# C Programming

## C Keywords and Identifiers

### Character set

A character set is a set of alphabets, letters and some special characters that are valid in C

language.

Alphabets

Uppercase: A B C .................................. X Y Z

Lowercase: a b c ...................................... x y z

C accepts both lowercase and uppercase alphabets as variables and functions.

Digits

0 1 2 3 4 5 6 7 8 9

Special Characters

Special Characters in C Programming

, &lt; &gt; . _

( ) ; $ :

% [ ] # ?

&#39; &amp; { } &quot;

^ ! * / |

- \ ~ +

White space Characters

Blank space, newline, horizontal tab, carriage return and form feed.

### C Keywords

Keywords are predefined; reserved words used in programming that have special meanings to

the compiler. Keywords are part of the syntax and they cannot be used as an identifier. For

example:

int money;

Here, int is a keyword that indicates money is a variable of type int (integer).

As C is a case sensitive language, all keywords must be written in lowercase. Here is a list of

all keywords allowed in ANSI C.

C Keywords

auto double int struct

break else long switch

case enum register typedef

char extern return union

continue for signed void

do if static while

default goto sizeof volatile

const float short unsigned

All these keywords, their syntax, and application will be discussed in their respective topics.

However, if you want a brief overview of these keywords without going further, visit List of

all keywords in C programming.

C Identifiers

Identifier refers to names given to entities such as variables, functions, structures etc.

Identifiers must be unique. They are created to give a unique name to an entity to identify it

during the execution of the program. For example:

int money;

double accountBalance;

Here, money and accountBalance are identifiers.

Also remember, identifier names must be different from keywords. You cannot use int as an

identifier because int is a keyword.

Rules for naming identifiers

A valid identifier can have letters (both uppercase and lowercase letters), digits and

underscores.

The first letter of an identifier should be either a letter or an underscore.

You cannot use keywords like int, while, etc. as identifiers.

There is no rule on how long an identifier can be. However, you may run into problems in

some compilers if the identifier is longer than 31 characters.

You can choose any name as an identifier if you follow the above rule, however, give

meaningful names to identifiers that make sense.

C Variables, Constants and Literals

Variables

In programming, a variable is a container (storage area) to hold data.

To indicate the storage area, each variable should be given a unique name (identifier).

Variable names are just the symbolic representation of a memory location. For example:

int playerScore = 95;

Here, playerScore is a variable of int type. Here, the variable is assigned an integer value 95.

The value of a variable can be changed, hence the name variable.

char ch = &#39;a&#39;;

ch = &#39;l&#39;;

Rules for naming a variable

A variable name can only have letters (both uppercase and lowercase letters), digits and

underscore.

The first letter of a variable should be either a letter or an underscore.

There is no rule on how long a variable name (identifier) can be. However, you may run into

problems in some compilers if the variable name is longer than 31 characters.

Note: You should always try to give meaningful names to variables. For example: firstName

is a better variable name than fn.

C is a strongly typed language. This means that the variable type cannot be changed once it is

declared. For example:

int number = 5; // integer variable

number = 5.5; // error

double number; // error

Here, the type of number variable is int. You cannot assign a floating-point (decimal) value

5.5 to this variable. Also, you cannot redefine the data type of the variable to double. By the

way, to store the decimal values in C, you need to declare its type to either double or float.

Visit this page to learn more about different types of data a variable can store.

Literals

Literals are data used for representing fixed values. They can be used directly in the code. For

example: 1, 2.5, &#39;c&#39; etc.

Here, 1, 2.5 and &#39;c&#39; are literals. Why? You cannot assign different values to these terms.

1. Integers

An integer is a numeric literal (associated with numbers) without any fractional or

exponential part. There are three types of integer literals in C programming:

decimal (base 10)

octal (base 8)

hexadecimal (base 16)

For example:

Decimal: 0, -9, 22 etc

Octal: 021, 077, 033 etc

Hexadecimal: 0x7f, 0x2a, 0x521 etc

In C programming, octal starts with a 0, and hexadecimal starts with a 0x.

2. Floating-point Literals

A floating-point literal is a numeric literal that has either a fractional form or an exponent

form. For example:

-2.0

0.0000234

-0.22E-5

Note: E-5 = 10-5

3. Characters

A character literal is created by enclosing a single character inside single quotation marks.

For example: &#39;a&#39;, &#39;m&#39;, &#39;F&#39;, &#39;2&#39;, &#39;}&#39; etc.

4. Escape Sequences

Sometimes, it is necessary to use characters that cannot be typed or has special meaning in C

programming. For example: newline (enter), tab, question mark etc.

In order to use these characters, escape sequences are used.

Escape Sequences

| Escape Sequences | Character |
|---|---|
| \b | Backspace |
| \f | Form feed |

\n Newline

\r Return

\t Horizontal tab

\v Vertical tab

\\ Backslash

\&#39; Single quotation mark

\&quot; Double quotation mark

\? Question mark

\0 Null character

For example: \n is used for a newline. The backslash \ causes escape from the normal way the

characters are handled by the compiler.

5. String Literals

A string literal is a sequence of characters enclosed in double-quote marks. For example:

&quot;good&quot; //string constant

&quot;&quot; //null string constant

&quot; &quot; //string constant of six white space

&quot;x&quot; //string constant having a single character.

&quot;Earth is round\n&quot; //prints string with a newline

Constants

If you want to define a variable whose value cannot be changed, you can use the const

keyword. This will create a constant. For example,

const double PI = 3.14;

Notice, we have added keyword const.

Here, PI is a symbolic constant; its value cannot be changed.

```
const double PI = 3.14;

PI = 2.9; //Error
```

You can also define a constant using the #define preprocessor directive. We will learn about

it in C Macros.

C Data Types

In C programming, data types are declarations for variables. This determines the type and

size of data associated with variables. For example,

```
int myVar;
```

Here, myVar is a variable of int (integer) type. The size of int is 4 bytes.

Basic types

Here&#39;s a table containing commonly used types in C programming for quick access.

Type Size (bytes) Format Specifier

Int at least 2, usually 4 %d, %i

char 1 %c

float 4 %f

double 8 %lf

short int 2 usually %hd

unsigned int at least 2, usually 4 %u

long int at least 4, usually 8 %ld, %li

long long int at least 8 %lld, %lli

unsigned long int at least 4 %lu

unsigned long long int at least 8 %llu

signed char 1 %c

unsigned char 1 %c

long double at least 10, usually 12 or 16 %Lf int

Integers are whole numbers that can have both zero, positive and negative values but no

decimal values. For example, 0, -5, 10

We can use int for declaring an integer variable.

5

int id;

Here, id is a variable of type integer.

