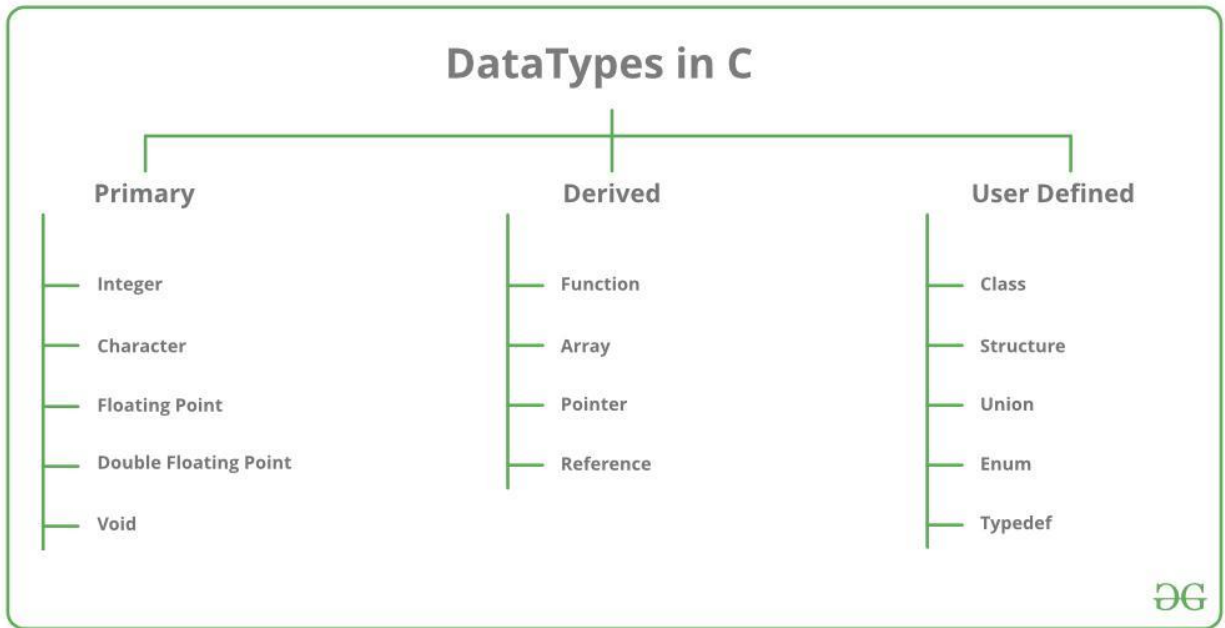


Fig 2.3: Data types in C

Different data types also have different ranges up to which they can store numbers. These ranges may vary from compiler to compiler. Below is a list of ranges along with the memory requirement and format specifiers on the *32-bit GCC compiler*.

Data Type	Size (bytes)	Range	Format Specifier
short int	2	-32,768 to 32,767	%hd
unsigned short int	2	0 to 65,535	%hu
unsigned int	4	0 to 4,294,967,295	%u
int	4	-2,147,483,648 to 2,147,483,647	%d
long int	4	-2,147,483,648 to 2,147,483,647	%ld
unsigned long int	4	0 to 4,294,967,295	%lu

Data Type	Size (bytes)	Range	Format Specifier
long long int	8	$-(2^{63})$ to $(2^{63})-1$	%lld
unsigned long long int	8	0 to 18,446,744,073,709,551,615	%llu
signed char	1	-128 to 127	%c
unsigned char	1	0 to 255	%c
float	4	1.2E-38 to 3.4E+38	%f
double	8	1.7E-308 to 1.7E+308	%lf
long double	16	3.4E-932 to 1.1E+4932	%Lf

*Note: The **long**, **short**, **signed** and **unsigned** are datatype modifiers that can be used with some primitive data types to change the size or length of the datatype.*

The following are some main primitive data types in C:

Integer Data Type

The integer datatype in C is used to store the integer numbers (any number including positive, negative and zero without decimal part). Octal values, hexadecimal values, and decimal values can be stored in int data type in C.

- **Range:** -2,147,483,648 to 2,147,483,647
- **Size:** 4 bytes
- **Format Specifier:** %d

Syntax of Integer

We use **int** keyword to declare the integer variable:

```
int var_name;
```

The integer data type can also be used as

- **unsigned int:** **unsigned int** data type in C is used to store the data values from zero to positive numbers but it can't store negative values like **signed int**.

- **short int:** It is lesser in size than the **int** by 2 bytes so can only store values from -32,768 to 32,767.
- **long int:** Larger version of the **int** datatype so can store values greater than **int**.
- **unsigned short int:** Similar in relationship with **short int** as **unsigned int** with **int**.
- *Note:* The size of an integer data type is compiler-dependent. We can use **sizeof** operator to check the actual size of any data type.

Example of **int**

```
// C program to print Integer data types.
#include <stdio.h>
int main()
{
    // Integer value with positive data.
    int a = 9;

    // integer value with negative data.
    int b = -9;

    // U or u is Used for Unsigned int in C.
    int c = 89U;

    // L or l is used for long int in C.
    long int d = 99998L;

    printf("Integer value with positive data: %d\n", a);
    printf("Integer value with negative data: %d\n", b);
    printf("Integer value with an unsigned int data: %u\n",
        c);
    printf("Integer value with a long int data: %ld", d);

    return 0;
}
```

Output

```
Integer value with positive data: 9
Integer value with negative data: -9
Integer value with an unsigned int data: 89
Integer value with a long int data: 99998
```

Character Data Type

Character data type allows its variable to store only a single character. The size of the character is 1 byte. It is the most basic data type in C. It stores a single character and requires a single byte of memory in almost all compilers.

- **Range:** (-128 to 127) or (0 to 255)
- **Size:** 1 byte

- **Format Specifier:** %c

Syntax of char

The `char` keyword is used to declare the variable of character type:

```
char var_name;
```

Example of char

```
// C program to print Integer data types.
#include <stdio.h>
int main()
{
    char a = 'a';
    char c;

    printf("Value of a: %c\n", a);

    a++;
    printf("Value of a after increment is: %c\n", a);

    // c is assigned ASCII values
    // which corresponds to the
    // character 'c'
    // a-->97 b-->98 c-->99
    // here c will be printed
    c = 99;

    printf("Value of c: %c", c);

    return 0;
}
```

Output

```
Value of a: a
Value of a after increment is: b
Value of c: c
```

Float Data Type

In C programming float data type is used to store floating-point values. Float in C is used to store decimal and exponential values. It is used to store decimal numbers (numbers with floating point values) with single precision.

- **Range:** 1.2E-38 to 3.4E+38
- **Size:** 4 bytes
- **Format Specifier:** %f

Syntax of float

The `float` keyword is used to declare the variable as a floating point:

```
float var_name;
```

Example of Float

```
// C Program to demonstrate use
// of Floating types
#include <stdio.h>
int main()
{
    float a = 9.0f;
    float b = 2.5f;

    // 2x10^-4
    float c = 2E-4f;
    printf("%f\n", a);
    printf("%f\n", b);
    printf("%f", c);

    return 0;
}
```

Output

```
9.000000
2.500000
0.000200
```

Double Data Type

A double data type in C is used to store decimal numbers (numbers with floating point values) with double precision. It is used to define numeric values which hold numbers with decimal values in C.

The double data type is basically a precision sort of data type that is capable of holding 64 bits of decimal numbers or floating points. Since double has more precision as compared to that float then it is much more obvious that it occupies twice the memory occupied by the floating-point type. It can easily accommodate about 16 to 17 digits after or before a decimal point.

- **Range:** 1.7E-308 to 1.7E+308
- **Size:** 8 bytes
- **Format Specifier:** %lf

Syntax of Double

The variable can be declared as double precision floating point using the `double` keyword:

```
double var_name;
```

Example of Double

```
// C Program to demonstrate
// use of double data type
```

```

#include <stdio.h>
int main()
{
    double a = 123123123.00;
    double b = 12.293123;
    double c = 2312312312.123123;

    printf("%lf\n", a);

    printf("%lf\n", b);

    printf("%lf", c);

    return 0;
}

```

Output

```

123123123.000000
12.293123
2312312312.123123

```

Void Data Type

The void data type in C is used to specify that no value is present. It does not provide a result value to its caller. It has no values and no operations. It is used to represent nothing. Void is used in multiple ways as function return type, function arguments as void, and pointers to void.

Syntax:

```

// function return type void
void exit(int check);

// Function without any parameter can accept void.
int print(void);

```

Example of Void

```

// C program to demonstrate
// use of void pointers
#include <stdio.h>
int main()
{
    int val = 30;
    void* ptr = &val;
    printf("%d", *(int*)ptr);
    return 0;
}

```

Output

2.7 Operators

An operator in C can be defined as the symbol that helps us to perform some specific mathematical, relational, bitwise, conditional, or logical computations on values and variables. The values and variables used with operators are called operands. So we can say that the operators are the symbols that perform operations on operands.

Operators in C

	Operators	Type
Unary Operator	++, --	Unary Operator
	+, -, *, /, %	Arithmetic Operator
Binary Operator	<, <=, >, >=, ==, !=	Relational Operator
	&&, , !	Logical Operator
	&, , <<, >>, ~, ^	Bitwise Operator
	=, +=, -=, *=, /=, %=	Assignment Operator
Ternary Operator	?:	Ternary or Conditional Operator

Fig 2.4: Different kinds of operators in C

Types of Operators in C

C language provides a wide range of operators that can be classified into 6 types based on their functionality:

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Bitwise Operators
5. Assignment Operators
6. Other Operators

2.7.1 Arithmetic operators: The arithmetic operators are used to perform arithmetic/mathematical operations on operands. There are 8 arithmetic operators in C language:

S. No.	Symbol	Operator	Description	Syntax
1	+	Plus	Adds two numeric values.	a + b
2	-	Minus	Subtracts right operand from left operand.	a - b

S. No.	Symbol	Operator	Description	Syntax
3	*	Multiply	Multiply two numeric values.	a * b
4	/	Divide	Divide two numeric values.	a / b
5	%	Modulus	Returns the remainder after dividing the left operand with the right operand.	a % b
6	+	Unary Plus	Used to specify the positive values.	+a
7	-	Unary Minus	Flips the sign of the value.	-a
8	++	Increment	Increases the value of the operand by 1.	a++
9	--	Decrement	Decreases the value of the operand by 1.	a--

Example:

// C program to illustrate the arithmetic operators

```
#include <stdio.h>
int main()
{
    int a = 25, b = 5;
```

```

// using operators and printing results
printf("a + b = %d\n", a + b);
printf("a - b = %d\n", a - b);
printf("a * b = %d\n", a * b);
printf("a / b = %d\n", a / b);
printf("a % b = %d\n", a % b);
printf("+a = %d\n", +a);
printf("-a = %d\n", -a);
printf("a++ = %d\n", a++);
printf("a-- = %d\n", a--);
return 0;
}

```

Output:

```

a + b = 30
a - b = 20
a * b = 125
a / b = 5
a % b = 0
+a = 25
-a = -25
a++ = 25
a-- = 26

```

2.7.2 Relational operators: The relational operators in C are used for the comparison of the two operands. All these operators are binary operators that return true or false values as the result of comparison.

These are a total of 6 relational operators in C:

S. No.	Symbol	Operator	Description	Syntax
1	<	Less than	Returns true if the left operand is less than the right operand. Else false	a < b
2	>	Greater than	Returns true if the left operand is greater than the	a > b

S. No.	Symbol	Operator	Description	Syntax
			right operand. Else false	
3	<=	Less than or equal to	Returns true if the left operand is less than or equal to the right operand. Else false	a <= b
4	>=	Greater than or equal to	Returns true if the left operand is greater than or equal to right operand. Else false	a >= b
5	==	Equal to	Returns true if both the operands are equal.	a == b
6	!=	Not equal to	Returns true if both the operands are NOT equal.	a != b

Example:

```
// C program to illustrate the relational operators
#include <stdio.h>
int main()
{
    int a = 25, b = 5;
    // using operators and printing results
    printf("a < b : %d\n", a < b);
    printf("a > b : %d\n", a > b);
    printf("a <= b: %d\n", a <= b);
    printf("a >= b: %d\n", a >= b);
    printf("a == b: %d\n", a == b);
    printf("a != b : %d\n", a != b);
}
```

```

    return 0;
}

```

Output:

```

a < b : 0
a > b : 1
a <= b: 0
a >= b: 1
a == b: 0
a != b : 1

```

2.7.3 Logical operators: Logical operators are used to combine two or more conditions/constraints or to complement the evaluation of the original condition in consideration. The result of the operation of a logical operator is a Boolean value either **true** or **false**.

S. No.	Symbol	Operator	Description	Syntax
1	&&	Logical AND	Returns true if both the operands are true.	a && b
2		Logical OR	Returns true if both or any of the operand is true.	a b
3	!	Logical NOT	Returns true if the operand is false.	!a

Example:

```

// C program to illustrate the logical operators
#include <stdio.h>
int main()
{
    int a = 25, b = 5;
    // using operators and printing results
    printf("a && b : %d\n", a && b);
    printf("a || b : %d\n", a || b);
    printf("!a: %d\n", !a);
    return 0;
}

```

Output:

```
a && b : 1  
a || b : 1  
!a : 0
```

2.7.4 Bitwise operators: The Bitwise operators are used to perform bit-level operations on the operands. The operators are first converted to bit-level and then the calculation is performed on the operands. Mathematical operations such as addition, subtraction, multiplication, etc. can be performed at the bit level for faster processing.

There are 6 bitwise operators in C:

S. No.	Symbol	Operator	Description	Syntax
1	&	Bitwise AND	Performs bit-by-bit AND operation and returns the result.	a & b
2		Bitwise OR	Performs bit-by-bit OR operation and returns the result.	a b
3	^	Bitwise XOR	Performs bit-by-bit XOR operation and returns the result.	a ^ b
4	~	Bitwise First Complement	Flips all the set and unset bits on the number.	~a
5	<<	Bitwise Leftshift	Shifts the number in binary form by one place in the operation and returns the result.	a << b

S. No.	Symbol	Operator	Description	Syntax
6	>>	Bitwise Rightshift	Shifts the number in binary form by one place in the operation and returns the result.	a >> b

Example:

```
// C program to illustrate the bitwise operators
#include <stdio.h>
int main()
{
    int a = 25, b = 5;
    // using operators and printing results
    printf("a & b: %d\n", a & b);
    printf("a | b: %d\n", a | b);
    printf("a ^ b: %d\n", a ^ b);
    printf("~a: %d\n", ~a);
    printf("a >> b: %d\n", a >> b);
    printf("a << b: %d\n", a << b);
    return 0;
}
```

Output:

```
a & b: 1
a | b: 29
a ^ b: 28
~a: -26
a >> b: 0
a << b: 800
```

2.7.5 Assignment operators: Assignment operators are used to assign value to a variable. The left side operand of the assignment operator is a variable and the right-side operand of the assignment operator is a value. The value on the right side must be of the same data type as the variable on the left side otherwise the compiler will raise an error.

The assignment operators can be combined with some other operators in C to provide multiple operations using a single operator. These operators are called compound operators.

In C, there are 11 assignment operators:

S. No.	Symbol	Operator	Description	Syntax
1	=	Simple Assignment	Assign the value of the right operand to the left operand.	a = b
2	+=	Plus and assign	Add the right operand and left operand and assign this value to the left operand.	a += b
3	-=	Minus and assign	Subtract the right operand and left operand and assign this value to the left operand.	a -= b
4	*=	Multiply and assign	Multiply the right operand and left operand and assign this value to the left operand.	a *= b
5	/=	Divide and assign	Divide the left operand with the right operand and assign this value to the left operand.	a /= b
6	%=	Modulus and assign	Assign the remainder in the division of left operand with the right operand to the left operand.	a %= b

S. No.	Symbol	Operator	Description	Syntax
7	&=	AND and assign	Performs bitwise AND and assigns this value to the left operand.	a &= b
8	=	OR and assign	Performs bitwise OR and assigns this value to the left operand.	a = b
9	^=	XOR and assign	Performs bitwise XOR and assigns this value to the left operand.	a ^= b
10	>>=	Rightshift and assign	Performs bitwise Rightshift and assign this value to the left operand.	a >>= b
11	<<=	Leftshift and assign	Performs bitwise Leftshift and assign this value to the left operand.	a <<= b

Example:

```
// C program to illustrate the assignment operators
#include <stdio.h>
int main()
{
    int a = 25, b = 5;
    // using operators and printing results
    printf("a = b: %d\n", a = b);
    printf("a += b: %d\n", a += b);
    printf("a -= b: %d\n", a -= b);
    printf("a *= b: %d\n", a *= b);
    printf("a /= b: %d\n", a /= b);
}
```

```

printf("a %= b: %d\n", a %= b);
printf("a &= b: %d\n", a &= b);
printf("a |= b: %d\n", a |= b);
printf("a >>= b: %d\n", a >>= b);
printf("a <<= b: %d\n", a <<= b);
return 0;
}

```

Output:

```

a = b: 5
a += b: 10
a -= b: 5
a *= b: 25
a /= b: 5
a %= b: 0
a &= b: 0
a |= b: 5
a >>= b: 0
a <<= b: 0

```

2.7.6 Other operators: Apart from the above operators, there are some other operators available in C used to perform some specific tasks.

sizeof operator: `sizeof` is much used in the C programming language. It is a compile-time unary operator which can be used to compute the size of its operand. The result of `sizeof` is of the unsigned integral type which is usually denoted by `size_t`. Basically, the `sizeof` operator is used to compute the size of the variable or datatype.

Syntax

```
sizeof (operand)
```

Comma operator (,): The comma operator (represented by the token `,`) is a binary operator that evaluates its first operand and discards the result, it then evaluates the second operand and returns this value (and type). The comma operator has the lowest precedence of any C operator. Comma acts as both operator and separator.

Syntax

```
operand1 , operand2
```

Conditional operator (? :): The conditional operator is the only ternary operator in C++. Here, `Expression1` is the condition to be evaluated.

Syntax

```
Expression1 ? Expression 2 : Expression 3;
```

If the condition (**Expression1**) is *True* then we will execute and return the result of **Expression2** otherwise if the condition (**Expression1**) is *false* then we will execute and return the result of **Expression3**. We may replace the use of **if..else** statements with conditional operators.

dot (.) and arrow (->) operators: Member operators are used to reference individual members of classes, structures, and unions. The dot operator is applied to the actual object. The arrow operator is used with a pointer to an object.

Syntax

```
structure_variable . member;
```

and

```
structure_pointer -> member;
```

Cast operator: Casting operators convert one data type to another. For example, **int(2.2000)** would return **2**. A cast is a special operator that forces one data type to be converted into another. The most general cast supported by most of the C compilers is as follows:

```
[ (type) expression ]
```

Syntax

```
(new_type) operand;
```

addressof (&) and dereference (*) operators: Pointer operator **&** returns the address of a variable. For example **&a**; will give the actual address of the variable.

The pointer operator ***** is a pointer to a variable. For example ***var**; will pointer to a variable **var**.