You can declare multiple variables at once in C programming. For example,

int id, age;

The size of int is usually 4 bytes (32 bits). And, it can take 232 distinct states from -

2147483648 to 2147483647.

float and double

float and double are used to hold real numbers.

float salary;

double price;

In C, floating-point numbers can also be represented in exponential. For example,

float normalizationFactor = 22.442e2;

What&#39;s the difference between float and double?

The size of float (single precision float data type) is 4 bytes. And the size of double (double

precision float data type) is 8 bytes.

char

Keyword char is used for declaring character type variables. For example,

char test = &#39;h&#39;;

The size of the character variable is 1 byte.

void

void is an incomplete type. It means &quot;nothing&quot; or &quot;no type&quot;. You can think of void as absent.

For example, if a function is not returning anything, its return type should be void.

Note that, you cannot create variables of void type.

short and long

If you need to use a large number, you can use a type specifier long. Here&#39;s how:

```
long a;

long long b;

long double c;
```

Here variables a and b can store integer values. And, c can store a floating-point number.

If you are sure, only a small integer ([−32,767, +32,767] range) will

be used, you can use short.

```
short d;
```

You can always check the size of a variable using the sizeof() operator.

```
#include &lt;stdio.h&gt;

int main() {

short a;

long b;

long long c;

long double d;

printf(&quot;size of short = %d bytes\n&quot;, sizeof(a));

printf(&quot;size of long = %d bytes\n&quot;, sizeof(b));

printf(&quot;size of long long = %d bytes\n&quot;, sizeof(c));

printf(&quot;size of long double= %d bytes\n&quot;, sizeof(d));
```

```
return 0;
```

```
}
```

signed and unsigned

In C, signed and unsigned are type modifiers. You can alter the data storage of a data type by

using them:

signed - allows for storage of both positive and negative numbers

unsigned - allows for storage of only positive numbers

For example,

```
// valid codes

unsigned int x = 35;

int y = -35; // signed int

int z = 36; // signed int

// invalid code: unsigned int cannot hold negative integers

unsigned int num = -35;
```

Here, the variables x and num can hold only zero and positive values because we have used

the unsigned modifier.

Considering the size of int is 4 bytes, variable y can hold values from -231 to 231-1, whereas

variable x can hold values from 0 to 232-1.

Derived Data Types

Data types that are derived from fundamental data types are derived types. For example:

arrays, pointers, function types, structures, etc.

We will learn about these derived data types in later tutorials.

bool type

Enumerated type

Complex types

C Input Output (I/O)

C Output

In C programming, printf() is one of the main output function. The function sends formatted output to the screen. For example,

Example 1: C Output

```
#include &lt;stdio.h&gt;
int main()
{
// Displays the string inside quotations
printf(&quot;C Programming&quot;);
return 0;
}
```

Output

```
C Programming
```

How does this program work?

● All valid C programs must contain the main() function. The code execution begins from the start of the main() function.

● The printf() is a library function to send formatted output to the screen. The function prints the string inside quotations.

● To use printf() in our program, we need to include stdio.h header file using the #include &lt;stdio.h&gt; statement.

● The return 0; statement inside the main() function is the &quot;Exit status&quot; of the program. It&#39;s optional.

Example 2: Integer Output

```
#include &lt;stdio.h&gt;

int main()

{

int testInteger = 5;

printf(&quot;Number = %d&quot;, testInteger);

return 0;

}
```

Output

```
Number = 5
```

We use %d format specifier to print int types. Here, the %d inside the quotations will be replaced by the value of testInteger.

Example 3: float and double Output

```
#include &lt;stdio.h&gt;

int main()

{

float number1 = 13.5;

double number2 = 12.4;

printf(&quot;number1 = %f\n&quot;, number1);

printf(&quot;number2 = %lf&quot;, number2);

return 0;

}
```

Output

```
number1 = 13.500000

number2 = 12.400000
```

To print float, we use %f format specifier. Similarly, we use %lf to print double values.

Example 4: Print Characters

```
#include &lt;stdio.h&gt;
int main()
{
char chr = &#39;a&#39;;
printf(&quot;character = %c&quot;, chr);
return 0;
}
```

Output

```
character = a
```

To print char, we use %c format specifier.

C Input

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

Example 5: Integer Input/Output

```
#include &lt;stdio.h&gt;
int main()
{
int testInteger;
printf(&quot;Enter an integer: &quot;);
scanf(&quot;%d&quot;, &amp;testInteger);
printf(&quot;Number = %d&quot;,testInteger);
return 0;
}
```

Output

```
Enter an integer: 4
Number = 4
```

Here, we have used %d format specifier inside the scanf() function to take int input

from the user. When the user enters an integer, it is stored in the testInteger variable.

Notice, that we have used &amp;testInteger inside scanf(). It is
because &amp;testInteger gets the

address of testInteger, and the value entered by the user is stored in that address.

Example 6: Float and Double Input/Output

```
#include &lt;stdio.h&gt;
int main()
{
float num1;
double num2;
printf(&quot;Enter a number: &quot;);
scanf(&quot;%f&quot;, &amp;num1);
printf(&quot;Enter another number: &quot;);
scanf(&quot;%lf&quot;, &amp;num2);
printf(&quot;num1 = %f\n&quot;, num1);
printf(&quot;num2 = %lf&quot;, num2);
return 0;
}
```

Output

```
Enter a number: 12.523
Enter another number: 10.2
num1 = 12.523000
```

```
num2 = 10.200000
```

We use %f and %lf format specifier for float and double respectively.

Example 7: C Character I/O

```
#include &lt;stdio.h&gt;
int main()
{
char chr;
printf(&quot;Enter a character: &quot;);
scanf(&quot;%c&quot;,&amp;chr);
printf(&quot;You entered %c.&quot;, chr);
return 0;
}
```

Output

```
Enter a character: g
You entered g
```

When a character is entered by the user in the above program, the character itself is not stored. Instead, an integer value (ASCII value) is stored.

And when we display that value using %c text format, the entered character is displayed. If we use %d to display the character, it&#39;s ASCII value is printed.

Example 8: ASCII Value

```
#include &lt;stdio.h&gt;
int main()
{
char chr;
printf(&quot;Enter a character: &quot;);
scanf(&quot;%c&quot;, &amp;chr);
// When %c is used, a character is displayed
printf(&quot;You entered %c.\n&quot;,chr);
// When %d is used, ASCII value is displayed
printf(&quot;ASCII value is %d.&quot;, chr);
return 0;
}
```

Output

```
Enter a character: g
You entered g.
ASCII value is 103.
```

I/O Multiple Values

Here&#39;s how you can take multiple inputs from the user and display them.

```
#include &lt;stdio.h&gt;
int main()
{
int a;
```

```
float b;
printf(&quot;Enter integer and then a float: &quot;);
// Taking multiple inputs
scanf(&quot;%d%f&quot;, &amp;a, &amp;b);
printf(&quot;You entered %d and %f&quot;, a, b);
return 0;
}
```

Output

```
Enter integer and then a float: -3
3.4
You entered -3 and 3.400000
```

Format Specifiers for I/O

As you can see from the above examples, we use

- %d for int
- %f for float
- %lf for double
- %c for char

Here&#39;s a list of commonly used C data types and their format specifiers.

| Data Type | Format Specifier |
|---|---|
| int | %d |
| char | %c |
| float | %f |
| double | %lf |
| short int | %hd |
| unsigned int | %u |
| long int | %li |
| long long int | %lli |
| unsigned long int | %lu |
| unsigned long long int | %llu |
| signed char | %c |
| unsigned char | %c |
| long double | %Lf |

The concept of operator precedence and associativity in C helps in determining which operators will be given priority when there are multiple operators in the expression.

Leap Year

```
#include &lt;stdio.h&gt;
int main() {
int year;
printf(&quot;Enter a year: &quot;);
scanf(&quot;%d&quot;, &amp;year);
// leap year if perfectly divisible by 400
if (year % 400 == 0) {
printf(&quot;%d is a leap year.&quot;, year);
}
else if (year % 100 == 0) {
printf(&quot;%d is not a leap year.&quot;, year);
}
else if (year % 4 == 0) {
printf(&quot;%d is a leap year.&quot;, year);
}
// all other years are not leap years
else {
printf(&quot;%d is not a leap year.&quot;, year);
}
return 0;
}
```

LCM using while and if

```
#include &lt;stdio.h&gt;
```

12

```
int main() {
int n1, n2, max;
printf(&quot;Enter two positive integers: &quot;);
scanf(&quot;%d %d&quot;, &amp;n1, &amp;n2);
// maximum number between n1 and n2 is stored in max
max = (n1 &gt; n2) ? n1 : n2;
while (1) {
if ((max % n1 == 0) &amp;&amp; (max % n2 == 0)) {
printf(&quot;The LCM of %d and %d is %d.&quot;, n1, n2, max);
break;
}
++max;
}
return 0;
}
```

********** OPERATOR PRECEDENCE AND ORDER OF EVALUATION

*********

```
// C Program to illustrate operator precedence

#include &lt;stdio.h&gt;
int main()
{
// printing the value of same expression
printf(&quot;10 + 20 * 30 = %d&quot;, 10 + 20 * 30);
return 0;
}
```

4. // C Program to illustrate operator Associativity

```
#include &lt;stdio.h&gt;
int main()
{
// Verifying the result of the same expression
printf(&quot;100 / 5 % 2 = %d&quot;, 100 / 5 % 2);
return 0;
}
```

5. // C Program to illustrate the precedence and associativity of the operators in an expression

```
#include &lt;stdio.h&gt;
int main()
{
// getting the result of the same expression as the
// example
```

```
int exp = 100 + 200 / 10 - 3 * 10;
printf(&quot;100 + 200 / 10 - 3 * 10 = %d&quot;, exp);
return 0;
}
```

WAP in C to take a single character .

```
#include &lt;stdio.h&gt;

#include &lt;conio.h&gt;

void main()

{

   char c;

   printf (&quot;\n Enter a character \n&quot;);

   c = getchar(); // get a single character

   printf(&quot; You have passed &quot;);

   putchar(c); // print a single character using putchar

     getch();

}
```

***** getch(), getche() and putch(): ****

```
// To input and output a single character

#include &lt;stdio.h&gt;

#include &lt;conio.h&gt;

Int main()

{

char x;

x = getch();

putch(x);

return 0;

}
```

*** STRING INPUT AND OUTPUT: ***

```
// To read and write string on screen

#include &lt;stdio.h&gt;
```

```
int main()

{

char name[20];

puts("Enter the name");

gets(name);

puts("name is");

puts(name);

return 0;

}
```

FORMATTED INPUT AND OUTPUT

```
#include &lt;stdio.h&gt;
```

```
int main()

{

int x = 123;

printf("Printing 123 using %%0d displays %od\n", x);

printf("Printing 123 using %%1d displays %1d\n", x);

printf("Printing 123 using %%2d displays %2d\n", x);

printf("Printing 123 using %%3d displays %3d\n", x);

printf("Printing 123 using %%4d displays %4d\n", x);

printf("Printing 123 using %%5d displays %5d\n", x);

printf("Printing 123 using %%6d displays %6d\n", x);

printf("Printing 123 using %%7d displays %7d\n", x);

printf("Printing 123 using %%8d displays %8d\n", x);

printf("Printing 123 using %%9d displays %9d\n", x);
printf("Printing 123 using %%09d displays %09d\n", x);
return 0;
}

#include &lt;stdio.h&gt;
int main()
{
float x = 3.141592;
printf("Printing 3.141592 using %%f \t displays %f \n", x);
printf("Printing 3.141592 using %%1.1f \t displays %1.1f \n", x);
printf("Printing 3.141592 using %%1.2f \t displays %1.2f \n", x);
printf("Printing 3.141592 using %%3.3f \t displays %3.3f \n", x);
printf("Printing 3.141592 using %%4.4f \t displays %4.4f \n", x);
printf("Printing 3.141592 using %%4.5f \t displays %4.5f \n", x);
printf("Printing 3.141592 using %%09.3f \t displays %09.3f \n", x);
printf("Printing 3.141592 using %%-09.3f \t displays %-09.3f \n", x);
printf("Printing 3.141592 using %%9.3f \t displays %9.3f \n", x);
printf("Printing 3.141592 using %%-9.3f \t displays %-9.3f \n", x);
return 0;
}
```

************************************************************************

************************************************************************

************************************************************************

# Chap 3. Control Structures

expression statement

compound statement

Sequential Execution

Types of control statements

● Decision Making Statements

● Iteration Statements or Looping

● Jump Statements

Types of Control Structures

DECISION MAKING STRUCTURES

1. if statement
2. if else statement, OR 2 way if statement
3. Nested else-if statements
4. Nested if-else statements
5. Switch statement
6. Conditional Operator

1. if statement:

SYNTAX of IF STATEMENT:

```
if (condition)
statement-1;
OR
if (condition)
{
statement-1;
statement-2;
:
statement-n;
}
```