Example of other C operators

```
// C Program to demonstrate the use of Misc operators
#include <stdio.h>
int main()
{
    // integer variable
    int num = 10;
    int* add_of_num = &num;
    printf("sizeof(num) = %d bytes\n", sizeof(num));
    printf("&num = %p\n", &num);
    printf("*add_of_num = %d\n", *add_of_num);
    printf("(10 < 5) ? 10 : 20 = %d\n", (10 < 5) ? 10 : 20);
    printf("(float)num = %f\n", (float)num);
    return 0;
}
```

Output


```
sizeof(num) = 4 bytes
&num = 0x7ffe2b7bdf8c
*add_of_num = 10
(10 < 5) ? 10 : 20 = 20
(float)num = 10.000000
```

Unary, binary and ternary operators in C

Operators can also be classified into three types on the basis of the number of operands they work on:

1. **Unary Operators:** Operators that work on single operand.
2. **Binary Operators:** Operators that work on two operands.
3. **Ternary Operators:** Operators that work on three operands.

Operator Precedence and Associativity in C

In C, it is very common for an expression or statement to have multiple operators and in this expression, there should be a fixed order or priority of operator evaluation to avoid ambiguity.

Operator Precedence and Associativity is the concept that decides which operator will be evaluated first in the case when there are multiple operators present in an expression.

The below table describes the precedence order and associativity of operators in C. The precedence of the operator decreases from top to bottom.

Precedence	Operator	Description	Associativity
1	()	Parentheses (function call)	left-to-right
	[]	Brackets (array subscript)	left-to-right
	.	Member selection via object name	left-to-right
	->	Member selection via a pointer	left-to-right
	a++, a-	Postfix increment/decrement (a is a variable)	left-to-right
2	++a, -a	Prefix increment/decrement (a is a variable)	right-to-left

Precedence	Operator	Description	Associativity
3	+ , -	Unary plus/minus	right-to-left
	! , ~	Logical negation/bitwise complement	right-to-left
	(type)	Cast (convert value to temporary value of type)	right-to-left
	*	Dereference	right-to-left
	&	Address (of operand)	right-to-left
	sizeof	Determine size in bytes on this implementation	right-to-left
4	* , / , %	Multiplication/division/modulus	left-to-right
5	+ , -	Addition/subtraction	left-to-right
6	<< , >>	Bitwise shift left, Bitwise shift right	left-to-right
	< , <=	Relational less than/less than or equal to	left-to-right
7	> , >=	Relational greater than/greater than or equal to	left-to-right
8	== , !=	Relational is equal to/is not equal to	left-to-right

Precedence	Operator	Description	Associativity
8	&	Bitwise AND	left-to-right
9	^	Bitwise XOR	left-to-right
10		Bitwise OR	left-to-right
11	&&	Logical AND	left-to-right
12		Logical OR	left-to-right
13	?:	Ternary conditional	right-to-left
	=	Assignment	right-to-left
	+= , -=	Addition/subtraction assignment	right-to-left
	*= , /=	Multiplication/division assignment	right-to-left
	%= , &=	Modulus/bitwise AND assignment	right-to-left
14	^= , =	Bitwise exclusive/inclusive OR assignment	right-to-left
	<<= , >>=	Bitwise shift left/right assignment	right-to-left
15	,	expression separator	left-to-right

2.8 Expressions

An expression is a combination of operators, constants and variables. An expression may consist of one or more operands, and zero or more operators to produce a value.

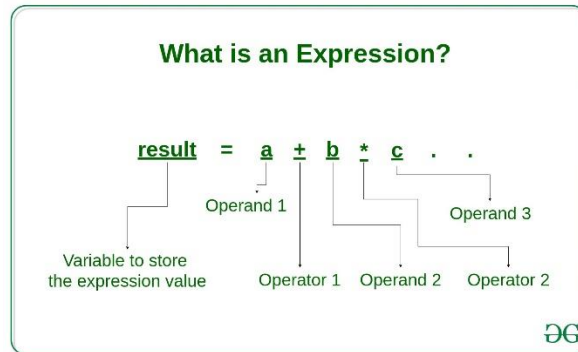


Fig 2.5: A C expression

Expressions may be of the following types:

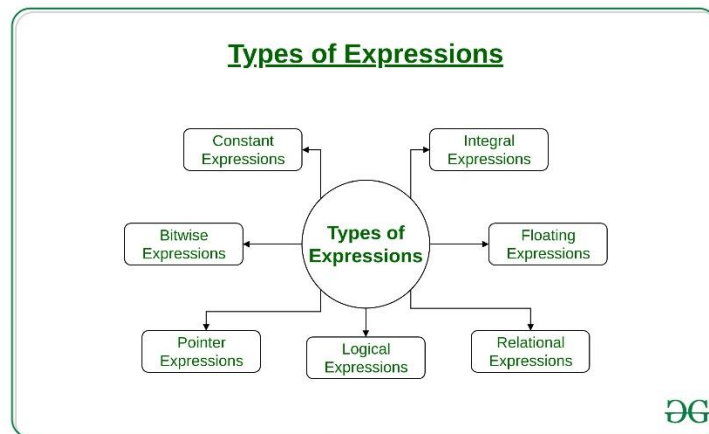


Fig 2.6: Types of expressions in C

- **Constant expressions:** Constant expressions consist of only constant values. A constant value is one that doesn't change.

Examples:

`5, 10 + 5 / 6.0, 'x'`

- **Integral expressions:** Integral expressions are those which produce integer results after implementing all the automatic and explicit type conversions.

Examples:

`x, x * y, x + int(5.0)`

where `x` and `y` are integer variables.

- **Floating expressions:** Float expressions are which produce floating point results after implementing all the automatic and explicit type conversions.

Examples:

`x + y, 10.75`

where `x` and `y` are floating point variables.

- **Relational expressions:** Relational expressions yield results of type `bool` which takes a value true or false. When arithmetic expressions are used on either side of a relational operator, they will be evaluated first and then the results compared. Relational expressions are also known as Boolean expressions.

Examples:

```
x <= y, x + y > 2
```

- **Logical expressions:** Logical expressions combine two or more relational expressions and produce `bool` type results.

Examples:

```
x > y && x == 10, x == 10 || y == 5
```

- **Pointer expressions:** Pointer expressions produce address values.

Examples:

```
&x, ptr, ptr++
```

where `x` is a variable and `ptr` is a pointer.

- **Bitwise expressions:** Bitwise expressions are used to manipulate data at bit level. They are basically used for testing or shifting bits.

Examples:

```
x << 3
```

shifts three bit position to left

```
y >> 1
```

shifts one bit position to right.

Shift operators are often used for multiplication and division by powers of two.

An expression may also use combinations of the above expressions. Such expressions are known as **compound expressions**.

2.9 Type Conversion

Type conversion in C is the process of converting one data type to another. The type conversion is only performed to those data types where conversion is possible. Type conversion is performed by a compiler. In type conversion, the destination data type can't be smaller than the source data type. Type conversion is done at compile time, and it is also called widening conversion because the destination data type can't be smaller than the source data type. *There are two types of Conversion:*

2.9.1 Implicit type conversion

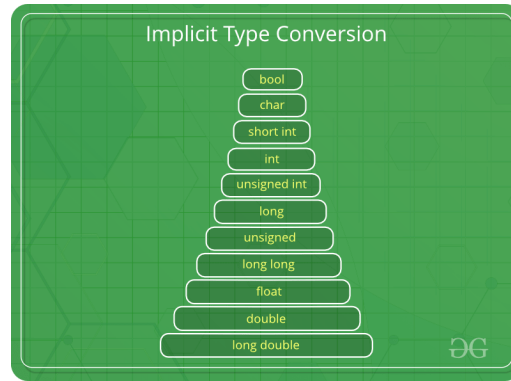


Fig 2.7: Implicit type conversion in C

Implicit type conversion is also called automatic type conversion. Implicit type conversion:

- A. is done by the compiler on its own, without any external trigger from the user.
- B. generally takes place when in an expression more than one data type is present. In such conditions type conversion (type promotion) takes place to avoid loss of data.
- C. upgrades all the data types of the variables to the data type of the variable with the largest data type.

```
bool -> char -> short int -> int ->
unsigned int -> long -> unsigned ->
long long -> float -> double -> long double
```

- D. has the possibility for to lose information, signs can be lost (when signed is implicitly converted to unsigned), and overflow can occur (when long is implicitly converted to float).

Some of its few occurrences are mentioned below:

- Conversion Rank
- Conversions in Assignment Expressions
- Conversion in other Binary Expressions
- Promotion
- Demotion

Example of Type Implicit Conversion

```
#include <stdio.h>
int main()
{
    int x = 10; // integer x
    char y = 'a'; // character c

    // y implicitly converted to int. ASCII
    // value of 'a' is 97
    x = x + y;

    // x is implicitly converted to float
    float z = x + 1.0;

    printf("x = %d, z = %f", x, z);
    return 0;
}
```

```
}
```

Output

```
x = 107, z = 108.000000
```

2.9.2 Explicit type conversion

This process is also called type casting, and it is user-defined. Here the user can typecast the result to make it of a particular data type. Following is the syntax in C programming:

```
(type) expression
```

Type indicates the data type to which the final result is converted.

Example

```
#include<stdio.h>
int main()
{
    double x = 1.2;
    // Explicit conversion from double to int
    int sum = (int)x + 1;
    printf("sum = %d", sum);
    return 0;
}
```

Output

```
sum = 2
```

Advantages of type conversion

- **Type safety:** Type conversions can be used to ensure that data is being stored and processed in the correct data type, avoiding potential type mismatches and type errors.
- **Improved code readability:** By explicitly converting data between different types, you can make the intent of your code clearer and easier to understand.
- **Improved performance:** In some cases, type conversions can be used to optimize the performance of your code by converting data to a more efficient data type for processing.
- **Improved compatibility:** Type conversions can be used to convert data between different types that are not compatible, allowing you to write code that is compatible with a wider range of APIs and libraries.
- **Improved data manipulation:** Type conversions can be used to manipulate data in various ways, such as converting an integer to a string, converting a string to an integer, or converting a floating-point number to an integer.

- **Improved data storage:** Type conversions can be used to store data in a more compact form, such as converting a large integer value to a smaller integer type or converting a large floating-point value to a smaller floating-point type.

Disadvantages of type conversion:

- **Loss of precision:** Converting data from a larger data type to a smaller data type can result in loss of precision, as some of the data may be truncated.
- **Overflow or underflow:** Converting data from a smaller data type to a larger data type can result in overflow or underflow if the value being converted is too large or too small for the new data type.
- **Unexpected behavior:** Type conversions can lead to unexpected behavior, such as when converting between signed and unsigned integer types, or when converting between floating-point and integer types.
- **Confusing syntax:** Type conversions can have confusing syntax, particularly when using typecast operators or type conversion functions, making the code more difficult to read and understand.
- **Increased complexity:** Type conversions can increase the complexity of your code, making it harder to debug and maintain.
- **Slower performance:** Type conversions can sometimes result in slower performance, particularly when converting data between complex data types, such as between structures and arrays.

2.10 Statements (Input/Output and Assignment)

C statements consist of tokens, expressions, and other statements. A statement that forms a component of another statement is called the "body" of the enclosing statement.

2.10.1 Input/output statements

C language has standard libraries that allow input and output in a program. The **stdio.h** or **standard input output library** in C that has methods for input and output.

The **scanf ()** method, in C, reads the value from the console as per the type specified and stores it in the given address.

Syntax:

```
scanf ("%X", &variableOfXType);
```

where **%X** is the format specifier in C. It is a way to tell the compiler what type of data is in a variable and **&** is the address operator in C, which tells the compiler to change the real value of **variableOfXType**, stored at this address in the memory.

The **printf ()** method, in C, prints the value passed as the parameter to it, on the console screen.

Syntax:

```
printf ("%X", variableOfXType);
```

where **%X** is the format specifier in C. It is a way to tell the compiler what type of data is in a variable and **variableOfXType** is the variable to be printed.

How to take input and output of basic types in C?

The basic type in C includes types like int, float, char, etc. In order to input or output the specific type, the **X** in the above syntax is changed with the specific format specifier of that type. The syntax for input and output for these are:

Integer:

Input: `scanf("%d", &intVariable);`

Output: `printf("%d", intVariable);`

Float:

Input: `scanf("%f", &floatVariable);`

Output: `printf("%f", floatVariable);`

Character:

Input: `scanf("%c", &charVariable);`

Output: `printf("%c", charVariable);`

Example:

```
#include <stdio.h>
int main()
{
    // Declare the variables
    int num;
    char ch;
    float f;

    // --- Integer ---

    // Input the integer
    printf("Enter the integer: ");
    scanf("%d", &num);

    // Output the integer
    printf("\nEntered integer is: %d", num);

    // --- Float ---

    //For input Clearing buffer
    while((getchar()) != '\n');

    // Input the float
    printf("\n\nEnter the float: ");
    scanf("%f", &f);
```

```

// Output the float
printf("\nEntered float is: %f", f);

// --- Character ---

// Input the Character
printf("\n\nEnter the Character: ");
scanf("%c", &ch);

// Output the Character
printf("\nEntered character is: %c", ch);

return 0;
}

```

Output

```

Enter the integer: 10
Entered integer is: 10

```

```

Enter the float: 2.5
Entered float is: 2.500000

```

```

Enter the Character: A
Entered Character is: A

```

2.10.2 Assignment statements

Assignment statement allows a variable to hold different types of values during its program lifespan. Another way of understanding an assignment statement is, it stores a value in the memory location which is denoted by a variable name.

Syntax

The symbol used in an assignment statement is called an **operator**. The symbol is '='.

Note: The assignment operator should never be used for equality purpose which is double equal sign '=='.

The basic syntax of assignment statement in a programming language is:

```
variable = expression;
```

where,

variable = variable name

expression = it could be either a direct value or a math expression/formula or a function call.

C requires data type to be specified for the variable, so that it is easy to allocate memory space and store those values during program execution.

Syntax:

```
data_type variable_name = value;
```

Examples:

```
int a = 50;
```

```
float b;
```

```
a = 25;
```

```
b = 34.25f;
```

In the above-given examples, variable 'a' is assigned a value in the same statement as per its defined data type. A data type is only declared for variable 'b'. In the 3rd line of code, variable 'a' is reassigned the value 25. The 4th line of code assigns the value for variable 'b'.

2.11 Use of Header and Library Files

2.11.1 Header Files

In C language, header files contain a set of predefined standard library functions. The .h is the extension of the header files in C and we request to use a header file in our program by including it with the C preprocessing directive "**#include**".

C header files offer the features like library functions, data types, macros, etc., by importing them into the program with the help of a preprocessor directive "**#include**".

Syntax of Header Files in C

We can include header files in C by using one of the given two syntax whether it is a pre-defined or user-defined header file.

```
#include <filename.h> // for files in system/default directory
```

or

```
#include "filename.h" // for files in same directory as source file
```

The "**#include**" preprocessor directs the compiler that the header file needs to be processed before compilation and includes all the necessary data types and function definitions.

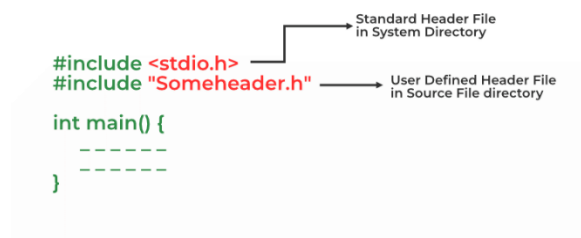


Fig 2.8: Using header files in a C program

Example of Header File in C

The below example demonstrates the use of header files using standard input and output `stdio.h` header file.

```
#include <stdio.h>
int main()
{
    printf(
        "Printf() is the function in stdio.h header file");
    return 0;
}
```

Output

```
Printf() is the function in stdio.h header file
```

Types of C Header Files

There are two types of header files in C:

1. **Standard / Pre-existing header files**
2. **Non-standard / User-defined header files**

1. Standard Header Files in C and Their Uses

Standard header files contain the libraries defined in the ISO standard of the C programming language. They are stored in the default directory of the compiler and are present in all the C compilers from any vendor.

There are 31 standard header files in the latest version of C language. Following is the list of some commonly used header files in C:

Header File	Description
<assert.h>	It contains information for adding diagnostics that aid program debugging.
<errno.h>	It is used to perform error handling operations like <code>errno()</code> , <code>strerror()</code> , <code>perror()</code> , etc.
<float.h>	It contains a set of various platform-dependent constants related to floating point values. These constants are proposed by ANSI C. They make programs more portable. Some examples of constants included in this header file are- <code>e(exponent)</code> , <code>b(base/radix)</code> , etc.

Header File	Description
<math.h>	It is used to perform mathematical operations like sqrt(), log2(), pow(), etc.
<signal.h>	It is used to perform signal handling functions like signal() and raise().
<stdarg.h>	It is used to perform standard argument functions like va_start() and va_arg(). It is also used to indicate start of the variable-length argument list and to fetch the arguments from the variable-length argument list in the program respectively.
<ctype.h>	It contains function prototypes for functions that test characters for certain properties, and also function prototypes for functions that can be used to convert uppercase letters to lowercase letters and vice versa.
<stdio.h>	It is used to perform input and output operations using functions like scanf(), printf(), etc.
<setjump.h>	It contains standard utility functions like malloc(), realloc(), etc. It contains function prototypes for functions that allow bypassing of the usual function call and return sequence.
<string.h>	It is used to perform various functionalities related to string manipulation like strlen(), strcmp(), strcpy(), size(), etc.
<limits.h>	It determines the various properties of the various variable types. The macros defined in this header limits the values of various variable types like char, int, and long. These limits specify that a variable cannot store any value beyond these limits, for example, an unsigned character can store up to a maximum value of 255.

Header File	Description
<time.h>	It is used to perform functions related to date() and time() like setdate() and getdate(). It is also used to modify the system date and get the CPU time respectively.
<stddef.h>	It contains common type definitions used by C for performing calculations.
<locale.h>	It contains function prototypes and other information that enables a program to be modified for the current locale on which it's running. It enables the computer system to handle different conventions for expressing data such as times, dates, or large numbers throughout the world.

Example

The below example demonstrates the use of some commonly used header files in C.

```
// C program to illustrate
// the use of header file
// in C
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Driver Code
int main()
{
    char s1[20] = "12345";
    char s2[10] = "Geeks";
    char s3[10] = "ForGeeks";
    long int res;

    // Find the value of 9^3 using a
    // function in math.h library
    res = pow(9, 3);
    printf("Using math.h, "
           "The value is: %ld\n",
           res);

    // Convert a string to long long int
```

```

// using a function in stdlib.h library
long int a = atol(s1);
printf("Using stdlib.h, the string");
printf(" to long int: %ld\n", a);

// Copy the string s3 into s2 using
// using a function in string.h library
strcpy(s2, s3);
printf("Using string.h, the strings"
      " s2 and s3: %s %s\n",
      s2, s3);
return 0;
}

```

Output

```

Using math.h, The value is: 729
Using stdlib.h, the string to long int: 12345
Using string.h, the strings s2 and s3: ForGeeks ForGeeks

```

2. Non-Standard Header Files in C and Their Uses

Non-standard header files are not part of the language's ISO standard. They are generally all the header files defined by the programmers for purposes like containing custom library functions etc. They are manually installed by the user or may be part of the compiler by some specific vendor.

There are lots of non-standard libraries for C language. Some commonly used non-standard/user-defined header files are listed below:

Header File	Description
<conio.h>	It contains some useful console functions.
<gtk/gtk.h>	It contains GNU's GUI library for C.