For example: if (a&lt;0)

```
{
printf(“a is negative number”);
}
```

```
// Program to display a number if it is negative
#include &lt;stdio.h&gt;
int main() {
int number;
printf(&quot;Enter an integer: &quot;);
scanf(&quot;%d&quot;, &amp;number);
// true if the number is less than 0
if (number &lt; 0) {
printf(&quot;You entered %d.\n&quot;, number);
}
printf(&quot;The if statement is easy.&quot;);
return 0;
}
```

2. if statement:

SYNTAX of IF ELSE STATEMENT:

```
if (condition)
{
statement-1;
statement-2;
:
statement-n;
}
else
{
statement-1;
statement-2;
:
statement-n;
}
```

For example: if (a&lt;0)

```
{
```

```
printf(“a is negative number”);
}
else
{
printf(“a is positive number”);
}
```

3. Switch Statement:

```
#include &lt;stdio.h&gt;
int main()
```

```
{
char choice;
printf(“Enter color (Red - R/r, White - W/w) =”);
choice=getchar();
switch(choice)
{
case ‘r’:
case ‘R’: printf(“\n Red”);
break;
case ‘w’:
case ‘W’: printf(“\n White”);
break;
default: printf(“\n No color”);
}
```

```
// Program to add even numbers below given numbers: using
while loop
# include &lt;stdio.h&gt;
int main()
{
int m,n,sum;
printf(“enter a number”);
scanf(“%d”, &amp;n);
m=1;
while (m&lt;=n)
{
if(m%2==0)
sum = sum + m;
m=m+1;
}
printf(“ Sum of all even numbers below %d is : %d”, n, sum)
}
```

// Program to add even numbers below given numbers: using do

while loop

```
# include &lt;stdio.h&gt;
int main()
{
int m,n,sum;
printf(“enter a number”);
scanf(“%d”, &amp;n);
m=1;
do
{
if(m%2==0)
sum = sum + m;
m=m+1;
} while (m&lt;=n);
printf(“ Sum of all even numbers below %d is : %d”, n, sum)
}
```

// Program to add even numbers below given numbers:

using for loop

```
# include &lt;stdio.h&gt;
```

```
int main()

{

int m,n,sum;

printf(“enter a number”);

scanf(“%d”, &amp;n);

for (m=1; m&lt;=n ; m++)

{

if(m%2==0)

sum = sum + m;

}

printf(“ Sum of all even numbers below %d is : %d”, n, sum)

}
```

***************************************************************************

*********************************************************************

***************************************************************

Nested else-if statement:

// Program to find minimum of 3 numbers using nested if else

statement.

```
# include &lt;stdio.h&gt;

int main()

{

int a,b,c;

printf(“ Enter 3 numbers = ”);
scanf(“%d %d %d”, &amp;a, &amp;b, &amp;c);

if (a&lt;b)

{

if(a&lt;c)

printf(“ Minimum number is a = %d”,&amp;a );

else

printf(“ Minimum number is c = %d”,&amp;c );


}

else

{

if (b&lt;c)

printf(“ Minimum number is b = %d”,&amp;b);

else

printf(“ Minimum number is c = %d”,&amp;c );


}

return 0;

}
```

********************************************************************************

*** DATE : 22-08-2023

********************************************************

Flow diagram of if statement:

20

//C program to print the square of a number if it is less than 10.

```
#include&lt;stdio.h&gt;

int main()

{

int n;

printf(&quot;Enter a number:&quot;);

scanf(&quot;%d&quot;,&amp;n);

if(n&lt;10)

{

printf(&quot;%d is less than 10 :&quot;,n);

printf(&quot;Square = %d &quot;,n*n);

}

return 0;

}
```

21

Flow diagram of if-else statement:

// C program to find if a number is odd or even.

```
#include&lt;stdio.h&gt;

int main()

{

int n;

printf(&quot;Enter a number : &quot;);

scanf(&quot;%d&quot;,&amp;n);

if(n%2 == 0)

printf(&quot;%d is even&quot;,n);
else
printf(&quot;%d is odd&quot;,n);
return 0;
}
```

Flow diagram of if-else if else statement:

```
// C program to find if a number is negative, positive or zero.

#include&lt;stdio.h&gt;
int main()
{
int n;
printf(&quot;Enter a number:&quot;);
scanf(&quot;%d&quot;,&amp;n);
if(n&lt;0)
printf(&quot;Number is negative&quot;);
else if(n&gt;0)
printf(&quot;Number is positive&quot;);
else
printf(&quot;Number is equal to zero&quot;);
return 0;
}
```

Flow diagram of nested if statement:

//C program to check if a number is less than 100 or not. If it is less than

100 then check if it is odd or even.

```
#include&lt;stdio.h&gt;

int main()

{

int n;

printf(&quot;Enter a number:&quot;);

scanf(&quot;%d&quot;,&amp;n);

if(n&lt;100)

{

printf(&quot;%d is less than 100 &quot;,n);

if(n%2 == 0)

{


printf(&quot;%d is even&quot;,n);

printf(&quot; square of %d is %d &quot;,n, n*n );
```

```
}

else

{


printf(&quot;%d is odd&quot;,n);

printf(&quot; CUBE of %d is %d &quot;,n, n*n*n );

}
```

```
}
else
printf(&quot;%d is equal to or greater than 100&quot;,n);
return 0;
}
```

Flow diagram of switch case statement:

25

// Check if the entered alphabet is a vowel or a consonant.

```
#include &lt;stdio.h&gt;
int main()
{
char alphabet;
printf(&quot;Enter an alphabet:&quot;);
scanf(&quot;%c&quot;,&amp;alphabet);
switch(alphabet)
{
case &#39;a&#39;:
case ‘A’:
printf(&quot;Alphabet a/A is a vowel.&quot;);
```

26

```
break;
case &#39;e&#39;:
```

```
case ‘E’:

printf(&quot;Alphabet e/E is a vowel.&quot;);

break;

case &#39;i&#39;:

case ‘I’:

printf(&quot;Alphabet i/I is a vowel.&quot;);

break;

case &#39;o&#39;:

case ‘O’:

printf(&quot;Alphabet o/O is a vowel.&quot;);

break;

case &#39;u&#39;:

case ‘U’:

printf(&quot;Alphabet u/U is a vowel.&quot;);

break;

default:

printf(&quot;You entered a consonant.&quot;);

break;

}

return 0;

}
```

Flow diagram of while statement:

// C program to print the multiplication table of 2 from 1 to 10.

```
#include&lt;stdio.h&gt;
int main()
{
int i=1;
while(i&lt;=10)
{
printf(&quot;2 * %d = %d &quot;, i, 2*i);
i++;
}
return 0;
}
```

Flow diagram of do while statement:

```
// C program to print the table of 5 from 1 to 10.

#include&lt;stdio.h&gt;
int main()
{
int i=1;
do
{
printf(&quot;5 * %d = %d\n&quot;,i,5*i);
i++;
}while(i&lt;=10);
return 0;
```

}

Flow diagram of for statement:

29

```
//C program to find the sum of first n natural numbers.

#include&lt;stdio.h&gt;
int main()
{
int i,n,s=0;
printf(&quot;Enter value of n:&quot;);
scanf(&quot;%d&quot;,&amp;n);
for(i=1;i&lt;=n;i++)
{
s=s+i;
}
printf(&quot;Sum = %d&quot;,s);
return 0;
}
```

DATE: 23-08-2023

30

Break

It was used to &quot;jump out&quot; of a switch statement.

The break statement can also be used to jump out of a loop.

This example jumps out of the for loop when i is equal to 4:

```
int i;
for (i = 0; i &lt; 10; i++)
{
if (i == 4)
{
break;
}
printf(&quot;%d\n&quot;, i);
}
```

Output

```
0
1
2
3
```

Continue

The continue statement breaks one iteration (in the loop), if a specified condition occurs, and continues with the next iteration in the loop.

This example skips the value of 4:

```
#include &lt;stdio.h&gt;
int main()
{
int i;
for (i = 0; i &lt; 10; i++)
{
if (i == 4)

{

continue;
}
printf(&quot;%d\n&quot;, i);
}
return 0;
}
```

OUTPUT:

```
0
1
2
3
5
6
7
8
9
```

Break Statement Continue Statement

The Break statement is used to exit

from the loop constructs.

The continue statement is not used
to exit from the loop constructs.

The break statement is usually used
with the switch statement, and it
can also be used within the while
loop, do-while loop, or the for-loop.

The continue statement is not used
with the switch statement, but it
can be used within the while loop,
do-while loop, or for-loop.

32

When a break statement is
encountered then the control is
exited from the loop construct
immediately.

When the continue statement is
encountered then the control
automatically passes from the
beginning of the loop statement.

Syntax:

```
break;
```

Syntax:

continue;

Break statements use switch and

label statements.

It does not use switch and label

statements.

Leftover iterations are not executed

after the break statement.

Leftover iterations can be executed

even if the continue keyword

appears in a loop.

```
//Program to reverse a number
#include &lt;stdio.h&gt;
int main()
{
int n, reverse = 0, remainder;
printf(&quot;Enter an integer: &quot;);
scanf(&quot;%d&quot;, &amp;n);
while (n != 0)
```

```
{
remainder = n % 10;
reverse = reverse * 10 + remainder;
n /= 10;
}
printf(&quot;Reversed number = %d&quot;, reverse);
return 0;
}

// Armstrong number

#include &lt;stdio.h&gt;
int main() {
int num, originalNum, remainder, result = 0;
printf(&quot;Enter a three-digit integer: &quot;);
scanf(&quot;%d&quot;, &amp;num);
originalNum = num;
while (originalNum != 0)
{
// remainder contains the last digit
remainder = originalNum % 10;
result += remainder * remainder * remainder;
// removing last digit from the orignal number
originalNum /= 10;
}
if (result == num)
printf(&quot;%d is an Armstrong number.&quot;, num);
else
```

```
printf(&quot;%d is not an Armstrong number.&quot;, num);
return 0;
}
```

34

DATE: 24-08-2023

```
// Program to input a number from the user and find all divisors of the given
number.
#include &lt;stdio.h&gt;
int main()
{
int i,num;
printf(“Enter number to find its factors : \t”);
scanf(“%d”, &amp;num);
printf(“All factors of %d are : \n”, num);
for (i=1; i&lt;=num; i++)

{
if (num%i == 0)
{
printf(“%d ”, i)
}
}
return 0;

}
// Program to check given number is prime or not
```

```
#include &lt;stdio.h&gt;
int main() {
int n, i, flag = 0;
printf(&quot;Enter a positive integer: &quot;);
scanf(&quot;%d&quot;, &amp;n);
// 0 and 1 are not prime numbers
// change flag to 1 for non-prime number
if (n == 0 || n == 1)
flag = 1;
for (i = 2; i &lt;= n / 2; ++i)
{
if (n % i == 0)
{
flag = 1;
break;
}
}
// flag is 0 for prime numbers
if (flag == 0)
```

```
printf(&quot;%d is a prime number.&quot;, n);
else
printf(&quot;%d is not a prime number.&quot;, n);
return 0;
}
```

```
//Program to print all Prime numbers between a given range.

#include &lt;stdio.h&gt;

int main()

{

int i,j, start, end, flg;

//input upper and lower limit to print prime.

printf(“Enter lower limit : \t”);

scanf(“%d”, &amp;start);

printf(“Enter upper limit : \t”);

scanf(“%d”, &amp;end);

printf(“All prime numbers between %d to %d are :\n”,start, end);

if (start &lt; 2)

start = 2;

/* Find all Prime numbers between range */

for (i=start; i&lt;=end; i++)

{

flg = 1;


// check if the current number i is prime or not


for (j=2; j&lt;=i/2; j++)

{

if(i%j == 0)

{

flg = 0;

break;

}
}

//If number is prime then print
if (flag == 1)
{
printf("%d ", i);
}
}
return 0;
```

```
}

//Program to print multiplication tables from 1 to 5.

#include &lt;stdio.h&gt;
int main()
{
// Loop counter variable declaration
int i,j;
// Outer Loop
for (i=1; i&lt;=10; i++)

{
//Inner loop
for (j=1; j&lt;=5; j++)
{
```

```
printf(“%d\t”,(i*j));
}
printf(“\n”);
}
return 0;

}
```

OUTPUT:

```
1 2 3 4 5
2 4 6 8 10
3 6 9 12 15
4 8 12 16 20
5 10 15 20 25
```

```
6 12 18 24 30
7 14 21 28 35
8 16 24 32 40
9 18 27 36 45
10 20 30 40 50
```

```
//Program to print following pattern:
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

```
#include &lt;stdio.h&gt;

int main()

{

int i,j;

for (i=1; i&lt;=5; i++)

{

for (j=1; j&lt;=i; j++)

{

printf(“%5d”,j);

}

printf(“\n\n”);

}

return 0;

}
```

DATE: 26/08/2023

```
// C program to print right half pyramid pattern of star

#include &lt;stdio.h&gt;

int main()

{

int rows = 5;

// first loop for printing rows

for (int i = 0; i &lt; rows; i++)

{

// second loop for printing character in each rows
for (int j = 0; j &lt;= i; j++)

{
printf(&quot;* &quot;);
}
printf(&quot;\n&quot;);
}
return 0;
}
```

OUTPUT:

```
*
* *
* * *
* * * *
* * * * *
```

```
// C program to print right half pyramid pattern of number
#include &lt;stdio.h&gt;
```