Example

The below example demonstrates the use of `conio.h` non-standard header file.

```

#include <stdio.h>
#include <stdio.h>
#include <conio.h>

// Function to display a welcome message
void displayMessage() {

```

```

    printf("Hello! SLIET\n");
}

int main() {
    // Using conio.h functions
    printf("Press any key to print message \n");
    getch(); // Wait for a key press

    // Call the additional function after a key press
    displayMessage();

    return 0;
}

```

Output

```

Press any key to print message
Hello! SLIET

```

Create your own Header File in C

Instead of writing a large and complex code, we can create our own header files and include them in our program to use whenever we want. It enhances code functionality and readability. Below are the steps to create our own header file:

Step 1: Write your own C code and save that file with the “.h” extension. Below is the illustration of the header file:

```

// Function to find the sum of two
// numbers passed
int sumOfTwoNumbers(int a, int b)
{
    return (a + b);
}

```

Step 2: Include your header file with “`#include`” in your C program as shown below:

```

// C++ program to find the sum of two
// numbers using function declared in
// header file
#include "iostream"

// Including header file
#include "sum.h"
using namespace std;

// Driver Code
int main()
{

```



```

// Given two numbers
int a = 13, b = 22;

// Function declared in header
// file to find the sum
printf("Sum is: %d", sumoftwonumbers(a, b));
}

```

Output

```
Sum is: 35
```

Including Multiple Header Files

You can use various header files in a program. When a header file is included twice within a program, the compiler processes the contents of that header file twice. This leads to an error in the program. To eliminate this error, conditional preprocessor directives are used.

Syntax

```

#ifndef HEADER_FILE_NAME
#define HEADER_FILE_NAME
the entire header file
#endif

```

This construct is called wrapper “`#ifndef`”. When the header is included again, the conditional will become false, because `HEADER_FILE_NAME` is defined. The preprocessor will skip over the entire file contents, and the compiler will not see it twice.

Sometimes it's essential to include several diverse header files based on the requirements of the program. For this, multiple conditionals are used.

Syntax

```

#if SYSTEM_ONE
#include "system1.h"
#elif SYSTEM_TWO
#include "system2.h"
#elif SYSTEM_THREE
....
#endif

```

2.11.2 Library Files

The Standard Function Library in C is a huge library of sub-libraries, each of which contains the code for several functions. In order to make use of these libraries, link each library in the broader library through the use of header files. The actual definitions of these functions are stored in separate library files, and declarations in header files. In order to use these functions, we have to include the header file in the program.

Examples:

To perform any operation related to mathematics, it is necessary to include `math.h` header file.

Example 1: `sqrt()`

Syntax:

```
double sqrt(double x)
```

Below is the C program to calculate the square root of any number:

```
// C program to implement
// the above approach
#include <math.h>
#include <stdio.h>
// Driver code
int main()
{
    double number, squareRoot;
    number = 12.5;

    // Computing the square root
    squareRoot = sqrt(number);

    printf("Square root of %.2lf = %.2lf",
           number, squareRoot);
    return 0;
}
```

Output

```
Square root of 12.50 = 3.54
```

Example 2: `pow()`

Syntax:

```
double pow(double x, double y)
```

Below is the C program to calculate the power of any number:

```
// C program to implement
// the above approach
#include <math.h>
#include <stdio.h>
// Driver code
int main()
{
    double base, power, result;
    base = 10.0;
    power = 2.0;
```

```

    // Calculate the result
    result = pow(base, power);
    printf("%.11f^%.11f = %.21f",
           base, power, result);
    return 0;
}

```

Output

```
10.0^2.0 = 100.00
```

The `limits.h` header determines various properties of the various variable types. The macros defined in this header limits the values of various variable types like `char`, `int`, and `long`. Below is the C program to implement the above approach.

```

// C program to implement
// the above approach
#include <limits.h>
#include <stdio.h>

// Driver code
int main()
{
    printf("Number of bits in a byte %d\n",
           CHAR_BIT);
    printf("Minimum value of SIGNED CHAR = %d\n",
           SCHAR_MIN);
    printf("Maximum value of SIGNED CHAR = %d\n",
           SCHAR_MAX);
    printf("Maximum value of UNSIGNED CHAR = %d\n",
           UCHAR_MAX);
    printf("Minimum value of SHORT INT = %d\n",
           SHRT_MIN);
    printf("Maximum value of SHORT INT = %d\n",
           SHRT_MAX);
    printf("Minimum value of INT = %d\n",
           INT_MIN);
    printf("Maximum value of INT = %d\n",
           INT_MAX);
    printf("Minimum value of CHAR = %d\n",
           CHAR_MIN);
    printf("Maximum value of CHAR = %d\n",
           CHAR_MAX);
    printf("Minimum value of LONG = %ld\n",
           LONG_MIN);
    printf("Maximum value of LONG = %ld\n",
           LONG_MAX);
}

```

```
    return (0);  
}
```

Output

```
Number of bits in a byte 8  
Minimum value of SIGNED CHAR = -128  
Maximum value of SIGNED CHAR = 127  
Maximum value of UNSIGNED CHAR = 255  
Minimum value of SHORT INT = -32768  
Maximum value of SHORT INT = 32767  
Minimum value of INT = -2147483648  
Maximum value of INT = 2147483647  
Minimum value of CHAR = -128  
Maximum value of CHAR = 127  
Minimum value of LONG = -9223372036854775808  
Maximum value of LONG = 9223372036854775807
```

`time.h` header file defines the date and time functions. Below is the C program to implement `time()` and `localtime()` functions:

```
// C program to implement  
// the above approach  
#include <stdio.h>  
#include <time.h>  
#define SIZE 256  
  
// Driver code  
int main(void)  
{  
    char buffer[SIZE];  
    time_t curtime;  
    struct tm* loctime;  
  
    // Get the current time.  
    curtime = time(NULL);  
  
    // Convert it to local time  
    // representation.  
    loctime = localtime(&curtime);  
  
    // Print out the date and time  
    // in the standard format.  
    fputs(asctime(loctime), stdout);  
  
    // Print it out  
    strftime(buffer, SIZE,  
             "Today is %A, %B %d.\n",
```

```

        localtime);
    fputs(buffer, stdout);
    strftime(buffer, SIZE,
            "The time is %I:%M %p.\n",
            localtime);
    fputs(buffer, stdout);
    return 0;
}

```

Output

```

Sun May 30 17:27:47 2021
Today is Sunday, May 30.
The time is 05:27 PM.

```

Difference between Header files and Library files

Header Files: The files that tell the compiler how to call some functionality (without knowing how the functionality actually works) are called header files. They contain function prototypes. They also contain data types and constants used with the libraries. We use `#include` to use these header files in programs. These files end with `.h` extension.

Library: Library is the place where the actual functionality is implemented i.e. they contain function body. Libraries have mainly two categories:

- Static
- Shared or Dynamic

Static: Static libraries contain object code linked with an end user application and then they become part of the executable. These libraries are specifically used at *compile time* which means the library should be present in the correct location when user wants to compile his/her C or C++ program. On Windows, they end with `.lib` extension and with `.a` for MacOS.

Shared or Dynamic: These libraries are only required at *run-time* i.e., the user can compile his/her code without using these libraries. In short, these libraries are linked against at compile time to resolve undefined references and then are distributed to the application so that the application can load it at run time. For example, when we open our game folders, we can find many `.dll` (dynamic link libraries) files. As these libraries can be shared by multiple programs, they are also called shared libraries. These files end with `.dll` or `.lib` extensions. In windows they end with `.dll` extension.

Example: `Math.h` is a header file which includes the prototype for function calls like `sqrt()`, `pow()` etc., whereas `libm.lib`, `libmmd.lib`, `libmmd.dll` are some of the math libraries. In simple terms a header file is like a visiting card and libraries are like a real person, so we use visiting card (header file) to reach to the actual person (library).

Let's see the difference between these two in tabular form, so that it can be easily comparable:

Header Files	Library Files
They have the extension .h	They have the extension .lib
They contain function declaration and even macros.	They contain function definitions
They are available inside “include sub directory” which itself is in Turbo compiler.	They are available inside “lib sub directory” which itself is in Turbo compiler.
Header files are human-readable. Since they are in the form of source code.	Library files are non-human readable. Since they are in the form of machine code.
Header files in our program are included by using a command #include which is internally handle by pre-processor.	Library files in our program are included in last stage by special software called as linker.

Chapter 3

Control Structures

3.1 Introduction

A control structure is a block of code that manages the flow of execution in a program. Control structures dictate the order in which statements are executed based on certain conditions.

Types of control structures:

- Sequential Structure
- Selection Structure (Conditional Statements)
- Repetition Structure (Loops)

Let's see each of them in brief:

Sequential Structure: The default structure is where statements are executed one after the other in sequence.

Selection Structure (Conditional Statements): Allows the program to make decisions and execute different blocks of code based on specified conditions. All conditional statements available in C language are mentioned below:

- `if` statement
- `if else`
- `else if` ladder
- Nested `if else`
- Ternary operator
- `switch case`

Repetition Structure (Loops): Allows the execution of a block of code repeatedly as long as a specified condition is true.

Types of loops in C language are mentioned below:

- Entry Controlled Loop: `while` loop, and `for` loop
- Exit Controlled Loop: `do while` loop

3.2 Decision making with `if`, `if...else` and nested `ifs`

3.2.1 `if` statement

The `if` in C is the simplest decision-making statement. It consists of the test condition and `if` block or body. If the given condition is true only then the `if` block will be executed.

Syntax of `if` Statement in C

```
if (condition)
{
```

```

// if body
// Statements to execute if condition is true
}

```

How to use `if` statement in C?

The following examples demonstrate how to use the `if` statement in C:

```

// C Program to demonstrate the syntax of if statement
#include <stdio.h>

int main()
{
    int gfg = 9;
    // if statement with true condition
    if (gfg < 10) {
        printf("%d is less than 10", gfg);
    }

    // if statement with false condition
    if (gfg > 20) {
        printf("%d is greater than 20", gfg);
    }

    return 0;
}

```

Output

```
9 is less than 10
```

How `if` in C works?

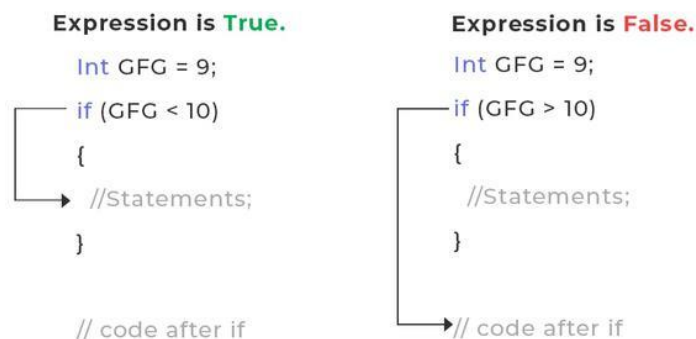


Fig 3.1: Working of `if` Statement in C

The working of the `if` statement in C is as follows:

STEP 1: When the program control comes to the `if` statement, the test expression is evaluated.

- STEP 2A:** If the condition is true, the statements inside the `if` block are executed.
- STEP 2B:** If the expression is false, the statements inside the `if` body are not executed.
- STEP 3:** Program control moves out of the `if` block and the code after the `if` block is executed.

Flowchart of `if` in C

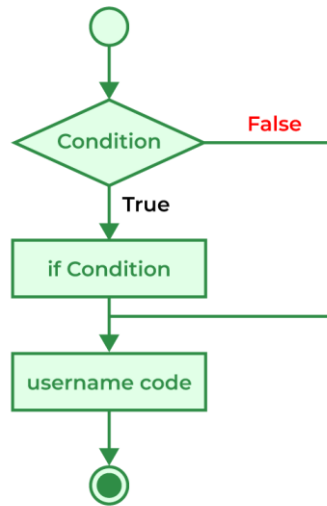


Fig 3.2: Flow Diagram of `if` Statement in C

Examples of `if` Statements in C

Example 1: C Program to check whether the number is even or odd.

In this program, we will make use of the logic that if the number is divisible by 2, then it is even else odd except one.

```

// C Program to check if the number is even or odd
#include <stdio.h>
int main()
{
    int n = 4956;

    // condition to check for even number
    if (n % 2 == 0) {
        printf("%d is Even", n);
    }

    // condition to check for odd number
    else {
        printf("%d is Odd", n);
    }
    return 0;
}
  
```

Output

```
4956 is Even
```

3.2.2 if...else statement

The **if-else** statement in C is a flow control statement used for decision-making in the C program. It is one of the core concepts of C programming. It is an extension of the **if** in C that includes an **else** block along with the already existing **if** block. It is used to decide whether the part of the code will be executed or not based on the specified condition (test expression). If the given condition is true, then the code inside the **if** block is executed, otherwise the code inside the **else** block is executed.

Syntax of if-else

```
if (condition) {  
    // code executed when the condition is true  
}  
else {  
    // code executed when the condition is false  
}
```

The following program demonstrates how to use **if-else** in C:

```
// C Program to demonstrate the use of if-else statement  
#include <stdio.h>  
int main()  
{  
    // if block with condition at the start  
    if (5 < 10) {  
  
        // will be executed if the condition is true  
        printf("5 is less than 10.");  
    }  
  
    // else block after the if block  
    else {  
  
        // will be executed if the condition is false  
        printf("5 is greater that 10.");  
    }  
    return 0;  
}
```

Output

```
5 is less than 10.
```

Note: Any non-zero and non-null values are assumed to be true, and zero or null values are assumed to be false.

How `if-else` Statement works?

Working of the `if-else` statement in C is explained below:

1. When the program control first comes to the `if-else` block, the test condition is checked.
2. If the test condition is `true`:
The `if` block is executed.
3. If the test condition is `false`:
The `else` block is executed
4. After that, the program control continues to the statements below the `if-else` statement.

Flowchart of the `if-else` statement

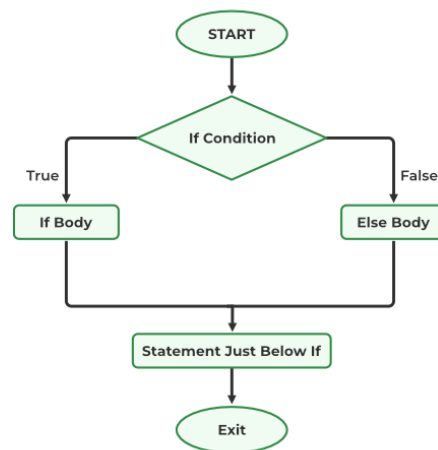


Fig 3.3: Flowchart of `if-else` in C

Examples of `if-else` Statement in C

The following are two basic examples of the `if-else` statement that shows the use of the `if-else` statement in a C program.

Example 1: C Program to check whether a given number is even or odd

For a given number to be even, it should be perfectly divisible by 2. We will use the `if-else` statement to check for this condition and execute different statements for when it is true and when it is false.

```
// C Program to Demonstrate the working of if-else statement
#include <stdio.h>
int main()
{
    // Some random number
    int num = 9911234;

    // checking the condition at the start of if block
    if (num % 2 == 0) {
        // executed when the number is even
```

```

        printf("Number is even");
    }
    // else block
    else {
        // executed when the number is odd
        printf("Number is Odd");
    }
    return 0;
}

```

Output

```
Number is even
```

Example 2: C Program to check whether a person is eligible to vote or not.

We know that a person is eligible to vote after he/she is at least 18 years old. Now we use this condition in the **if-else** statement to check the eligibility of the person.

```

// C Program to check whether the person is eligible to vote
// or not
#include <stdio.h>
int main()
{
    // declaring age of two person
    int p1_age = 15;
    int p2_age = 25;

    // checking eligibility of person 1
    if (p1_age < 18)
        printf("Person 1 is not eligible to vote.\n");
    else
        printf("Person 1 is eligible to vote.\n");

    // checking eligiblity of person 2
    if (p2_age < 18)
        printf("Person 2 is not eligible to vote.\n");
    else
        printf("Person 2 is eligible to vote.");

    return 0;
}

```

Output

```
Person 1 is not eligible to vote.
Person 2 is eligible to vote.
```

You may notice that in the second example, we did not enclose the body of the **if** and **else** statement in the braces and still the code is running without error. This is because the C language allows the skipping of the braces around the body of the **if-else** statement when there is only one statement in the body.

3.2.3 Nested ifs

A nested `if` in C is an `if` statement that is the target of another `if` statement. Nested `if` statements mean an `if` statement inside another `if` statement.

Syntax of Nested `if-else`

```
if (condition1)
{
    // Executes when condition1 is true
    if (condition_2)
    {
        // statement 1
    }
    else
    {
        // Statement 2
    }
}
else {
    if (condition_3)
    {
        // statement 3
    }
    else
    {
        // Statement 4
    }
}
```

Example of Nested `if-else`

```
// C program to illustrate nested-if statement
#include <stdio.h>
int main()
{
    int i = 10;

    if (i == 10) {
        // First if statement
        if (i < 15)
            printf("i is smaller than 15\n");

        // Nested - if statement
        // Will only be executed if statement above
        // is true
        if (i < 12)
            printf("i is smaller than 12 too\n");
        else
            printf("i is greater than 15");
    }
}
```

```

    }
    else {
        if (i == 20) {

            // Nested - if statement
            // Will only be executed if statement above
            // is true
            if (i < 22)
                printf("i is smaller than 22 too\n");
            else
                printf("i is greater than 25");
        }
    }

    return 0;
}

```

Output

```

i is smaller than 15
i is smaller than 12 too

```

3.2.4 if-else-if ladder

The **if-else-if** statements are used when the user has to decide among multiple options. The C **if** statements are executed from the top down. As soon as one of the conditions controlling the **if** is true, the statement associated with that **if** is executed, and the rest of the C **else-if** ladder is bypassed. If none of the conditions is true, then the final **else** statement will be executed. **if-else-if** ladder is similar to the **switch** statement.

Syntax of if-else-if Ladder

```

if (condition)
    statement;
else if (condition)
    statement;
.
.
else
    statement;

```

Flowchart of if-else-if Ladder

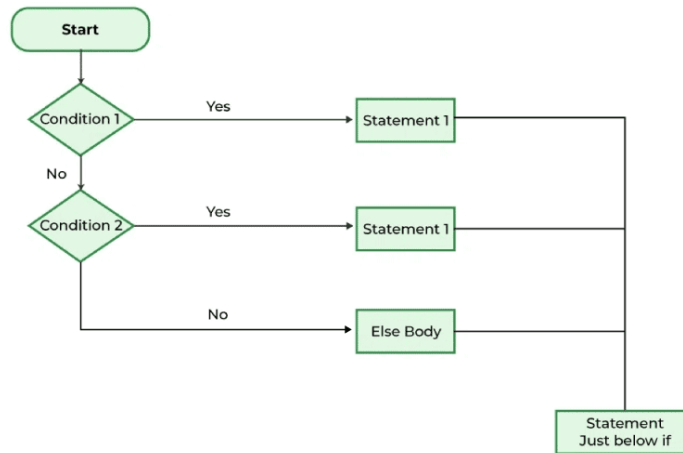


Fig 3.4: Flow Diagram of **if-else-if**

Example of **if-else-if** Ladder

```
// C program to illustrate nested-if statement
#include <stdio.h>
```

```
int main()
{
    int i = 20;

    if (i == 10)
        printf("i is 10");
    else if (i == 15)
        printf("i is 15");
    else if (i == 20)
        printf("i is 20");
    else
        printf("i is not present");
}
```

Output

```
i is 20
```

3.3 The **while** loop

The **while** loop is an entry-controlled loop in C programming language. This loop can be used to iterate a part of code while the given condition remains true.

Syntax

The while loop syntax is as follows:

```
while (test expression)
{
```

```
// body consisting of multiple statements
}
```

Example

```
#include <stdio.h>
int main()
{
    // Initialization of loop variable
    int i = 0;

    // setting test expression as (i < 5), means the loop
    // will execute till i is less than 5
    while (i < 5) {

        // loop statements
        printf("SLIET\n");

        // updating the loop variable
        i++;
    }
    return 0;
}
```

Output

```
SLIET
SLIET
SLIET
SLIET
SLIET
```

while Loop Structure

The **while** loop works by following a very structured top-down approach that can be divided into the following parts:

1. **Initialization:** In this step, we initialize the **loop variable** to some **initial value**. Initialization is not part of **while** loop syntax, but it is essential when we are using some variable in the test expression.
2. **Conditional Statement:** This is one of the most crucial steps as it decides whether the block in the **while** loop code will be executed. The **while** loop body will be executed if and only the **test condition** defined in the conditional statement is **true**.
3. **Body:** It is the actual set of statements that will be executed till the specified condition is true. It is generally enclosed inside **{ } braces**.
4. **Updation:** It is an expression that **updates** the value of the **loop variable** in each iteration. It is also not part of the syntax, but we have to define it explicitly in the body of the loop.

Flowchart of while loop in C

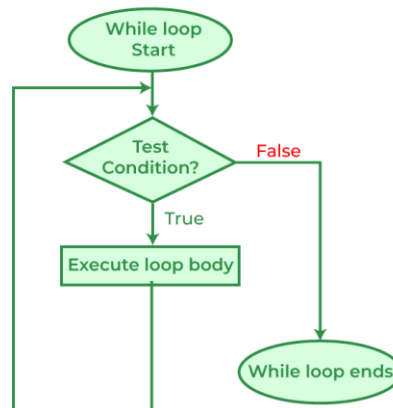


Fig 3.5: Flow Diagram of **while** loop

Working of while Loop

We can understand the working of the while loop by looking at the above flowchart:

1. **STEP 1:** When the program first comes to the loop, the test condition will be evaluated.
2. **STEP 2A:** If the test condition is **false**, the body of the loop will be skipped program will continue.
3. **STEP 2B:** If the expression evaluates to true, the body of the loop will be executed.
4. **STEP 3:** After executing the body, the program control will go to STEP 1. This process will continue till the test expression is true.

3.4 The **do...while** loop

The **do...while** in C is a loop statement used to repeat some part of the code till the given condition is fulfilled. It is a form of an **exit-controlled or post-tested loop** where the test condition is checked after executing the body of the loop. Due to this, the statements in the **do...while** loop will always be executed at least once no matter what the condition is.

Syntax of **do...while** Loop in C

```
do {  
  
    // body of do-while loop  
  
} while (condition);
```

How to Use **do...while** Loop in C

The following example demonstrates the use of **do...while** loop in C programming language.

```

#include <stdio.h>
int main()
{
    // loop variable declaration and initialization
    int i = 0;
    // do while loop
    do {
        printf("SLIET\n");
        i++;
    } while (i < 3);
    return 0;
}

```

Output

```

SLIET
SLIET
SLIET

```

How does the `do...while` Loop works?

1. When the program control first comes to the `do...while` loop, **the body of the loop is executed first and then the test condition/expression is checked**, unlike other loops where the test condition is checked first. Due to this property, the `do...while` loop is also called exit controlled or post-tested loop.
2. When the test condition is evaluated as **true**, the **program control goes to the start** of the loop and the body is executed once more.
3. The above process repeats till the test condition is true.
4. When the test condition is evaluated as **false**, **the program controls move on to the next statements** after the `do...while` loop.

As with the `while` loop in C, initialization and updation is not a part of the `do...while` loop syntax. We have to do that explicitly before and, in the loop, respectively.

The flowchart below shows the visual representation of the flow of the `do...while` loop in C.

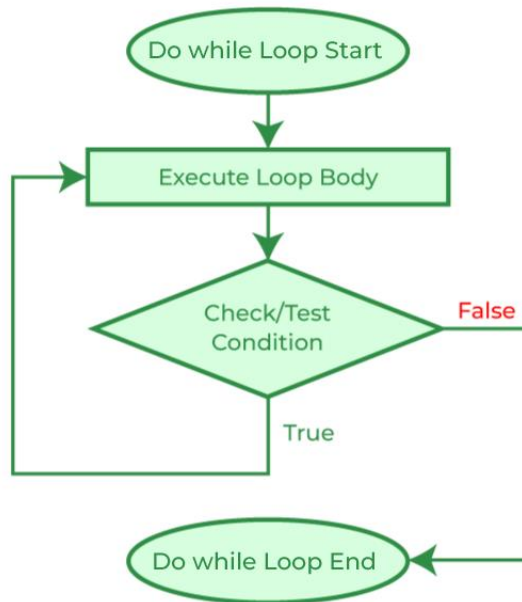


Fig. 3.6: Flowchart of `do...while` Loop in C

Difference between `while` and `do...while` Loop in C

<code>while</code> Loop	<code>do...while</code> Loop
The test condition is checked before the loop body is executed.	The test condition is checked after executing the body.
When the condition is false, the body is not executed even once.	The body of the <code>do...while</code> loop is executed at least once even when the condition is false.
It is a type of pre-tested or entry-controlled loop.	It is a type of post-tested or exit-controlled loop.
Semicolon is not required.	Semicolon is required at the end.

3.5 The `for` loop

The **for** loop in C Language provides a functionality/feature to repeat a set of statements a defined number of times. The **for** loop is in itself a form of an **entry-controlled loop**.

Unlike the **while** loop and **do...while** loop, the **for** loop contains the initialization, condition, and updating statements as part of its syntax. It is mainly used to traverse arrays, vectors, and other data structures.

Syntax of **for** Loop

for(initialization; check/test expression; updation)

```
{  
    // body consisting of multiple statements  
}
```

Structure of **for** Loop

The **for** loop follows a very structured approach where it begins with initializing a condition then checks the condition and, in the end, executes conditional statements followed by an updating of values.

1. **Initialization:** This step initializes a loop control variable with an initial value that helps to progress the loop or helps in checking the condition. It acts as the index value when iterating an array or string.
2. **Check/Test Condition:** This step of the **for** loop defines the condition that determines whether the loop should continue executing or not. The condition is checked before each iteration and if it is true then the iteration of the loop continues otherwise the loop is terminated.
3. **Body:** It is the set of statements i.e. variables, functions, etc. that is executed repeatedly till the condition is true. It is enclosed within curly braces { }.
4. **Updation:** This specifies how the loop control variable should be updated after each iteration of the loop. Generally, it is the incrementation (variable++) or decrementation (variable--) of the loop control variable.

5. How **for** Loop Works?

The working of for loop is mentioned below:

6. **Step 1:** Initialization is the basic step of **for** loop this step occurs only once during the start of the loop. During Initialization, variables are declared, or already existing variables are assigned some value.
 - **Step 2:** During the second step, the condition statements are checked and only if the condition is the satisfied loop we can further process otherwise loop is broken.
 - **Step 3:** All the statements inside the loop are executed.
 - **Step 4:** Updating the values of variables has been done as defined in the loop. Continue to Step 2 till the loop breaks.

Flowchart of **for** Loop

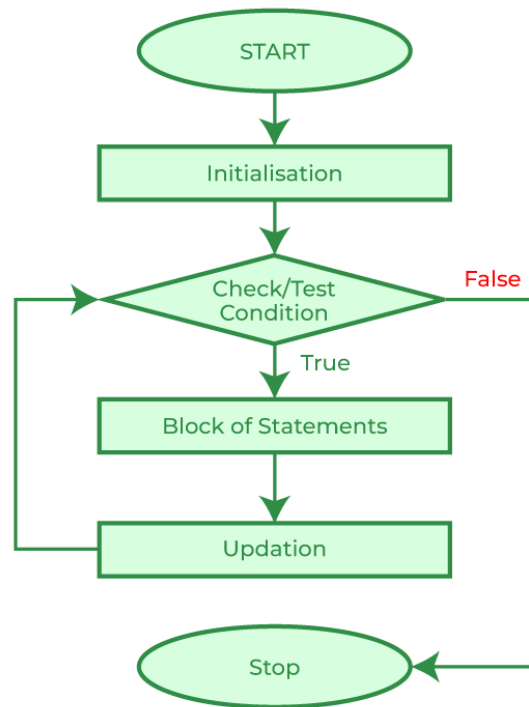


Fig. 3.7: C `for` Loop Flow Diagram

Example of `for` loop

```

#include <stdio.h>
int main()
{
    int i = 0;
    // i <= 5 is the check/test expression
    // The loop will function if and only if i is less
    // than 5
    // 'i++' will increment its value by this so that the
    // loop can iterate for further evaluation

    // conditional statement
    for (i = 1; i <= 5; i++)
    {
        // statement will be printed
        printf("SLIET\n");
    }
    // Return statement to tell that everything executed
    // safely
    return 0;
}
  
```

Output

```
SLIET
SLIET
SLIET
SLIET
SLIET
```

3.6 The `break` statement

The `break` in C is a loop control statement that breaks out of the loop when encountered. It can be used inside loops or switch statements to bring the control out of the block. The `break` statement can only break out of a single loop at a time.

Syntax of `break` in C

```
break;
```

We just put the break wherever we want to terminate the execution of the loop.

Use of `break` in C

The `break` statement in C is used for breaking out of the loop. We can use it with any type of loop to bring the program control out of the loop. In C, we can use the `break` statement in the following ways:

- Simple Loops
- Nested Loops
- Infinite Loops
- Switch case

Example of `break` in C

```
// C Program to demonstrate break statement with for loop
#include <stdio.h>

int main()
{
    // using break inside for loop to terminate after 2
    // iteration
    printf("break in for loop\n");
    for (int i = 1; i < 5; i++) {
        if (i == 3) {
            break;
        }
        else {
            printf("%d ", i);
        }
    }
}
```

```

// using break inside while loop to terminate after 2
// iteration
printf("\nbreak in while loop\n");
int i = 1;
while (i < 20) {
    if (i == 3)
        break;
    else
        printf("%d ", i);
    i++;
}
return 0;
}

```

Output

```

break in for loop
1 2
break in while loop
1 2

```

3.7 The `continue` statement

The C `continue` statement resets program control to the beginning of the loop when encountered. As a result, the current iteration of the loop gets skipped and the control moves on to the next iteration. Statements after the `continue` statement in the loop are not executed.

Syntax of `continue` in C

```
continue;
```

Use of `continue` in C

The `continue` statement in C can be used in any kind of loop to skip the current iteration. In C, we can use it in the following types of loops:

- Single Loops
- Nested Loops

Example of `continue` in C

```

// C program to explain the use
// of continue statement with for loop
#include <stdio.h>
int main()
{
    // for loop to print 1 to 8
    for (int i = 1; i <= 8; i++) {
        // when i = 4, the iteration will be skipped and for

```

```

        // will not be printed
        if (i == 4) {
            continue;
        }
        printf("%d ", i);
    }
    printf("\n");

    int i = 0;
    // while loop to print 1 to 8
    while (i < 8) {
        // when i = 4, the iteration will be skipped and for
        // will not be printed
        i++;
        if (i == 4) {
            continue;
        }
        printf("%d ", i);
    }
    return 0;
}

```

Output

```

1 2 3 5 6 7 8
1 2 3 5 6 7 8

```

3.8 The `switch` statement

`switch case` statement evaluates a given expression and based on the evaluated value (matching a certain condition), it executes the statements associated with it. Basically, it is used to perform different actions based on different conditions (cases).

- `switch case` statements follow a selection-control mechanism and allow a value to change control of execution.
- They are a substitute for long `if` statements that compare a variable to several integral values.
- The `switch` statement is a multiway branch statement. It provides an easy way to dispatch execution to different parts of code based on the value of the expression.

In C, the `switch case` statement is used for executing one condition from multiple conditions. It is similar to an `if-else-if` ladder. The `switch` statement consists of conditional-based cases and a default case.

Syntax of `switch` Statement in C

```

switch(expression)
{
    case value1: statement_1;
                break;

```



```

    case value2: statement_2;
        break;
    .
    .
    .
    case value_n: statement_n;
        break;
    default: default_statement;
}

```

How to use `switch case` Statement in C?

The following are some of the rules that we need to follow while using the `switch` statement:

- In a `switch` statement, the “`case value`” must be of “`char`” and “`int`” type.
- There can be one or N number of cases.
- The values in the case must be **unique**.
- Each statement of the `case` can have a `break` statement. It is optional.
- The `default` Statement is also optional.

Example

```

// C program to Demonstrate returning of day based numeric value
#include <stdio.h>
int main()
{
    // switch variable
    int var = 1;

    // switch statement
    switch (var)
    {
        case 1:
            printf("Case 1 is Matched.");
            break;

        case 2:
            printf("Case 2 is Matched.");
            break;

        case 3:
            printf("Case 3 is Matched.");
            break;

        default:
            printf("Default case is Matched.");
            break;
    }
}

```

```
return 0;  
}
```

Output

Case 1 is Matched.

How switch Statement Works?

The working of the `switch` statement in C is as follows:

1. **Step 1:** The `switch` variable is evaluated.
2. **Step 2:** The evaluated value is matched against all the present cases.
3. **Step 3A:** If the matching case value is found, the associated code is executed.
4. **Step 3B:** If the matching code is not found, then the default case is executed, if present.
5. **Step 4A:** If the `break` keyword is present in the case, then program control breaks out of the switch statement.
6. **Step 4B:** If the `break` keyword is not present, then all the cases after the matching case are executed.
7. **Step 5:** Statements after the `switch` statement are executed.

Flowchart of switch Statement

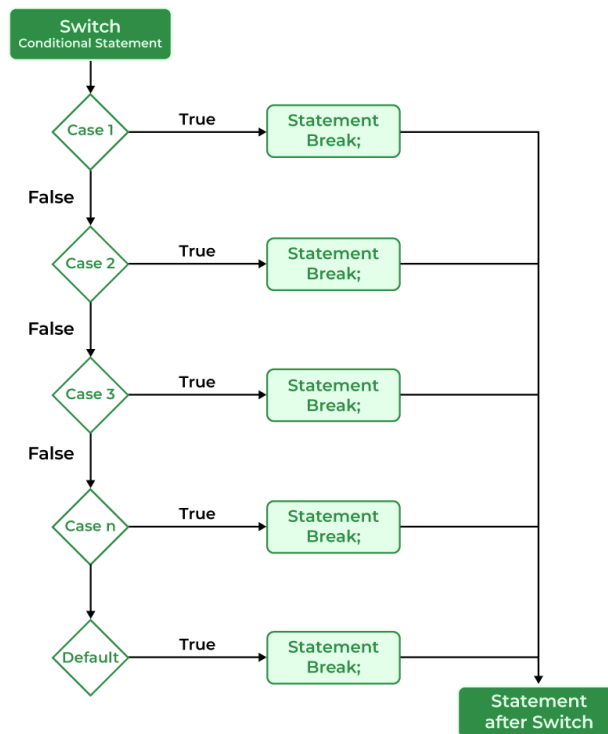


Fig. 3.8: Flowchart of switch statement in C

Important Points About switch case Statements

1. `switch` expression should result in a constant value

If the expression provided in the `switch` statement does not result in a constant value, it would not be valid. Some valid expressions for switch case will be,

```
// Constant expressions allowed
switch(1+2+23)
switch(1*2+3%4)

// Variable expression are allowed provided
// they are assigned with fixed values
switch(a*b+c*d)
switch(a+b+c)
```

2. Expression value should be only of `int` or `char` type.

The `switch` statement can only evaluate the integer or character value. So, the switch expression should return the values of type `int` or `char` only.

3. Case Values must be Unique

In the C `switch` statement, duplicate case values are not allowed.

3. Nesting of `switch` Statements

Nesting of `switch` statements is allowed, which means you can have `switch` statements inside another `switch`. However nested `switch` statements should be avoided as it makes the program more complex and less readable.

4. The `default` block can be placed anywhere

Regardless of its placement, the `default` case only gets executed if none of the other case conditions are met. So, putting it at the beginning, middle, or end doesn't change the core logic.

Chapter 4

Functions

4.1 Introduction to functions

A function in C is a set of statements that when called perform some specific tasks. It is the basic building block of a C program that provides modularity and code reusability. The programming statements of a function are enclosed within { } braces, having certain meanings and performing certain operations. They are also called subroutines or procedures in other languages.

Syntax of Functions in C

The syntax of function can be divided into 3 aspects:

1. **Function Declaration**
2. **Function Definition**
3. **Function Calls**

Function Declarations

In a function declaration, we must provide the function name, its return type, and the number and type of its parameters. A function declaration tells the compiler that there is a function with the given name defined somewhere else in the program.

Syntax

```
return_type name_of_the_function (parameter_1, parameter_2);
```

The parameter name is not mandatory while declaring functions. We can also declare the function without using the name of the data variables.

Example

```
int sum(int a, int b); // Function declaration with parameter names  
int sum(int , int); // Function declaration without parameter names
```

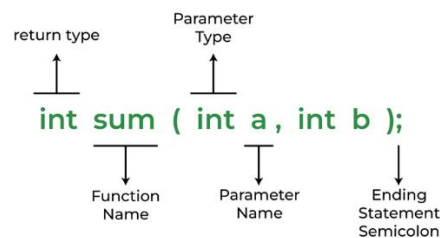


Fig. 4.1: Function Declaration

Note: A function in C must always be declared globally before calling it.

Function Definition

The function definition consists of actual statements which are executed when the function is called (i.e. when the program control comes to the function).

A C function is generally defined and declared in a single step because the function definition always starts with the function declaration, so we do not need to declare it explicitly. The below example serves as both a function definition and a declaration.

```
return_type function_name (para1_type para1_name, para2_type
para2_name)
{
    // body of the function
}
```

Function Definition

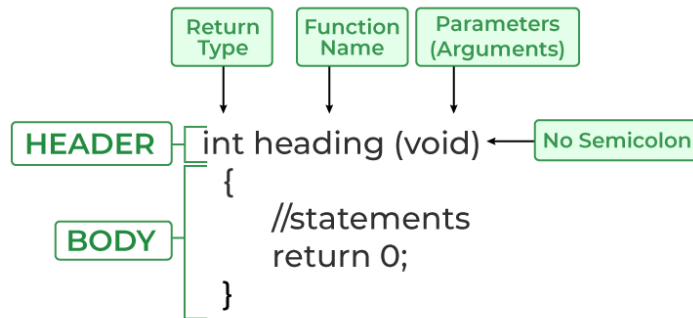


Fig. 4.2: Function Definition in C

Function Call

A function call is a statement that instructs the compiler to execute the function. We use the function name and parameters in the function call.

In the below example, the first `sum` function is called and 10, 30 are passed to the `sum` function. After the function call sum of `a` and `b` is returned and control is also returned back to the `main` function of the program.