```
int main()
{
int rows = 5;
// first loop for printing rows
for (int i = 0; i &lt; rows; i++)
{
// second loop for printing number in each rows
for (int j = 0; j &lt;= i; j++)

{
printf(&quot;%d &quot;, j + 1);
}
printf(&quot;\n&quot;);
}
return 0;
}
```

OUTPUT:

```
1
12
123
1234
12345
```

```
// C program to print right half pyramid pattern of
// alphabets
```

```
#include &lt;stdio.h&gt;
int main()
{
int rows = 5;
// first loop for printing rows
for (int i = 0; i &lt; rows; i++)
{
```

```
// second loop for printing alphabets in each rows
for (int j = 0; j &lt;= i; j++)

{
printf(&quot;%c &quot;, &#39;A&#39; + j);
}
printf(&quot;\n&quot;);
}
return 0;
}
```

OUTPUT:

```
A
AB
ABC
ABCD
ABCDE
```

Date : 31/08/2023

Example 1: Print a pattern

```
* * * * *
* * * *
* * *
* *
*
```

```
#include &lt;stdio.h&gt;

int main()

{

int i, j, rows;

printf(&quot;Enter the number of rows: &quot;);

scanf(&quot;%d&quot;, &amp;rows);

for (i = rows; i &gt;= 1; --i)

{

for (j = 1; j &lt;= i; ++j)

{

printf(&quot;* &quot;);

}

printf(&quot;\n&quot;);

}

return 0;

}
```

Example 2: Print a pattern

42

```
1 2 3 4 5

1 2 3 4

1 2 3

1 2

1
```

```
#include &lt;stdio.h&gt;
```

```
int main()

{

int i, j, rows;

printf(&quot;Enter the number of rows: &quot;);

scanf(&quot;%d&quot;, &amp;rows);

for (i = rows; i &gt;= 1; --i)

{

for (j = 1; j &lt;= i; ++j)

{

printf(&quot;%d &quot;, j);

}

printf(&quot;\n&quot;);

}

return 0;

}
```

Example 3: Full Pyramid of *

43

```
*

* * *

* * * * *

* * * * * * *

* * * * * * * * *
```

C Program

```
#include &lt;stdio.h&gt;
```

```
int main()

{

int i, space, rows, k;

printf(&quot;Enter the number of rows: &quot;);

scanf(&quot;%d&quot;, &amp;rows);

for (i = 1; i &lt;= rows; ++i, k = 0)

{

for (space = 1; space &lt;= rows - i; ++space)

{

printf(&quot; &quot;);

}

while (k != 2 * i - 1)

{

printf(&quot;* &quot;);

++k;

}

printf(&quot;\n&quot;);

}

return 0;

}
```

Example 4: Full Pyramid of Numbers

44

```
1

2 3 2

3 4 5 4 3
```

```
4 5 6 7 6 5 4

5 6 7 8 9 8 7 6 5
```

C Program

```
#include &lt;stdio.h&gt;

int main()

{

int i, space, rows, k = 0, count = 0, count1 = 0;

printf(&quot;Enter the number of rows: &quot;);

scanf(&quot;%d&quot;, &amp;rows);

for (i = 1; i &lt;= rows; ++i)

{

for (space = 1; space &lt;= rows - i; ++space)

{

printf(&quot; &quot;);

++count;

}

while (k != 2 * i - 1)


{


if (count &lt;= rows - 1)


{


printf(&quot;%d &quot;, i + k);

++count;

}
else
{
++count1;
printf(&quot;%d &quot;, (i + k - 2 * count1));

}
++k;
}
count1 = count = k = 0;
printf(&quot;\n&quot;);
}
return 0;
```

```
}
```

Example 5: Inverted full pyramid of *

```
* * * * * * * * *
* * * * * * *
* * * * *
* * *
*
```

C Program

```
#include &lt;stdio.h&gt;
int main()
```

```
{
int rows, i, j, space;
printf(&quot;Enter the number of rows: &quot;);
scanf(&quot;%d&quot;, &amp;rows);
for (i = rows; i &gt;= 1; --i)
{
for (space = 0; space &lt; rows - i; ++space)
printf(&quot; &quot;);
for (j = i; j &lt;= 2 * i - 1; ++j)
printf(&quot;* &quot;);
for (j = 0; j &lt; i - 1; ++j)
printf(&quot;* &quot;);
printf(&quot;\n&quot;);
}
return 0;
}
```

Example 6: Pascal&#39;s Triangle

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
```

C Program

```
#include &lt;stdio.h&gt;

int main()

{

int rows, coef = 1, space, i, j;

printf(&quot;Enter the number of rows: &quot;);

scanf(&quot;%d&quot;, &amp;rows);

for (i = 0; i &lt; rows; i++)

{

for (space = 1; space &lt;= rows - i; space++)

printf(&quot; &quot;);

for (j = 0; j &lt;= i; j++)

{

if (j == 0 || i == 0)

coef = 1;

else

coef = coef * (i - j + 1) / j;

printf(&quot;%4d&quot;, coef);

}

printf(&quot;\n&quot;);

}

return 0;

}
```

Example 7: Floyd&#39;s Triangle.

```
1

2 3
```

```
4 5 6

7 8 9 10
```

C Program

```
#include &lt;stdio.h&gt;

int main()

{

int rows, i, j, number = 1;

printf(&quot;Enter the number of rows: &quot;);

scanf(&quot;%d&quot;, &amp;rows);

for (i = 1; i &lt;= rows; i++)

{

for (j = 1; j &lt;= i; ++j)

{

printf(&quot;%d &quot;, number);

++number;

}

printf(&quot;\n&quot;);

}

return 0;

}
```

**********************************************************************

**********************************************************************

# Ch. 4 C Functions

- Some programs might have thousands or millions of lines and to manage such programs it becomes quite difficult as there might be too many syntax errors or logical errors present in the program. To manage such types of programs, the concept of Modular Programming is used.
- The entire program is divided into a series of modules and each module is intended to perform a particular task. The detailed work to be solved by the module is described in the module (sub program) only and the main program only contains a series of modules that are to be executed.
- Modular programming emphasis on breaking large programs into small problems to increase the maintainability, readability of the code and to make the program handy to make any changes in future or to correct the errors.
- Sometimes there is a need for us to repeatedly execute in many parts of the program, then repeated coding as well as wastage of vital computer resource memory will be wasted.

Concept of Function

a named independent or self contained block of C code that performs a specific task and optionally returns a value to the calling program or may receive values from the calling program.