Working of Function in C

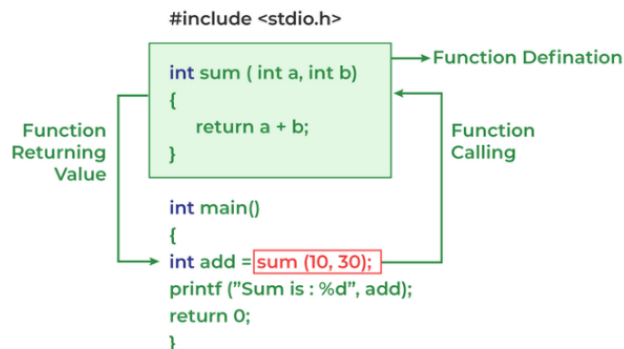


Fig. 4.3: Working of function in C

Note: Function call is necessary to bring the program control to the function definition. If not called, the function statements will not be executed.

Example of C Function

```
// C program to show function
// call and definition
#include <stdio.h>

// Function that takes two parameters
// a and b as inputs and returns
// their sum
int sum(int a, int b)
{
    return a + b;
}

// Driver code
int main()
{
    // Calling sum function and
    // storing its value in add variable
    int add = sum(10, 30);

    printf("Sum is: %d", add);
    return 0;
}
```

Output

```
Sum is: 40
```

As we noticed, we have not used explicit function declaration. We simply defined and called the function.

Function Return Type

Function return type tells what type of value is returned after all function is executed. When we don't want to return a value, we can use the **void** data type.

Example:

```
int func(parameter_1, parameter_2);
```

The above function will return an integer value after running statements inside the function.

Note: Only one value can be returned from a C function. To return multiple values, we have to use pointers or structures.

Function Arguments

Function Arguments (also known as Function Parameters) are the data that is passed to a function.

Example:

```
int function_name(int var1, int var2);
```

Conditions of Return Types and Arguments

In C programming language, functions can be called either with or without arguments and might return values. They may or might not return values to the calling functions.

1. Function with no arguments and no return value
2. Function with no arguments and with return value
3. Function with argument and with no return value
4. Function with arguments and with return value

How Does C Function Work?

Working of the C function can be broken into the following steps as mentioned below:

1. **Declaring a function:** Declaring a function is a step where we declare a function. Here we specify the return types and parameters of the function.
2. **Defining a function:** This is where the function's body is provided. Here, we specify what the function does, including the operations to be performed when the function is called.
3. **Calling the function:** Calling the function is a step where we call the function by passing the arguments in the function.
4. **Executing the function:** Executing the function is a step where we can run all the statements inside the function to get the final result.
5. **Returning a value:** Returning a value is the step where the calculated value after the execution of the function is returned. Exiting the function is the final step where all the allocated memory to the variables, functions, etc., is destroyed before giving full control back to the caller.

4.2 Types of Functions

There are two types of functions in C:

1. **Library Functions**
2. **User Defined Functions**

1. Library Function

A library function is also referred to as a “**built-in function**”. A compiler package already exists that contains these functions, each of which has a specific meaning and is included in the package. Built-in functions have the advantage of being directly usable without being defined, whereas user-defined functions must be declared and defined before being used.

For Example:

`pow()`, `sqrt()`, `strcmp()`, `strcpy()` etc.

Example:

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

```

#include <stdio.h>

// Driver code
int main()
{
    double Number;
    Number = 49;

    // Computing the square root with
    // the help of predefined C
    // library function
    double squareRoot = sqrt(Number);

    printf("The Square root of %.2lf = %.2lf",
           Number, squareRoot);
    return 0;
}

```

Output

```
The Square root of 49.00 = 7.00
```

2. User Defined Function

Functions that the programmer creates are known as User-Defined functions or “**taylor-made functions**”. User-defined functions can be improved and modified according to the needs of the programmer. Whenever we write a function that is case-specific and is not defined in any header file, we need to declare and define our own functions according to the syntax.

Advantages of User-Defined Functions

- Changeable functions can be modified as per need.
- The Code of these functions is reusable in other programs.
- These functions are easy to understand, debug and maintain.

Example:

```

// C program to show
// user-defined functions
#include <stdio.h>

int sum(int a, int b)
{
    return a + b;
}

// Driver code
int main()
{

```



```
int a = 30, b = 40;

// function call
int res = sum(a, b);

printf("Sum is: %d", res);
return 0;
}
```

Output

Sum is: 70

4.3 Passing Parameters to Functions

The data passed when the function is being invoked is known as the actual parameters. In the below program, 10 and 30 are known as actual parameters. Formal Parameters are the variables and the data type as mentioned in the function declaration. In the below program, a and b are known as formal parameters.

```
#include <stdio.h>
int sum(int a, int b)
{
    return a + b;
}

int main()
{
    int add = sum(10, 30);
    printf("Sum is: %d", add);
    return 0;
}
```

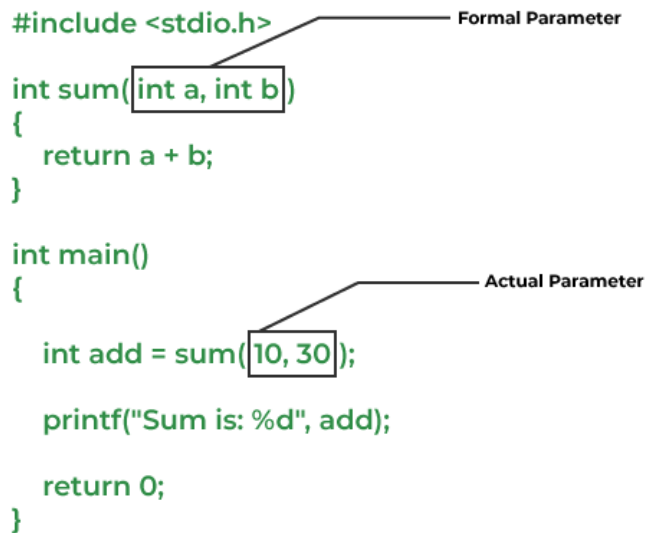


Fig. 4.4: Passing Parameters to Functions

We can pass arguments to the C function in two ways:

1. Pass by Value
2. Pass by Reference

1. Pass by Value

Parameter passing in this method copies values from actual parameters into formal function parameters. As a result, any changes made inside the functions do not reflect in the caller's parameters.

Example:

```
// C program to show use
// of call by value
#include <stdio.h>

void swap(int var1, int var2)
{
    int temp = var1;
    var1 = var2;
    var2 = temp;
}

// Driver code
int main()
{
    int var1 = 3, var2 = 2;
    printf("Before swap Value of var1 and var2 is: %d, %d\n",
           var1, var2);
    swap(var1, var2);
    printf("After swap Value of var1 and var2 is: %d, %d",
           var1, var2);
    return 0;
}
```

Output

```
Before swap Value of var1 and var2 is: 3, 2
After swap Value of var1 and var2 is: 3, 2
```

2. Pass by Reference

The caller's actual parameters and the function's actual parameters refer to the same locations, so any changes made inside the function are reflected in the caller's actual parameters.

Example:

```
// C program to show use of
// call by Reference
#include <stdio.h>

void swap(int *var1, int *var2)
{
    int temp = *var1;
    *var1 = *var2;
    *var2 = temp;
}

// Driver code
int main()
```

```

{
    int var1 = 3, var2 = 2;
    printf("Before swap Value of var1 and var2 is: %d, %d\n",
        var1, var2);
    swap(&var1, &var2);
    printf("After swap Value of var1 and var2 is: %d, %d",
        var1, var2);
    return 0;
}

```

Output

```

Before swap Value of var1 and var2 is: 3, 2
After swap Value of var1 and var2 is: 2, 3

```

4.4 Global and Local Variables

Local Variable:

Local variables are variables that are declared within a specific scope, such as within a function or a block of code. These variables are only accessible within that particular scope and are typically used for temporary storage of data or for performing calculations within a limited context. Once the scope in which a local variable is defined ends, the variable typically goes out of scope and its memory is released.

In many programming languages, local variables have a limited visibility and lifespan compared to global variables, which are accessible from any part of the program. This encapsulation of variables within specific scopes helps to organize code, prevent unintended modifications, and manage memory efficiently.

Example of Local Variable:

```

#include <stdio.h>
void exampleFunction() {
    // Local variable declaration
    int x = 10;
    int y = 20;
    int z = x + y;
    printf("The sum is: %d\n", z);
}

int main() {
    exampleFunction();
    return 0;
}

```

Output

```

The sum is: 30

```

Advantages of local variable:

1. **Encapsulation:** Local variables help encapsulate data within specific functions or blocks, reducing the risk of unintended modification.
2. **Memory Management:** They promote efficient memory usage by automatically releasing memory once the scope exits.
3. **Code Clarity:** Local variables make code easier to read and understand by limiting the scope of variables to where they are needed.
4. **Name Reusability:** Local variables allow the reuse of variable names without causing conflicts with variables in other scopes.

Disadvantages of local variable:

1. **Limited Accessibility:** Local variables cannot be accessed outside of the scope in which they are defined, which may restrict their use in certain scenarios.
2. **Potential for Shadowing Bugs:** Shadowing, where a local variable hides another variable with the same name in an outer scope, can lead to bugs and confusion if not handled properly.
3. **Lifetime Limited to Scope:** Local variables cease to exist once the scope in which they are defined exits, which may be a disadvantage if persistent data storage is required.

Global Variable:

Global variables are variables that are declared outside of any function or block of code and can be accessed from any part of the program. Unlike local variables, which have limited scope, global variables have a broader scope and can be used across multiple functions, modules, or files within a program. Here are some characteristics, features, advantages, disadvantages, and uses of global variables:

Example of Global Variable:

```
#include <stdio.h>
// Global variable declaration
int global_var = 100;

void exampleFunction() {
    // Local variable declaration
    int x = 10;
    int y = 20;
    int z = x + y + global_var;
    printf("The sum is: %d\n", z);
}

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

Output

```
The sum is: 130
```

Advantages of global variables:

1. **Accessibility:** Global variables provide a convenient way to share data across different parts of the program without passing them as function arguments.
2. **Ease of Use:** They simplify the sharing of data between functions and modules, reducing the need for complex parameter passing mechanisms.
3. **Persistence:** Global variables retain their values throughout the entire execution of the program, making them suitable for storing persistent data.
4. **Reduced Code Duplication:** Global variables can help reduce code duplication by centralizing data that is used in multiple parts of the program.

Disadvantages of global variables:

1. **Encapsulation Issues:** Global variables can lead to encapsulation issues by allowing any part of the program to modify their values, potentially leading to unintended side effects.
2. **Debugging Complexity:** Since global variables can be accessed and modified from anywhere in the program, tracking down bugs related to their usage can be challenging.
3. **Potential for Race Conditions:** In multithreaded or concurrent programs, global variables can introduce race conditions if accessed and modified concurrently by multiple threads or processes.
4. **Maintainability:** Excessive use of global variables can make code harder to understand and maintain, as their effects may not be localized to specific functions or modules.

4.5 Recursion

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 kind of function calls are called recursive calls.

In C, recursion is used to solve complex problems by breaking them down into simpler sub-problems. We can solve large numbers of problems using recursion in C. For example, factorial of a number, generating Fibonacci series, generating subsets, etc.

Recursive Functions in C

In C, a function that calls itself is called Recursive Function. The recursive functions contain a call to themselves somewhere in the function body. Moreover, such functions can contain multiple recursive calls.

Basic Structure of Recursive Functions

The basic syntax structure of the recursive functions is:

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

Example: C Program to Implement Recursion

In the below program, recursion is used to calculate the sum of the first N natural numbers.

```
// C Program to calculate the sum of first N natural numbers
// using recursion
#include <stdio.h>

int nSum(int n)
{
    // base condition to terminate the recursion when N = 0
    if (n == 0) {
        return 0;
    }

    // recursive case / recursive call
    int res = n + nSum(n - 1);

    return res;
}

int main()
{
    int n = 5;

    // calling the function
    int sum = nSum(n);

    printf("Sum of First %d Natural Numbers: %d", n, sum);
    return 0;
}
```

Output

```
Sum of First 5 Natural Numbers: 15
```

We will understand the different concepts of recursion using this example.

Fundamentals of C Recursion

The fundamental of recursion consists of two objects which are essential for any recursive function. These are:

1. Recursion Case
2. Base Condition

```

int nSum(int n)
{
    if (n==0) {
        return 0;
    }
    int res = n+ nsum(n-1);
    return res;
}

```

Base condition

Recursive case

Fig. 4.5: Base Condition and Recursion Case for nSum() Function

1. Recursion Case

The recursion case refers to the recursive call present in the recursive function. It decides what type of recursion will occur and how the problem will be divided into smaller subproblems.

The recursion case defined in the `nSum()` function of the above example is:

```
int res = n + nSum(n - 1);
```

2. Base Condition

The base condition specifies when the recursion is going to terminate. It is the condition that determines the exit point of the recursion.

Note: It is important to define the base condition before the recursive case otherwise, the base condition may never encounter and recursion might continue till infinity.

Considering the above example again, the base condition defined for the `nSum()` function:

```
if (n == 0) {
    return 0;
}
```

How Recursion works in C?

To understand how C recursion works, we will again refer to the example above and trace the flow of the program. In the `nSum()` function, Recursive Case is

```
int res = n + nSum(n - 1);
```

In the example, `n = 5`, so as `nSum(5)`'s recursive case, we get

```
int res = 5 + nSum(4);
```

In `nSum(4)`, the recursion case and everything else will be the same, but `n = 4`. Let's evaluate the recursive case for `n = 4`,

```
int res = 4 + nSum(3);
```

Similarly, for `nSum(3)`, `nSum(2)` and `nSum(1)`

```
int res = 3 + nSum(2); // nSum(3)
```

```
int res = 2 + nSum(1); // nSum(2)
```

```
int res = 1 + nSum(0); // nSum(1)
```

Let's not evaluate `nSum(0)` and further for now.

Now recall that the return value of the `nSum()` function in this same integer named `res`. So, instead of the function, we can put the value returned by these functions. As such, for `nSum(5)`, we get

```
int res = 5 + 4 + nSum(3);
```

Similarly, putting return values of `nSum()` for every `n`, we get

```
int res = 5 + 4 + 3 + 2 + 1 + nSum(0);
```

In `nSum()` function, the base condition is

```
if (n == 0) {  
    return 0;  
}
```

which means that when `nSum(0)` will return 0. Putting this value in `nSum(5)`'s recursive case, we get

```
int res = 5 + 4 + 3 + 2 + 1 + 0 = 15
```

At this point, we can see that there is no function call left. So, the recursion will stop here, and the final value returned by the function will be 15 which is the sum of the first 5 natural numbers.

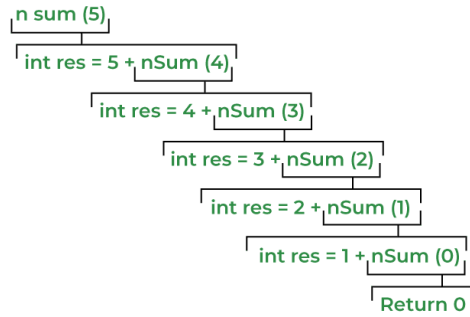


Fig. 4.6: Recursion Tree Diagram of $nSum(5)$ Function

Memory Allocation for C Recursive Function

To further improve our understanding of recursion in C, we will look into how the recursion is internally handled by the C compiler and how the memory is managed for recursive functions.

As you may know, all the function's local variables and other stuff are stored inside the stack frame in stack memory and once the function returns some value, its stack frame is removed from the memory. The recursion follows a similar concept but with a little twist. In Recursion,

- A stack frame is created on top of the existing stack frames each time a recursive call is encountered and the data of each recursive copy of the function will be stored in their respective stack.
- Once, some 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 returned some value, we come back to the base function and the finally return the control to the caller function.

Let's use the first example again and see how the memory is managed for the `nSum()` function.

Step 1:

When `nSum()` is called from the `main()` function with `5` as an argument, a stack frame for `nSum(5)` is created.

Step 2:

While executing `nSum(5)`, a recursive call is encountered as `nSum(4)`. The compiler will now create a new stack frame on top of the `nSum(5)`'s stack frame and maintain an instruction pointer at the statement where `nSum(4)` was encountered.

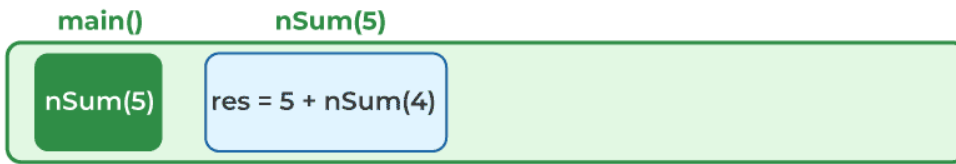


Fig. 4.7: Function Call Stack at the Execution of `nSum(5)`

Step 3:

In the execution of `nSum(4)`, we encounter another recursive call as `nSum(3)`. The compiler will again follow the same steps and maintain another instruction pointer and stack frame for `nSum(3)`.

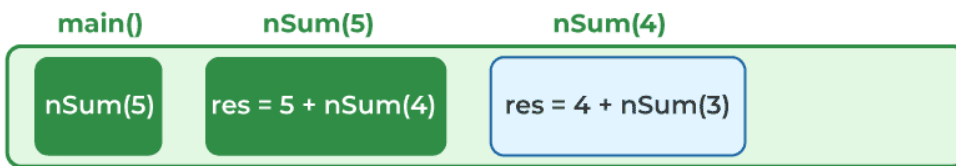


Fig. 4.8: Function Call Stack at the Execution of `nSum(4)`

Step 4:

The same thing will happen with `nSum(3)`, `nSum(2)`, and `nSum(1)`'s execution.

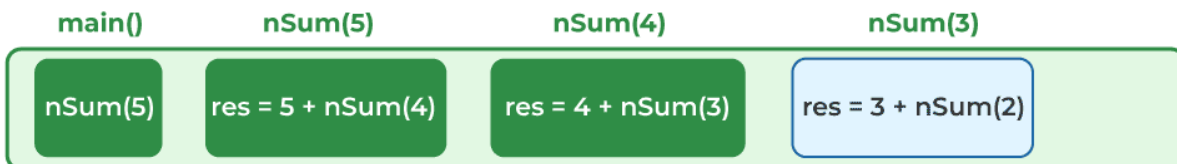


Fig. 4.9: Function Call Stack at the Execution of `nSum(3)`

Step 5:

But when the control comes to `nSum(0)`, the condition `(n == 0)` becomes true and the statement `return 0` is executed.

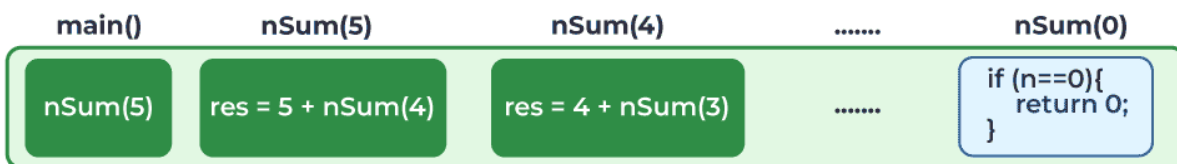


Fig. 4.10: Function Call Stack at the Execution of `nSum(0)`

Step 6:

As the value is returned by the `nSum(0)`, the stack frame for the `nSum(0)` will be destroyed. Using the instruction pointer, the program control will return to the `nSum(1)` function and the `nSum(0)` call will be replaced by value `0`.

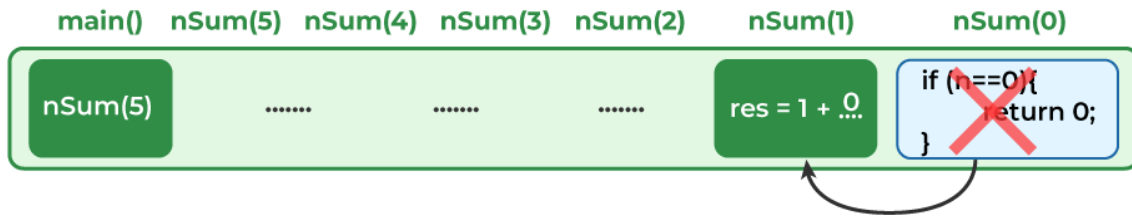


Fig. 4.11: `nSum(0)` Function Returning Value

Step 7:

Now, in `nSum(1)`, the expression `int res = 1 + 0` will be evaluated and the statement `return res` will be executed. The program control will move to the `nSum(2)`.

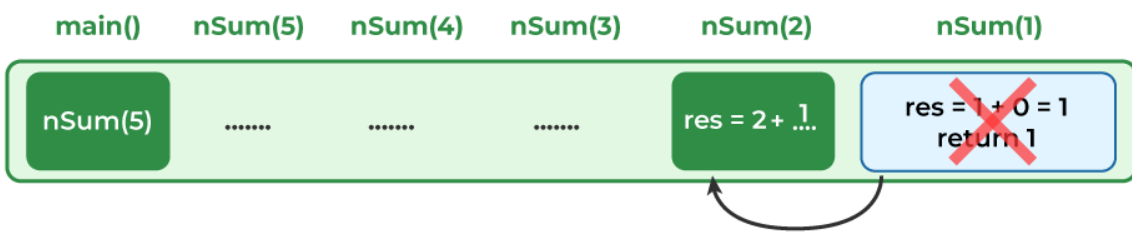


Fig. 4.12: `nSum(1)` Function Returning Value

Step 8:

In `nSum(2)`, `nSum(1)` call will be replaced by the value it returned, which is `1`. So, after evaluating `int res = 2 + 1`, `3` will be returned to `nSum(3)`. The same thing will keep happening till the control comes to the `nSum(5)` again.

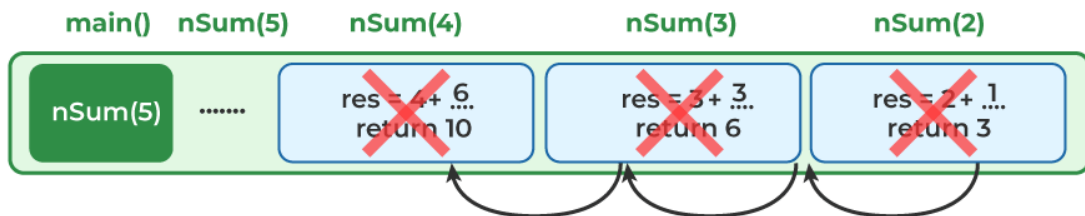


Fig. 4.13: `nSum(2)`, `nSum(3)` and `nSum(4)` Functions Returning Value

Step 9:

When the control reaches the `nSum(5)`, the expression `int res = 5 + nSum(4)` will look like `int res = 5 + 10`. Finally, this value will be returned to the `main()` function and the execution of `nSum()` function will be completed.

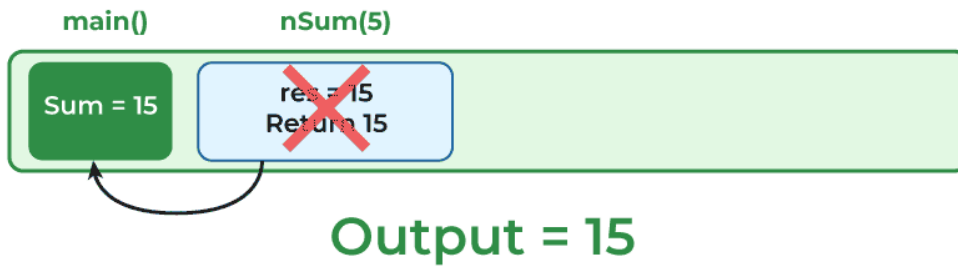


Fig. 4.14: Final Result Returned to `main()`

Stack Overflow

The program's call stack has limited memory assigned to it by the operating system and is generally enough for program execution. But if we have a recursive function that goes on for infinite times, sooner or later, the memory will be exhausted, and no more data can be stored. This is called stack overflow. In other words, stack overflow is an error that occurs when the call stack of the program cannot store more data resulting in program termination.

Chapter 5

Arrays

5.1 Introduction to Arrays

An array in C is a fixed-size collection of similar data items stored in contiguous memory locations. It can be used to store the collection of primitive data types such as int, char, float, etc., and also derived and user-defined data types such as pointers, structures, etc.

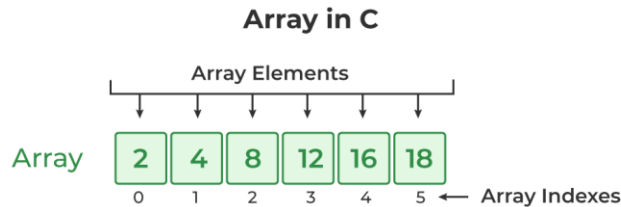


Fig. 5.1: An array in C

5.2 C Array Declaration

In C, we have to declare the array like any other variable before using it. We can declare an array by specifying its name, the type of its elements, and the size of its dimensions. When we declare an array in C, the compiler allocates the memory block of the specified size to the array name.

Syntax of Array Declaration

```
data_type array_name [size];  
or  
data_type array_name [size1] [size2]...[sizeN];
```

where **N** is the number of dimensions.

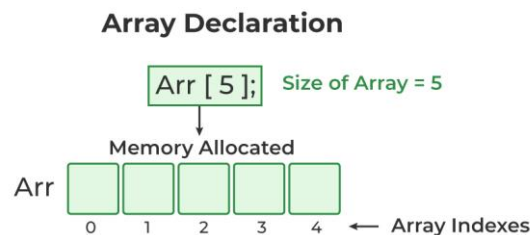


Fig. 5.2: Declaring an array in C

The C arrays are static in nature, i.e., they are allocated memory at the compile time.

Example of Array Declaration

```
// C Program to illustrate the array declaration  
#include <stdio.h>  
int main()
```

```

{
    // declaring array of integers
    int arr_int[5];
    // declaring array of characters
    char arr_char[5];
    return 0;
}

```

5.3 C Array Initialization

Initialization in C is the process to assign some initial value to the variable. When the array is declared or allocated memory, the elements of the array contain some garbage value. So, we need to initialize the array to some meaningful value. There are multiple ways in which we can initialize an array in C.

1. Array Initialization with Declaration

In this method, we initialize the array along with its declaration. We use an initializer list to initialize multiple elements of the array. An initializer list is the list of values enclosed within braces `{ }` separated by a comma.

```
data_type array_name [size] = {value1, value2, ... valueN};
```

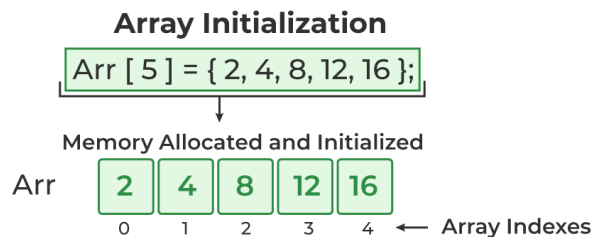


Fig. 5.3: Array initialization with declaration

2. Array Initialization with Declaration without Size

If we initialize an array using an initializer list, we can skip declaring the size of the array as the compiler can automatically deduce the size of the array in these cases. The size of the array in these cases is equal to the number of elements present in the initializer list as the compiler can automatically deduce the size of the array.

```
data_type array_name[] = {1,2,3,4,5};
```

The size of the above arrays is 5 which is automatically deduced by the compiler.

3. Array Initialization after Declaration (Using Loops)

We initialize the array after the declaration by assigning the initial value to each element individually. We can use `for` loop, `while` loop, or `do-while` loop to assign the value to each element of the array.

```
for (int i = 0; i < N; i++) {
    array_name[i] = valuei;
}
```

Example of Array Initialization in C

```
// C Program to demonstrate array initialization
#include <stdio.h>

int main()
{
    // array initialization using initializer list
    int arr[5] = { 10, 20, 30, 40, 50 };

    // array initialization using initializer list without
    // specifying size
    int arr1[] = { 1, 2, 3, 4, 5 };

    // array initialization using for loop
    float arr2[5];
    for (int i = 0; i < 5; i++) {
        arr2[i] = (float)i * 2.1;
    }
    return 0;
}
```

5.4 Accessing Array Elements

We can access any element of an array in C using the array subscript operator `[]` and the index value `i` of the element.

```
array_name [index];
```

One thing to note is that the indexing in the array always starts with 0, i.e., the **first element** is at index **0** and the **last element** is at $N - 1$ where N is the number of elements in the array.

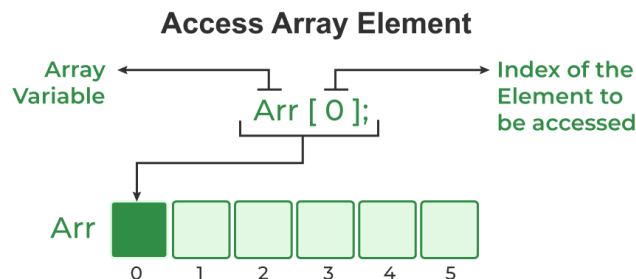


Fig. 5.4: Accessing array elements

Example of Accessing Array Elements using Array Subscript Operator

```
// C Program to illustrate element access using array
// subscript
#include <stdio.h>

int main()
{
    // array declaration and initialization
    int arr[5] = { 15, 25, 35, 45, 55 };

    // accessing element at index 2 i.e 3rd element
    printf("Element at arr[2]: %d\n", arr[2]);

    // accessing element at index 4 i.e last element
    printf("Element at arr[4]: %d\n", arr[4]);

    // accessing element at index 0 i.e first element
    printf("Element at arr[0]: %d", arr[0]);

    return 0;
}
```

Output

```
Element at arr[2]: 35
Element at arr[4]: 55
Element at arr[0]: 15
```

5.5 Updating Array Elements

We can update the value of an element at the given index **i** in a similar way to accessing an element by using the array subscript operator **[]** and assignment operator **=**.

```
array_name[i] = new_value;
```

5.6 C Array Traversal

Traversal is the process in which we visit every element of a data structure. For C array traversal, we use loops to iterate through each element of the array.

Array Traversal using for Loop

```
for (int i = 0; i < N; i++) {
    array_name[i];
}
```

5.7 How to use Arrays in C?

The following program demonstrates how to use an array in the C programming language:

```
// C Program to demonstrate the use of array
#include <stdio.h>

int main()
{
    // array declaration and initialization
    int arr[5] = { 10, 20, 30, 40, 50 };

    // modifying element at index 2
    arr[2] = 100;

    // traversing array using for loop
    printf("Elements in Array: ");
    for (int i = 0; i < 5; i++) {
        printf("%d ", arr[i]);
    }

    return 0;
}
```

Output

```
Elements in Array: 10 20 100 40 50
```

5.8 Types of Arrays in C

There are two types of arrays based on the number of dimensions it has. They are as follows:

1. One Dimensional Arrays (1D Array)
2. Multidimensional Arrays

1. One Dimensional Array in C

The One-dimensional arrays, also known as 1-D arrays in C, are those arrays that have only one dimension.

Syntax of 1D Array in C

```
array_name [size];
```

Example of 1D Array in C

```
// C Program to illustrate the use of 1D array
#include <stdio.h>

int main()
{
```

```

// 1d array declaration
int arr[5];

// 1d array initialization using for loop
for (int i = 0; i < 5; i++) {
    arr[i] = i * i - 2 * i + 1;
}

printf("Elements of Array: ");
// printing 1d array by traversing using for loop
for (int i = 0; i < 5; i++) {
    printf("%d ", arr[i]);
}

return 0;
}

```

Output

```
Elements of Array: 1 0 1 4 9
```

Array of Characters (Strings)

In C, we store the words, i.e., a sequence of characters in the form of an array of characters terminated by a NULL character. These are called strings in C language.

```

// C Program to illustrate strings
#include <stdio.h>

int main()
{
    // creating array of character
    char arr[6] = { 'S', 'L', 'I', 'E', 'T', '\0' };

    // printing string
    int i = 0;
    while (arr[i]) {
        printf("%c", arr[i++]);
    }
    return 0;
}

```

Output

```
SLIET
```

2. Multidimensional Array in C

Multi-dimensional Arrays in C are those arrays that have more than one dimension. Some of the popular multidimensional arrays are 2D arrays and 3D arrays. We can declare arrays with more dimensions than 3d arrays, but they are avoided as they get very complex and occupy a large amount of space.

A. Two-Dimensional Array in C

A Two-Dimensional array or 2D array in C is an array that has exactly two dimensions. They can be visualized in the form of rows and columns organized in a two-dimensional plane.

Syntax of 2D Array in C

```
array_name[size1] [size2];
```

Here,

- **size1**: Size of the first dimension.
- **size2**: Size of the second dimension.

Example of 2D Array in C

```
// C Program to illustrate 2d array
#include <stdio.h>

int main()
{
    // declaring and initializing 2d array
    int arr[2][3] = { 10, 20, 30, 40, 50, 60 };

    printf("2D Array:\n");
    // printing 2d array
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 3; j++) {
            printf("%d ", arr[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

Output

```
2D Array:
10 20 30
40 50 60
```

B. Three-Dimensional Array in C

Another popular form of a multi-dimensional array is Three Dimensional Array or 3D Array. A 3D array has exactly three dimensions. It can be visualized as a collection of 2D arrays stacked on top of each other to create the third dimension.

Syntax of 3D Array in C

```
array_name [size1] [size2] [size3];
```

Example of 3D Array

```
// C Program to illustrate the 3d array
#include <stdio.h>

int main()
{
    // 3D array declaration
    int arr[2][2][2] = { 10, 20, 30, 40, 50, 60 };

    // printing elements
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            for (int k = 0; k < 2; k++) {
                printf("%d ", arr[i][j][k]);
            }
            printf("\n");
        }
        printf("\n \n");
    }
    return 0;
}
```

Output

```
10 20
30 40
```

```
50 60
0 0
```

5.9 Relationship between Arrays and Pointers

Arrays and Pointers are closely related to each other such that we can use pointers to perform all the possible operations of the array. The array name is a constant pointer to the first element of the array and the array decays to the pointers when passed to the function.

```
// C Program to demonstrate the relation between arrays and
// pointers
```

```

#include <stdio.h>

int main()
{
    int arr[5] = { 10, 20, 30, 40, 50 };
    int* ptr = &arr[0];

    // comparing address of first element and address stored
    // inside array name
    printf("Address Stored in Array name: %p\nAddress of "
           "1st Array Element: %p\n",
           arr, &arr[0]);

    // printing array elements using pointers
    printf("Array elements using pointer: ");
    for (int i = 0; i < 5; i++) {
        printf("%d ", *ptr++);
    }
    return 0;
}

```

Output

```

Address Stored in Array name: 0x7ffcab67d8e0
Address of 1st Array Element: 0x7ffcab67d8e0
Array elements using pointer: 10 20 30 40 50

```

5.10 Passing an Array to a Function in C

An array is always passed as pointers to a function in C. Whenever we try to pass an array to a function, it decays to the pointer and then passed as a pointer to the first element of an array.

We can verify this using the following C Program:

```

// C Program to pass an array to a function
#include <stdio.h>

void printArray(int arr[])
{
    printf("Size of Array in Functions: %d\n", sizeof(arr));
    printf("Array Elements: ");
    for (int i = 0; i < 5; i++) {
        printf("%d ", arr[i]);
    }
}

// driver code
int main()

```

```

{
    int arr[5] = { 10, 20, 30, 40, 50 };

    printf("Size of Array in main(): %d\n", sizeof(arr));
    printArray(arr);
    return 0;
}

```

Output

```

Size of Array in main(): 20
Size of Array in Functions: 8
Array Elements: 10 20 30 40 50

```

5.11 Returning an Array from a Function in C

In C, we can only return a single value from a function. To return multiple values or elements, we have to use pointers. We can return an array from a function using a pointer to the first element of that array.

```

// C Program to return array from a function
#include <stdio.h>

// function
int* func()
{
    static int arr[5] = { 1, 2, 3, 4, 5 };

    return arr;
}

// driver code
int main()
{
    int* ptr = func();

    printf("Array Elements: ");
    for (int i = 0; i < 5; i++) {
        printf("%d ", *ptr++);
    }
    return 0;
}

```

Output

```

Array Elements: 1 2 3 4 5

```

Note: You may have noticed that we declared static array using static keyword. This is due to the fact that when a function returns a value, all the local variables and other entities declared inside that function are deleted. So, if we create a local array instead of static, we will get segmentation fault while trying to access the array in the main function.

5.12 Properties of Arrays in C

It is very important to understand the properties of the C array so that we can avoid bugs while using it.

1. Fixed Size

The array in C is a fixed-size collection of elements. The size of the array must be known at the compile time and it cannot be changed once it is declared.

2. Homogeneous Collection

We can only store one type of element in an array. There is no restriction on the number of elements but the type of all of these elements must be the same.

3. Indexing in Array

The array index always starts with 0 in C language. It means that the index of the first element of the array will be 0 and the last element will be $N - 1$.

4. Dimensions of an Array

A dimension of an array is the number of indexes required to refer to an element in the array. It is the number of directions in which you can grow the array size.

5. Contiguous Storage

All the elements in the array are stored continuously one after another in the memory. It is one of the defining properties of the array in C which is also the reason why random access is possible in the array.

6. Random Access

The array in C provides random access to its element i.e we can get to a random element at any index of the array in constant time complexity just by using its index number.