Characteristics:

1. Unique Name:
2. Performs a Specific Task
3. Independent
4. May receive values from the calling program (Caller)
5. May receive values to the calling program

Types of Function:

A. Predefined or standard library functions

B. User Defined Functions

49

Advantages of Modular Design:

1. Avoid Redundancy
2. Reusability of Code
3. Use of Customized Library
4. Better Readability and the Logical Clarity
5. Easy Maintenance and Debugging
6. Abstraction
7. Portability

Standard Library Functions:

– to use library function we have to use corresponding header file using the preprocessor directive #include.

ex. to use input output functions like printf() and scanf() we have to include &lt;stdio.h&gt;, for mathematical functions we have to include &lt;math.h&gt;

– all character handling functions are defined in ctype.h.

– These functions take the int equivalent of one character as a parameter and return an int that can either be another character or a value representing a boolean value namely true(non zero) or false(zero).

1. int isalnum(int c) = check passed char is alphanumeric
2. int isalpha(int c) = check passed char is alphabetic
3. int iscntrl(int c) = control character
4. int isdigit(int c) = decimal digit
5. int isgraph(int c) = graphical representation using locale
6. int islower(int c) = lowercase letter

7. int isprint(int c) = printable character

8. int ispunct(int c) = punctuation character

9. int isspace(int c) = white space

10. int isupper(int c) = uppercase letter

11. isxdigit(int c) = hexadecimal digit

two conversion functions that accepts and returns an int

1. int tolower(int c)

2. int toupper(int c)

50

Prog. 1: isalnum() function return value.

```
#include &lt;stdio.h&gt;
#include &lt;ctype.h&gt;
int main()
{
char c;
int result;
c = &#39;5&#39;;
result = isalnum(c);
printf(&quot;When %c is passed, return value is %d\n&quot;, c, result);
c = &#39;Q&#39;;
result = isalnum(c);
printf(&quot;When %c is passed, return value is %d\n&quot;, c, result);
c = &#39;l&#39;;
result = isalnum(c);
printf(&quot;When %c is passed, return value is %d\n&quot;, c, result);
c = &#39;+&#39;;
result = isalnum(c);
printf(&quot;When %c is passed, return value is %d\n&quot;, c, result);
return 0;
}
```

Output

```
When 5 is passed, return value is 1
When Q is passed, return value is 1
When l is passed, return value is 1
When + is passed, return value is 0
```

Prog.2 To check whether the entered character is uppercase or lowercase or space.

```
#include &lt;stdio.h&gt;
#include &lt;ctype.h&gt;
int main()
{
char c;
printf(“Enter character: ”);
```

```
scanf(“%c”,&amp;c);
if (isupper(c))
printf(“%c is an uppercase character \n”, c);
else if (islower(c))
printf(“%c is an lowercase character \n”, c);
else if (isspace(c))
printf(“%c is space \n”, c);
```

```
else

printf("%c is none from uppercase, lowercase and space. \n", c);

return 0;

}
```

Prog. 3 Print all Punctuations.

```
#include &lt;stdio.h&gt;

#include &lt;ctype.h&gt;

int main()

{

int i;

printf(&quot;All punctuations in C: \n&quot;);

// looping through all ASCII characters

for (i = 0; i &lt;= 127; ++i)

if(ispunct(i)!= 0)

printf(&quot;%c &quot;, i);

return 0;

}
```

Prog. 4: Program to find factorial of a given number.

```
#include&lt;stdio.h&gt;

long int multiplyNumbers(int n);

int main() {

int n;

printf(&quot;Enter a positive integer: &quot;);

scanf(&quot;%d&quot;,&amp;n);

printf(&quot;Factorial of %d = %ld&quot;, n, multiplyNumbers(n));

return 0;

}
```

```
long int multiplyNumbers(int n) {
if (n&gt;=1)
return n*multiplyNumbers(n-1);
else
return 1;
}
```

Prog. 5: Program to Convert Decimal to Octal

```
#include &lt;stdio.h&gt;
#include &lt;math.h&gt;
int convertDecimalToOctal(int decimalNumber);
// function prototype
int main()
{
int decimalNumber;
printf(&quot;Enter a decimal number: &quot;);
scanf(&quot;%d&quot;, &amp;decimalNumber);
printf(&quot;%d in decimal = %d in octal&quot;, decimalNumber,
convertDecimalToOctal(decimalNumber));
return 0;
}
// function to convert decimalNumber to octal
int convertDecimalToOctal(int decimalNumber)
{
int octalNumber = 0, i = 1;
while (decimalNumber != 0)

{

octalNumber += (decimalNumber % 8) * i;

decimalNumber /= 8;

i *= 10;

}

return octalNumber;

}
```

Prog. 6: Write a C Program to convert decimal to binary.

A function in C is a set of statements that when called perform some specific task. 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.

We will learn about functions, function definition, declaration, arguments and parameters, return values, and many more.

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 arguments.

A function declaration tells the compiler that there is a function with the given name defined somewhere else in the program.

Syntax

```
return_type function_name (datatype arg1, datatype arg1);
```

The argument 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);
int sum(int , int);
```

Function Declaration

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

Function Definition

The function definition contains the block of code to perform a specific task. It may include a list of the arguments, return type and local declarations.

```
return_type function_name (datatype arg1, datatype arg2, ..., datatype argn)
{
local variable declaration;
set of statements - Block of code // body of the function
}
```

where,

● First line of function definition must be same as function header or prototype

● Second line may be variable or constant declaration to that function.

● Set of statements performs some specific task.

● Function may contain return statement.

Example:

```
int max(int a, int b)
{
```

```
int result;
if(a &gt; b)
result = a;
else
result = b;
return result;
}
```

Note that: Function can be defined anywhere in the program means before or after main() function, but we can’t define function inside another function definition.

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.

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

```
// C program to show function call and definition
#include &lt;stdio.h&gt;
// Function that takes two parameters a and b as inputs and //returns their
sum
int sum(int a, int b)
{
return a + b;
}
int main()
{
//Calling sum function and storing its value in add variable
int add = sum(10, 30);
printf(&quot;Sum is: %d&quot;, add);
return 0;
}
```

Output

```
Sum is: 40
```

Here is an example to add two integers. To perform this task, we have created an user-defined addNumbers().