7. No Index Out of Bounds Checking

There is no index out-of-bounds checking in C/C++, for example, the following program compiles fine but may produce unexpected output when run.

```
// This C program compiles fine  
// as index out of bound  
// is not checked in C.
```

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

```
int main()
```

```
{
```

```
    int arr[2];
```

```

printf("%d ", arr[3]);
printf("%d ", arr[-2]);

return 0;
}

```

Output

```
0 0
```

In C, it is not a compiler error to initialize an array with more elements than the specified size. For example, the below program compiles fine and shows just a Warning.

```

#include <stdio.h>
int main()
{
    // Array declaration by initializing it
    // with more elements than specified size.
    int arr[2] = { 10, 20, 30, 40, 50 };

    return 0;
}

```

Output

```

Warnings:
prog.c: In function 'main':
prog.c:7:25: warning: excess elements in array initializer
    int arr[2] = { 10, 20, 30, 40, 50 };
                   ^
prog.c:7:25: note: (near initialization for 'arr')
prog.c:7:29: warning: excess elements in array initializer
    int arr[2] = { 10, 20, 30, 40, 50 };
                   ^
prog.c:7:29: note: (near initialization for 'arr')
prog.c:7:33: warning: excess elements in array initializer
    int arr[2] = { 10, 20, 30, 40, 50 };
                   ^
prog.c:7:33: note: (near initialization for 'arr')

```

5.13 Examples of Arrays in C

Example 1: C Program to print the average of the given list of numbers

In this program, we will store the numbers in an array and traverse it to calculate the average of the number stored.


```

// C Program to the average to two numbers
#include <stdio.h>

// function to calculate average of the function
float getAverage(float* arr, int size)
{
    int sum = 0;
    // calculating cumulative sum of all the array elements
    for (int i = 0; i < size; i++) {
        sum += arr[i];
    }

    // returning average
    return sum / size;
}

// driver code
int main()
{
    // declaring and initializing array
    float arr[5] = { 10, 20, 30, 40, 50 };
    // size of array using sizeof operator
    int n = sizeof(arr) / sizeof(float);

    // printing array elements
    printf("Array Elements: ");
    for (int i = 0; i < n; i++) {
        printf("%.0f ", arr[i]);
    }

    // calling getAverage function and printing average
    printf("\nAverage: %.2f", getAverage(arr, n));
    return 0;
}

```

Output

```

Array Elements: 10 20 30 40 50
Average: 30.00

```

Example 2: C Program to find the largest number in the array.

```

// C Program to find the largest number in the array.
#include <stdio.h>

// function to return max value

```

```

int getMax(int* arr, int size)
{
    int max = arr[0];
    for (int i = 1; i < size; i++) {
        if (max < arr[i]) {
            max = arr[i];
        }
    }
    return max;
}

// Driver code
int main()
{
    int arr[10]
        = { 135, 165, 1, 16, 511, 65, 654, 654, 169, 4 };

    printf("Largest Number in the Array: %d",
        getMax(arr, 10));

    return 0;
}

```

Output

```
Largest Number in the Array: 654
```

5.14 Advantages and Disadvantages of Arrays in C

The following are the main advantages of an array:

1. Random and fast access of elements using the array index.
2. Use of fewer lines of code as it creates a single array of multiple elements.
3. Traversal through the array becomes easy using a single loop.
4. Sorting becomes easy as it can be accomplished by writing fewer lines of code.

The following are the main disadvantages of an array:

1. Allows a fixed number of elements to be entered which is decided at the time of declaration. Unlike a linked list, an array in C is not dynamic.
2. Insertion and deletion of elements can be costly since the elements need to be rearranged after insertion and deletion.

Chapter 6

Structures and Unions

The structure in C is a user-defined data type that can be used to group items of possibly different types into a single type. The `struct` keyword is used to define the structure in the C programming language. The items in the structure are called its member and they can be of any valid data type. Additionally, the values of a structure are stored in contiguous memory locations.

6.1 C Structure Declaration

We have to declare structure in C before using it in our program. In structure declaration, we specify its member variables along with their datatype. We can use the `struct` keyword to declare the structure in C using the following syntax:

Syntax

```
struct structure_name {  
    data_type member_name1;  
    data_type member_name1;  
    ....  
    ....  
};
```

The above syntax is also called a structure template or structure prototype and no memory is allocated to the structure in the declaration.

6.2 C Structure Definition

To use structure in our program, we have to define its instance. We can do that by creating variables of the structure type. We can define structure variables using two methods:

1. Structure Variable Declaration with Structure Template

```
struct structure_name {  
    data_type member_name1;  
    data_type member_name1;  
    ....  
    ....  
}variable1, variable2, ...;
```

2. Structure Variable Declaration after Structure Template

```
// structure declared beforehand  
struct structure_name variable1, variable2, .....;
```

6.3 Accessing Structure Members

We can access structure members by using the `(.)` dot operator.

Syntax

```
structure_name.member1;  
structure_name.member2;
```

In the case where we have a pointer to the structure, we can also use the arrow operator to access the members.

6.4 Initializing Structure Members

Structure members **cannot be** initialized with the declaration. For example, the following C program fails in the compilation.

```
struct Point  
{  
    int x = 0; // COMPILER ERROR: cannot initialize members here  
    int y = 0; // COMPILER ERROR: cannot initialize members here  
};
```

The reason for the above error is simple. When a datatype is declared, no memory is allocated for it. Memory is allocated only when variables are created.

Default Initialization

By default, structure members are not automatically initialized to 0 or NULL. Uninitialized structure members will contain garbage values. However, when a structure variable is declared with an initializer, all members not explicitly initialized are zero-initialized.

```
struct Point  
{  
    int x;  
    int y;  
};  
struct Point p = {0}; // Both x and y are initialized to 0
```

We can initialize structure members in 3 ways which are as follows:

1. Using Assignment Operator.
2. Using Initializer List.
3. Using Designated Initializer List.

1. Initialization using Assignment Operator

```
struct structure_name str;  
str.member1 = value1;  
str.member2 = value2;  
str.member3 = value3;  
.
```

.
.

2. Initialization using Initializer List

```
struct structure_name str = { value1, value2, value3 };
```

In this type of initialization, the values are assigned in sequential order as they are declared in the structure template.

3. Initialization using Designated Initializer List

Designated Initialization allows structure members to be initialized in any order. This feature has been added in the C99 standard.

```
struct structure_name str = { .member1 = value1, .member2 = value2,  
.member3 = value3 };
```

Example of Structure in C

The following C program shows how to use structures

```
// C program to illustrate the use of structures  
#include <stdio.h>  
  
// declaring structure with name str1  
struct str1 {  
    int i;  
    char c;  
    float f;  
    char s[30];  
};  
  
// declaring structure with name str2  
struct str2 {  
    int ii;  
    char cc;  
    float ff;  
} var; // variable declaration with structure template  
  
// Driver code  
int main()  
{  
    // variable declaration after structure template  
    // initialization with initializer list and designated  
    // initializer list  
    struct str1 var1 = { 1, 'A', 1.00, "SLIET" }, var2;  
    struct str2 var3 = { .ff = 5.00, .ii = 5, .cc = 'a' };
```

```

// copying structure using assignment operator
var2 = var1;

printf("Struct 1:\n\ti = %d, c = %c, f = %f, s = %s\n",
      var1.i, var1.c, var1.f, var1.s);
printf("Struct 2:\n\ti = %d, c = %c, f = %f, s = %s\n",
      var2.i, var2.c, var2.f, var2.s);
printf("Struct 3\n\ti = %d, c = %c, f = %f\n", var3.i,
      var3.c, var3.f);

return 0;
}

```

Output

```

Struct 1:
  i = 1, c = A, f = 1.000000, s = SLIET
Struct 2:
  i = 1, c = A, f = 1.000000, s = SLIET
Struct 3
  i = 5, c = a, f = 5.000000

```

6.5 Nested Structures

C language allows us to insert one structure into another as a member. This process is called nesting and such structures are called nested structures. There are two ways in which we can nest one structure into another:

1. Embedded Structure Nesting

In this method, the structure being nested is also declared inside the parent structure.

Example

```

struct parent {
  int member1;
  struct member_str member2 {
    int member_str1;
    char member_str2;
    ...
  }
  ...
}

```

2. Separate Structure Nesting

In this method, two structures are declared separately and then the member structure is nested inside the parent structure.

Example

```

struct member_str {
    int member_str1;
    char member_str2;
    ...
}

```

```

struct parent {
    int member1;
    struct member_str member2;
    ...
}

```

One thing to note here is that the declaration of the structure should always be present before its definition as a structure member. For example, the declaration below is invalid as the `struct mem` is not defined when it is declared inside the parent structure.

```

struct parent {
    struct mem a;
};

```

```

struct mem {
    int var;
};

```

Accessing Nested Members

We can access nested Members by using the same (.) dot operator two times as shown:

```

str_parent.str_child.member;

```

Example of Structure Nesting

```

// C Program to illustrate structure nesting along with
// forward declaration
#include <stdio.h>

// child structure declaration
struct child {
    int x;
    char c;
};

// parent structure declaration
struct parent {
    int a;
    struct child b;
};

```



```

.....
};

```

The same initialization can also be done as:

```

struct structure_name array_name [number_of_elements] = {
    element1_value1, element1_value2 .....,
    element2_value1, element2_value2 .....,
};

```

GNU C compilers support designated initialization for structures so we can also use this in the initialization of an array of structures.

```

struct structure_name array_name [number_of_elements] = {
    {.element3 = value, .element1 = value, ....},
    {.element2 = value, .elementN = value, ....},
    .....,
    .....,
};

```

Example of Array of Structure in C

```

// C program to demonstrate the array of structures
#include <stdio.h>

// structure template
struct Employee {
    char Name[20];
    int employeeID;
    int WeekAttendance[7];
};

// driver code
int main()
{
    // defining array of structure of type Employee
    struct Employee emp[5];

    // adding data
    for (int i = 0; i < 5; i++) {
        emp[i].employeeID = i;
        strcpy(emp[i].Name, "Amit");
        int week;
        for (week = 0; week < 7; week++) {
            int attendance;
            emp[i].WeekAttendance[week] = week;
        }
    }
    printf("\n");
}

```

```

// printing data
for (int i = 0; i < 5; i++) {
    printf("Employee ID: %d - Employee Name: %s\n",
           emp[i].employeeID, emp[i].Name);
    printf("Attendance\n");
    int week;
    for (week = 0; week < 7; week++) {
        printf("%d ", emp[i].WeekAttendance[week]);
    }
    printf("\n");
}
return 0;
}

```

Output

```

Employee ID: 0 - Employee Name: Amit
Attendance
0 1 2 3 4 5 6
Employee ID: 1 - Employee Name: Amit
Attendance
0 1 2 3 4 5 6
Employee ID: 2 - Employee Name: Amit
Attendance
0 1 2 3 4 5 6
Employee ID: 3 - Employee Name: Amit
Attendance
0 1 2 3 4 5 6
Employee ID: 4 - Employee Name: Amit
Attendance
0 1 2 3 4 5 6

```

6.7 Uses of Structures in C

C structures are used for the following:

1. The structure can be used to define the custom data types that can be used to create some complex data types such as dates, time, complex numbers, etc. which are not present in the language.
2. It can also be used in data organization where a large amount of data can be stored in different fields.
3. Structures are used to create data structures such as trees, linked lists, etc.
4. They can also be used for returning multiple values from a function.

6.8 Limitations of C Structures

In C language, structures provide a method for packing together data of different types. A Structure is a helpful tool to handle a group of logically related data items. However, C structures also have some limitations.

- **Higher Memory Consumption:** It is due to structure padding.
- **No Data Hiding:** C Structures do not permit data hiding. Structure members can be accessed by any function, anywhere in the scope of the structure.
- **Functions inside Structure:** C structures do not permit functions inside the structure so we cannot provide the associated functions.
- **Static Members:** C Structure cannot have static members inside its body.
- **Construction creation in Structure:** Structures in C cannot have a constructor inside Structures.

6.9 Unions

The Union is a user-defined data type in C language that can contain elements of the different data types just like structure. But unlike structures, all the members in the C union are stored in the same memory location. Due to this, only one member can store data at the given instance.

Syntax of Union in C

The syntax of the union in C can be divided into three steps which are as follows:

C Union Declaration

In this part, we only declare the template of the union, i.e., we only declare the members' names and data types along with the name of the union. No memory is allocated to the union in the declaration.

```
union union_name {  
    datatype member1;  
    datatype member2;  
    ...  
};
```

Keep in mind that we have to always end the union declaration with a semi-colon.

Different Ways to Define a Union Variable

We need to define a variable of the union type to start using union members. There are two methods using which we can define a union variable.

1. With Union Declaration
2. After Union Declaration

1. Defining Union Variable with Declaration

```
union union_name {  
    datatype member1;  
    datatype member2;  
    ...  
} var1, var2, ...;
```

2. Defining Union Variable after Declaration

```
union union_name var1, var2, var3...;
```

where `union_name` is the name of an already declared union.

Accessing Union Members

We can access the members of a union by using the (.) dot operator just like structures.

```
var1.member1;
```

where `var1` is the **union variable** and `member1` is the **member of the union**.

The above method of accessing the members of the union also works for the nested unions.

```
var1.member1.memberA;
```

Here,

- `var1` is a union variable.
- `member1` is a member of the union.
- `memberA` is a member of member1.

Initialization of Union in C

The initialization of a union is the initialization of its members by simply assigning the value to it.

```
var1.member1 = some_value;
```

One important thing to note here is that **only one member can contain some value at a given instance of time**.

Example of Union

```
// C Program to demonstrate how to use union
```

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

```
// union template or declaration
```

```
union un {
```

```
    int member1;
```

```
    char member2;
```

```
    float member3;
```

```
};
```

```
// driver code
```

```
int main()
```

```
{
```

```
    // defining a union variable
```

```
    union un var1;
```

```
    // initializing the union member
```

```
    var1.member1 = 15;
```

```

    printf("The value stored in member1 = %d",
           var1.member1);

    return 0;
}

```

Output

```
The value stored in member1 = 15
```

Size of Union

The size of the union will always be equal to the size of the largest member of the array. All the less-sized elements can store the data in the same space without any overflow.

Example 1: C program to find the size of the union

```

// C Program to find the size of the union
#include <stdio.h>

// declaring multiple unions
union test1 {
    int x;
    int y;
} Test1;

union test2 {
    int x;
    char y;
} Test2;

union test3 {
    int arr[10];
    char y;
} Test3;

// driver code
int main()
{
    // finding size using sizeof() operator
    int size1 = sizeof(Test1);
    int size2 = sizeof(Test2);
    int size3 = sizeof(Test3);

    printf("Sizeof test1: %d\n", size1);
    printf("Sizeof test2: %d\n", size2);
    printf("Sizeof test3: %d", size3);
    return 0;
}

```

Output

```
Sizeof test1: 4
Sizeof test2: 4
Sizeof test3: 40
```

6.10 Difference between C Structure and C Union

The following table lists the key difference between the structure and union in C:

Structure	Union
The size of the structure is equal to or greater than the total size of all of its members.	The size of the union is the size of its largest member.
The structure can contain data in multiple members at the same time.	Only one member can contain data at the same time.
It is declared using the <code>struct</code> keyword.	It is declared using the <code>union</code> keyword.

Chapter 7

Pointers

A pointer is defined as a derived data type that can store the address of other C variables or a memory location. We can access and manipulate the data stored in that memory location using pointers.

As the pointers in C store the memory addresses, their size is independent of the type of data they are pointing to. This size of pointers in C only depends on the system architecture.

7.1 Syntax of C Pointers

The syntax of pointers is similar to the variable declaration in C, but we use the (*****) dereferencing operator in the pointer declaration.

```
datatype * ptr;
```

where

- **ptr** is the name of the pointer.
- **datatype** is the type of data it is pointing to.

The above syntax is used to define a pointer to a variable. We can also define pointers to functions, structures, etc.

7.2 How to Use Pointers?

The use of pointers in C can be divided into three steps:

1. **Pointer Declaration**
2. **Pointer Initialization**
3. **Pointer Dereferencing**

1. Pointer Declaration

In pointer declaration, we only declare the pointer but do not initialize it. To declare a pointer, we use the (*****) **dereference operator** before its name.

Example

```
int *ptr;
```

The pointer declared here will point to some random memory address as it is not initialized. Such pointers are called wild pointers.

2. Pointer Initialization

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

Example

```
int var = 10;  
int * ptr;  
ptr = &var;
```

We can also declare and initialize the pointer in a single step. This method is called **pointer definition** as the pointer is declared and initialized at the same time.

Example

```
int *ptr = &var;
```

Note: It is recommended that the pointers should always be initialized to some value before starting using it. Otherwise, it may lead to number of errors.

3. Pointer Dereferencing

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

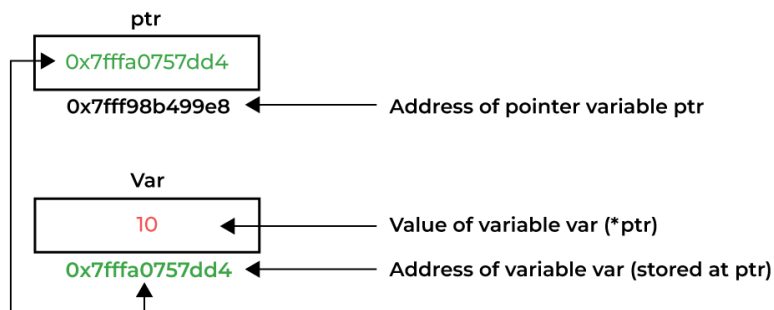


Fig. 7.1: Dereferencing a Pointer in C

C Pointer Example

```
// C program to illustrate Pointers
#include <stdio.h>

void func()
{
    int var = 10;

    // declare pointer variable
    int* ptr;

    // note that data type of ptr and var must be same
    ptr = &var;

    // assign the address of a variable to a pointer
    printf("Value at ptr = %p \n", ptr);
    printf("Value at var = %d \n", var);
    printf("Value at *ptr = %d \n", *ptr);
}

// Driver program
int main()
{
    func();
}
```



```
    return 0;
}
```

Output

```
Value at ptr = 0x7ffca84068dc
```

```
Value at var = 10
```

```
Value at *ptr = 10
```

7.3 Types of Pointers in C

Pointers in C can be classified into many different types based on the parameter on which we are defining their types. If we consider the type of variable stored in the memory location pointed by the pointer, then the pointers can be classified into the following types:

1. Integer Pointers

As the name suggests, these are the pointers that point to the integer values.

Syntax

```
int *ptr;
```

These pointers are pronounced as **Pointer to Integer**. Similarly, a pointer can point to any primitive data type. It can also point to derived data types such as arrays and user-defined data types such as structures.

2. Array Pointer

Pointers and Array are closely related to each other. Even the array name is the pointer to its first element. They are also known as Pointer to Arrays. We can create a pointer to an array using the given syntax.

Syntax

```
char *ptr = &array_name;
```

3. Structure Pointer

The pointer pointing to the structure type is called Structure Pointer or Pointer to Structure. It can be declared in the same way as we declare the other primitive data types.

Syntax

```
struct struct_name *ptr;
```

In C, structure pointers are used in data structures such as linked lists, trees, etc.

4. Function Pointers

Function pointers point to the functions. They are different from the rest of the pointers in the sense that instead of pointing to the data, they point to the code. Let's consider a function prototype – `int func(int, char)`, the function pointer for this function will be

Syntax

```
int (*ptr)(int, char);
```

Note: The syntax of the function pointers changes according to the function prototype.

5. Double Pointers

In C language, we can define a pointer that stores the memory address of another pointer. Such pointers are called double-pointers or pointers-to-pointer. Instead of pointing to a data value, they point to another pointer.

Syntax

```
datatype ** pointer_name;
```

Dereferencing Double Pointer

```
*pointer_name; // get the address stored in the inner level pointer
```

```
**pointer_name; // get the value pointed by inner level pointer
```

*Note: In C, we can create multi-level pointers with any number of levels such as – *****ptr3**, ******ptr4**, *******ptr5** and so on.*

6. NULL Pointer

The Null Pointers are those pointers that do not point to any memory location. They can be created by assigning a NULL value to the pointer. A pointer of any type can be assigned the NULL value.

Syntax

```
data_type *pointer_name = NULL;
```

OR

```
pointer_name = NULL;
```

It is said to be good practice to assign NULL to the pointers currently not in use.

7. Void Pointer

The Void pointers in C are the pointers of type **void**. It means that they do not have any associated data type. They are also called **generic pointers** as they can point to any type and can be typecasted to any type.

Syntax

```
void * pointer_name;
```

One of the main properties of void pointers is that they cannot be dereferenced.

8. Wild Pointers

The Wild Pointers are pointers that have not been initialized with something yet. These types of C-pointers can cause problems in our programs and can eventually cause them to crash. If values is updated using wild pointers, they could cause data abort or data corruption.

Example

```
int *ptr;  
char *str;
```

9. Constant Pointers

In constant pointers, the memory address stored inside the pointer is constant and cannot be modified once it is defined. It will always point to the same memory address.

Syntax

```
data_type * const pointer_name;
```

10. Pointer to Constant

The pointers pointing to a constant value that cannot be modified are called pointers to a constant. Here we can only access the data pointed by the pointer but cannot modify it. Although, we can change the address stored in the pointer to constant.

Syntax

```
const data_type * pointer_name;
```

7.4 Size of Pointers in C

The size of the pointers in C is equal for every pointer type. The size of the pointer does not depend on the type it is pointing to. It only depends on the operating system and CPU architecture. The size of pointers in C is

- 8 bytes for a 64-bit System
- 4 bytes for a 32-bit System

The reason for the same size is that the pointers store the memory addresses, no matter what type they are. As the space required to store the addresses of the different memory locations is the same, the memory required by one pointer type will be equal to the memory required by other pointer types.

How to find the size of pointers in C?

We can find the size of pointers using the **sizeof operator** as shown in the following program:

Example: C Program to find the size of different pointer types.

```
// C Program to find the size of different pointer types
#include <stdio.h>

// dummy structure
struct str {
};

// dummy function
void func(int a, int b){};

int main()
{
    // dummy variables definitions
    int a = 10;
    char c = 'G';
    struct str x;
```

```

// pointer definitions of different types
int* ptr_int = &a;
char* ptr_char = &c;
struct str* ptr_str = &x;
void (*ptr_func)(int, int) = &func;
void* ptr_vn = NULL;

// printing sizes
printf("Size of Integer Pointer \t:\t%d bytes\n",
      sizeof(ptr_int));
printf("Size of Character Pointer\t:\t%d bytes\n",
      sizeof(ptr_char));
printf("Size of Structure Pointer\t:\t%d bytes\n",
      sizeof(ptr_str));
printf("Size of Function Pointer\t:\t%d bytes\n",
      sizeof(ptr_func));
printf("Size of NULL Void Pointer\t:\t%d bytes",
      sizeof(ptr_vn));

return 0;
}

```

Output

```

Size of Integer Pointer      :      8 bytes
Size of Character Pointer   :      8 bytes
Size of Structure Pointer   :      8 bytes
Size of Function Pointer    :      8 bytes
Size of NULL Void Pointer   :      8 bytes

```

As we can see, no matter what the type of pointer it is, the size of each and every pointer is the same.

Now, one may wonder that if the size of all the pointers is the same, then why do we need to declare the pointer type in the declaration? **The type declaration is needed in the pointer for dereferencing and pointer arithmetic purposes.**

7.5 C Pointer Arithmetic

The Pointer Arithmetic refers to the legal or valid arithmetic operations that can be performed on a pointer. It is slightly different from the ones that we generally use for mathematical calculations as only a limited set of operations can be performed on pointers. These operations include:

- Increment in a Pointer
- Decrement in a Pointer
- Addition of integer to a pointer
- Subtraction of integer to a pointer
- Subtracting two pointers of the same type
- Comparison of pointers of the same type.
- Assignment of pointers of the same type.

```

// C program to illustrate Pointer Arithmetic

```

```

#include <stdio.h>

int main()
{
    // Declare an array
    int v[3] = { 10, 100, 200 };

    // Declare pointer variable
    int* ptr;

    // Assign the address of v[0] to ptr
    ptr = v;

    for (int i = 0; i < 3; i++) {

        // print value at address which is stored in ptr
        printf("Value of *ptr = %d\n", *ptr);

        // print value of ptr
        printf("Value of ptr = %p\n\n", ptr);

        // Increment pointer ptr by 1
        ptr++;
    }
    return 0;
}

```

Output

```

Value of *ptr = 10
Value of ptr = 0x7ffcfe7a77a0

Value of *ptr = 100
Value of ptr = 0x7ffcfe7a77a4

Value of *ptr = 200
Value of ptr = 0x7ffcfe7a77a8

```

7.6 C Pointers and Arrays

In C programming language, pointers and arrays are closely related. An array name acts like a pointer constant. The value of this pointer constant is the address of the first element. For example, if we have an array named `val` then `val` and `&val[0]` can be used interchangeably.

If we assign this value to a non-constant pointer of the same type, then we can access the elements of the array using this pointer.

Example 1: Accessing Array Elements using Pointer with Array Subscript

```

// C Program to access array elements using pointer

```

```

#include <stdio.h>

void func()
{
    // Declare an array
    int val[3] = { 5, 10, 15 };

    // Declare pointer variable
    int* ptr;

    // Assign address of val[0] to ptr.
    // We can use ptr=&val[0];(both are same)
    ptr = val;

    printf("Elements of the array are: ");

    printf("%d, %d, %d", ptr[0], ptr[1], ptr[2]);

    return;
}

// Driver program
int main()
{
    func();
    return 0;
}

```

Output

```
Elements of the array are: 5, 10, 15
```

Not only that, as the array elements are stored continuously, we can pointer arithmetic operations such as increment, decrement, addition, and subtraction of integers on pointer to move between array elements.

Example 2: Accessing Array Elements using Pointer Arithmetic

```

// C Program to access array elements using pointers
#include <stdio.h>

int main()
{
    // defining array
    int arr[5] = { 1, 2, 3, 4, 5 };

    // defining the pointer to array
    int* ptr_arr = arr;

    // traversing array using pointer arithmetic

```

```
    for (int i = 0; i < 5; i++) {  
        printf("%d ", *ptr_arr++);  
    }  
    return 0;  
}
```

Output

1 2 3 4 5

This concept is not limited to the one-dimensional array, we can refer to a multidimensional array element as well using pointers.

7.7 Uses of Pointers in C

The C pointer is a very powerful tool that is widely used in C programming to perform various useful operations. It is used to achieve the following functionalities in C:

1. Pass Arguments by Reference
2. Accessing Array Elements
3. Return Multiple Values from Function
4. Dynamic Memory Allocation
5. Implementing Data Structures
6. In System-Level Programming, memory addresses are useful.
7. In locating the exact value at some memory location.
8. To avoid compiler confusion for the same variable name.
9. To use in Control Tables.

7.8 Advantages and disadvantages of Pointers

The following are the major advantages of pointers in C:

- Pointers are used for dynamic memory allocation and deallocation.
- An Array or a structure can be accessed efficiently with pointers
- Pointers are useful for accessing memory locations.
- Pointers are used to form complex data structures such as linked lists, graphs, trees, etc.
- Pointers reduce the length of the program and its execution time as well.

Pointers are vulnerable to errors and have the following disadvantages:

- Memory corruption can occur if an incorrect value is provided to pointers.
- Pointers are a little bit complex to understand.
- Pointers are majorly responsible for memory leaks in C.
- Pointers are comparatively slower than variables in C.
- Uninitialized pointers might cause a segmentation fault.

Chapter 8

Files

File handling in C is the process of handling file operations such as creating, opening, writing data, reading data, renaming, and deleting using the C language functions. With the help of these functions, we can perform file operations to store and retrieve the data in/from the file in our program.

8.1 Need for File Handling in C

If we perform input and output operations using the C program, the data exists as long as the program is running, when the program is terminated, we cannot use that data again. **File handling** is required to work with files stored in external memory, i.e., to store and access the information to/from the computer's external memory. You can keep the data permanently using file handling.

8.2 Types of Files

A file represents a sequence of bytes. There are two types of files: **text files** and **binary files** –

1. **Text file** – A text file contains data in the form of ASCII characters and is generally used to store a stream of characters. Each line in a text file ends with a new line character ("**\n**"), and generally has a ".txt" extension.
2. **Binary file** – A binary file contains data in raw bits (0 and 1). Different application programs have different ways to represent bits and bytes and use different file formats. The image files (.png, .jpg), the executable files (.exe, .com), etc. are the examples of binary files.

8.3 The FILE Pointer (**FILE***)

While working with **file handling**, you need a **file pointer** to store the reference of the **FILE** structure returned by the **fopen()** function. The file pointer is required for all file-handling operations.

The **fopen()** function returns a pointer of the **FILE** type. **FILE** is a predefined struct type in **stdio.h** and contains attributes such as the file descriptor, size, and position, etc.

```
typedef struct {  
    int fd;           /* File descriptor */  
    unsigned char *buf; /* Buffer */  
    size_t size;      /* Size of the file */  
    size_t pos;       /* Current position in the file */  
} FILE;
```

Declaring a File Pointer (**FILE***)

Below is the syntax to declare a file pointer –

```
FILE* file_pointer;
```

8.4 Opening (Creating) a File

A file must be opened to perform any operation. The **fopen()** function is used to create a new file or open an existing file. You need to specify the mode in which you want to open. There are various file opening modes explained below, any one of them can be used during creating/opening a file.

The `fopen()` function returns a `FILE` pointer which will be used for other operations such as reading, writing, and closing the files.

Syntax

```
FILE *fopen(const char *filename, const char *mode);
```

Here, `filename` is the name of the file to be opened, and `mode` defines the file's opening mode.

8.5 File Opening Modes

The file access modes by default open the file in the text or ASCII mode. If you are going to handle binary files, then you will use the following access modes instead of the above-mentioned ones:

"rb", "wb", "ab", "rb+", "r+b", "wb+", "w+b", "ab+", "a+b"

There are various modes in which a file can be opened. The following are the different file opening modes:

Mode	Description
r	Opens an existing text file for reading purposes.
w	Opens a text file for writing. If it does not exist, then a new file is created. Here your program will start writing content from the beginning of the file.
a	Opens a text file for writing in appending mode. If it does not exist, then a new file is created. Here your program will start appending content in the existing file content.
r+	Opens a text file for both reading and writing.
	Opens a text file for both reading and writing. It first truncates the file to zero length if it exists, otherwise creates a file if it does not exist.
a+	Opens a text file for both reading and writing. It creates the file if it does not exist. The reading will start from the beginning but writing can only be appended.

Example of Creating a File

In the following example, we are creating a new file. The file mode to create a new file will be "w" (write-mode).

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

```
int main() {  
    FILE * file;
```

```

// Creating a file
file = fopen("file1.txt", "w");

// Checking whether file is
// created or not
if (file == NULL) {
    printf("Error in creating file");
    return 1;
}
printf("File created.");

return 0;
}

```

Output

File created.

Example of Opening a File

In the following example, we are opening an existing file. The file mode to open an existing file will be "r" (read-only). You may also use other file opening mode options explained above.

Note: There must be a file to be opened.

```

#include <stdio.h>

int main() {
    FILE * file;

    // Opening a file
    file = fopen("file1.txt", "r");

    // Checking whether file is
    // opened or not
    if (file == NULL) {
        printf("Error in opening file");
        return 1;
    }
    printf("File opened.");

    return 0;
}

```

Output

File opened.

Closing a File

Each file must be closed after performing operations on it. The `fclose()` function closes an opened file.

Syntax

```
int fclose(FILE *fp);
```

The `fclose()` function returns zero on success, or `EOF` if there is an error in closing the file.

The `fclose()` function actually flushes any data still pending in the buffer to the file, closes the file, and releases any memory used for the file. The `EOF` is a constant defined in the header file `stdio.h`.

Example of Closing a File

```
#include <stdio.h>
int main() {
    FILE * file;

    // Opening a file
    file = fopen("file1.txt", "w");

    // Checking whether file is
    // opened or not
    if (file == NULL) {
        printf("Error in opening file");
        return 1;
    }
    printf("File opened.");

    // Closing the file
    fclose(file);
    printf("\nFile closed.");

    return 0;
}
```

Output

```
File opened.
File closed.
```

8.6 Writing to a Text File

The following library functions are provided to write data in a file opened in writeable mode –

- `fputc()`: Writes a single character to a file.
- `fputs()`: Writes a string to a file.
- `fprintf()`: Writes a formatted string (data) to a file.

Writing Single Character to a File

The `fputc()` function is an unformatted function that writes a single character value of the argument "`c`" to the output stream referenced by "`fp`".

```
int fputc(int c, FILE *fp);
```

Example

In the following code, one character from a given char array is written into a file opened in the "w" mode:

```
#include <stdio.h>

int main() {

    FILE *fp;
    char * string = "C Programming tutorial from TutorialsPoint";
    int i;
    char ch;

    fp = fopen("file1.txt", "w");

    for (i = 0; i < strlen(string); i++) {
        ch = string[i];
        if (ch == EOF)
            break;
        fputc(ch, fp);
    }
    printf ("\n");
    fclose(fp);

    return 0;
}
```

Output

After executing the program, "file1.txt" will be created in the current folder and the string is written to it.

Writing String to a File

The `fputs()` function writes the string "`s`" to the output stream referenced by "`fp`". It returns a non-negative value on success, else `EOF` is returned in case of any error.

```
int fputs(const char *s, FILE *fp);
```

Example

The following program writes strings from the given two-dimensional char array to a file.

```
#include <stdio.h>

int main() {
```

```

FILE *fp;
char *sub[] = {"C Programming Tutorial\n", "C++ Tutorial\n", "Python
Tutorial\n", "Java Tutorial\n"};
fp = fopen("file2.txt", "w");

for (int i = 0; i < 4; i++) {
    fputs(sub[i], fp);
}

fclose(fp);

return 0;
}

```

Output

When the program is run, a file named "file2.txt" is created in the current folder and saves the following lines:

```

C Programming Tutorial
C++ Tutorial
Python Tutorial
Java Tutorial

```

Writing Formatted String to a File

The `fprintf()` function sends a formatted stream of data to the disk file represented by the `FILE` pointer.

```

int fprintf(FILE *stream, const char *format [, argument, ...])

```

Example

In the following program, we have an array of struct type called "`employee`". The structure has a string, an integer, and a float element. Using the `fprintf()` function, the data is written to a file.

```

#include <stdio.h>

struct employee {
    int age;
    float percent;
    char *name;
};

int main() {

    FILE *fp;
    struct employee emp[] = {
        {25, 65.5, "Ravi"},
        {21, 75.5, "Roshan"},
        {24, 60.5, "Reena"}
    };
}

```

```

char *string;
fp = fopen("file3.txt", "w");

for (int i = 0; i < 3; i++) {
    fprintf(fp, "%d %f %s\n", emp[i].age, emp[i].percent,
emp[i].name);
}
fclose(fp);

return 0;
}

```

Output

When the above program is executed, a text file is created with the name "file3.txt" that stores the employee data from the struct array.

8.7 Reading from a Text File

The following library functions are provided to read data from a file that is opened in read mode –

- `fgetc()`: Reads a single character from a file.
- `fgets()`: Reads a string from a file.
- `fscanf()`: Reads a formatted string from a file.

Reading Single Character from a File

The `fgetc()` function reads a character from the input file referenced by "`fp`". The return value is the character read, or in case of any error, it returns `EOF`.

```
int fgetc(FILE * fp);
```

Example

The following example reads the given file in a character by character manner till it reaches the end of file.

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

```
int main(){
```

```
    FILE *fp ;
```

```
    char ch ;
```

```
    fp = fopen ("file1.txt", "r");
```

```
    while(1) {
```

```
        ch = fgetc (fp);
```

```
        if (ch == EOF)
```

```
            break;
```

```
        printf ("%c", ch);
```

```
    }
```

```
    printf ("\n");
```

```
    fclose (fp);
```

```
}
```

Output

Run the code and check its output.

Reading String from a File

The `fgets()` function reads up to "n - 1" characters from the input stream referenced by "`fp`". It copies the read string into the buffer "`buf`", appending a null character to terminate the string.

Example

This following program reads each line in the given file till the end of the file is detected:

```
# include <stdio.h>

int main() {

    FILE *fp;
    char *string;
    fp = fopen ("file2.txt", "r");

    while (!feof(fp)) {
        fgets(string, 256, fp);
        printf ("%s", string) ;
    }
    fclose (fp);
}
```

Output

Run the code and check its output.

Reading Formatted String from a File

The `fscanf()` function in C programming language is used to read formatted input from a file.

```
int fscanf(FILE *stream, const char *format, ...)
```

Example

In the following program, we use the `fscanf()` function to read the formatted data in different types of variables. Usual format specifiers are used to indicate the field types (`%d`, `%f`, `%s`, etc.)

```
#include <stdio.h>

int main() {

    FILE *fp;
    char *s;
    int i, a;
    float p;
```

```

fp = fopen ("file3.txt", "r");

if (fp == NULL) {
    puts ("Cannot open file"); return 0;
}

while (fscanf(fp, "%d %f %s", &a, &p, s) != EOF)
printf ("Name: %s Age: %d Percent: %f\n", s, a, p);
fclose(fp);

return 0;
}

```

Output

When the above program is executed, it opens the text file "file3.txt" and prints its contents on the screen. After running the code, you will get an output like this:

```

Name: Ravi Age: 25 Percent: 65.500000
Name: Roshan Age: 21 Percent: 75.500000
Name: Reena Age: 24 Percent: 60.500000

```

8.8 Binary Read and Write Functions

The read/write operations are done in a binary form in the case of a binary file. You need to include the character "b" in the access mode ("wb" for writing a binary file, "rb" for reading a binary file).

There are two functions that can be used for binary input and output: the `fread()` function and the `fwrite()` function. Both of these functions should be used to read or write blocks of memories, usually arrays or structures.

8.9 Writing to Binary File

The `fwrite()` function writes a specified chunk of bytes from a buffer to a file opened in binary write mode. Here is the prototype to use this function:

```

fwrite(*buffer, size, no, FILE);

```

Example

In the following program, an array of a struct type called `employee` has been declared. We use the `fwrite()` function to write one block of byte, equivalent to the size of one `employee` data, in a file that is opened in `wb` mode.

```

#include <stdio.h>

struct employee {
    int age;
    float percent;
    char name[10];
};

int main() {

```



```

FILE *fp;
struct employee e[] = {
    {25, 65.5, "Ravi"},
    {21, 75.5, "Roshan"},
    {24, 60.5, "Reena"}
};

char *string;

fp = fopen("file4.dat", "wb");
for (int i = 0; i < 3; i++) {
    fwrite(&e[i], sizeof (struct employee), 1, fp);
}

fclose(fp);

return 0;
}

```

Output

When the above program is run, the given file will be created in the current folder. It will not show the actual data, because the file is in binary mode.

8.10 Reading from Binary File

The `fread()` function reads a specified chunk of bytes from a file opened in binary read mode to a buffer of the specified size. Here is the prototype to use this function:

```
fread(*buffer, size, no, FILE);
```

Example

In the following program, an array of a struct type called `employee` has been declared. We use the `fread()` function to read one block of byte, equivalent to the size of one `employee` data, in a file that is opened in `rb` mode.

```

#include <stdio.h>

struct employee {
    int age;
    float percent;
    char name[10];
};

int main() {

    FILE *fp;
    struct employee e;

```

```
fp = fopen ("file4.dat", "rb");

if (fp == NULL) {
    puts ("Cannot open file");
    return 0;
}

while (fread (&e, sizeof (struct employee), 1, fp) == 1)
    printf ("Name: %s Age: %d Percent: %f\n", e.name, e.age, e.percent);

fclose(fp);

return 0;
}
```

Output

When the above program is executed, it opens the file "file4.dat" and prints its contents on the screen. After running the code, you will get an output like this:

```
Name: Ravi Age: 25 Percent: 65.500000
Name: Roshan Age: 21 Percent: 75.500000
Name: Reena Age: 24 Percent: 60.500000
```