```
#include &lt;stdio.h&gt;
int addNumbers(int a, int b); // function prototype
int main()
{
int n1,n2,sum;
printf(&quot;Enters two numbers: &quot;);
scanf(&quot;%d %d&quot;,&amp;n1,&amp;n2);
sum = addNumbers(n1, n2); // function call
printf(&quot;sum = %d&quot;,sum);
return 0;
}
int addNumbers(int a, int b) // function definition
{
int result;
result = a+b;
return result; // return statement
}
```

Passing arguments to a function

In programming, argument refers to the variable passed to the function. In the above example, two variables n1 and n2 are passed during the function call.

The parameters a and b accept the passed arguments in the function definition.

These arguments are called formal parameters of the function.

Passing Argument to Function

The type of arguments passed to a function and the formal parameters must match, otherwise, the compiler will throw an error.

If n1 is of char type, a also should be of char type. If n2 is of float type, variable b also should be of float type.

A function can also be called without passing an argument.

Return Statement

The return statement terminates the execution of a function and returns a value to the calling function. The program control is transferred to the calling function after the return statement.

58

In the above example, the value of the result variable is returned to the main function. The sum variable in the main() function is assigned this value.

Return Statement of Function

Syntax of return statement

```
return (expression);
```

For example,

```
return a;
return (a+b);
```

The type of value returned from the function and the return type specified in the function prototype and function definition must match.

59

Example 1: Program to add to numbers by using function

```
#include&lt;stdio.h&gt;
void sum();
void main()
{
printf(&quot;\nGoing to calculate the sum of two numbers:&quot;);
sum();
}
void sum()
{
int a,b;
printf(&quot;\nEnter two numbers&quot;);

scanf(&quot;%d %d&quot;,&amp;a,&amp;b);

printf(&quot;The sum is %d&quot;,a+b);

}
```

Output:

```
Going to calculate the sum of two numbers:

Enter two numbers 10

24

The sum is 34
```

60

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

```
#include&lt;stdio.h&gt;

int even_odd(int);

void main()

{

int n,flag=0;

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

printf(&quot;\nEnter the number: &quot;);

scanf(&quot;%d&quot;,&amp;n);

flag = even_odd(n);

if(flag == 0)

{

printf(&quot;\nThe number is odd&quot;);

}

else

{
```

```
printf(&quot;\nThe number is even&quot;);
}
}
int even_odd(int n)
{
if(n%2 == 0)
{
return 1;
}
else
{
return 0;
}
}
```

Example 3: program to calculate the area of the square

```
#include&lt;stdio.h&gt;
int square();
void main()
{
printf(&quot;Going to calculate the area of the square\n&quot;);
float area = square();
printf(&quot;The area of the square: %f\n&quot;,area);
}
int square()
```

```
{
float side;
printf(&quot;Enter the length of the side in meters: &quot;);
scanf(&quot;%f&quot;,&amp;side);
return side * side;
}
```

Output

```
Going to calculate the area of the square
Enter the length of the side in meters: 10
The area of the square: 100.000000
```

Example 4: program to calculate the average of five numbers.

```
#include&lt;stdio.h&gt;
void average(int, int, int, int, int);
void main()
{
int a,b,c,d,e;
printf(&quot;\nGoing to calculate the average of five numbers:&quot;);
printf(&quot;\nEnter five numbers:&quot;);
scanf(&quot;%d %d %d %d %d&quot;,&amp;a,&amp;b,&amp;c,&amp;d,&amp;e);
average(a,b,c,d,e);
}
void average(int a, int b, int c, int d, int e)
{
float avg;
avg = (a+b+c+d+e)/5;
printf(&quot;The average of given five numbers : %f&quot;,avg);
}
```

63

Example 5: Program to Check Prime and Armstrong

```
#include &lt;math.h&gt;
#include &lt;stdio.h&gt;
int checkPrimeNumber(int n);
int checkArmstrongNumber(int n);
int main() {
int n, flag;
printf(&quot;Enter a positive integer: &quot;);
scanf(&quot;%d&quot;, &amp;n);
// check prime number
flag = checkPrimeNumber(n);
if (flag == 1)
printf(&quot;%d is a prime number.\n&quot;, n);
else
printf(&quot;%d is not a prime number.\n&quot;, n);
// check Armstrong number
flag = checkArmstrongNumber(n);
if (flag == 1)
printf(&quot;%d is an Armstrong number.&quot;, n);
else
printf(&quot;%d is not an Armstrong number.&quot;, n);
return 0;
```

```
}
// function to check prime number
int checkPrimeNumber(int n) {
int i, flag = 1, squareRoot;
// computing the square root
squareRoot = sqrt(n);
```

```
for (i = 2; i &lt;= squareRoot; ++i) {
// condition for non-prime number
if (n % i == 0) {
flag = 0;
break;
}
}
return flag;
}
// function to check Armstrong number
int checkArmstrongNumber(int num) {
int originalNum, remainder, n = 0, flag;
double result = 0.0;
// store the number of digits of num in n
for (originalNum = num; originalNum != 0; ++n) {
originalNum /= 10;
}
for (originalNum = num; originalNum != 0; originalNum /= 10) {
remainder = originalNum % 10;
// store the sum of the power of individual digits in result
result += pow(remainder, n);
}
// condition for Armstrong number
if (round(result) == num)
flag = 1;
else
flag = 0;
return flag;
}
```

Parameter Passing (By Value):

● Function parameters or arguments are the means of communication between the calling and called functions. The function i.e. main() and user-defined function interact with each other through parameters.

● Parameters may classify under two groups namely, Formal Parameters and Actual Parameters.

○ Formal or Dummy Parameters : parameters given in function declaration and function definition. When function is invoked,the formal parameters are replaced by the actual parameters.

○ Actual Parameter: Parameters appearing in the function call are referred to as actual parameters. Actual parameters must be the same as the number of formal parameters, and each actual parameter must be of the same data type as its corresponding formal parameter.

Call by value in C

In the call by value method, the value of the actual parameters is copied into the formal parameters. In other words, we can say that the value of the variable is used in the function call in the call by value method.
In the call by value method, we can not modify the value of the actual parameter by the formal parameter.
In call by value, different memory is allocated for actual and formal parameters since the value of the actual parameter is copied into the formal parameter.
The actual parameter is the argument which is used in the function call whereas the formal parameter is the argument which is used in the function definition.

Let&#39;s try to understand the concept of call by value in c language by the example given below:

```
#include&lt;stdio.h&gt;
void change(int num)
{ printf(&quot;Before adding value inside function num=%d \n&quot;,num);
num=num+100;
printf(&quot;After adding value inside function num=%d \n&quot;, num);
}
int main()
{ int x=100;
printf(&quot;Before function call x=%d \n&quot;, x);
change(x);//passing value in function
printf(&quot;After function call x=%d \n&quot;, x);
return 0;
}
```

Output

```
Before function call x=100
Before adding value inside function num=100
After adding value inside function num=200
After function call x=100
```

Call by Value Example: Swapping the values of the two variables

```
#include &lt;stdio.h&gt;
void swap(int , int); //prototype of the function
int main()
{
int a = 10;
int b = 20;
printf(&quot;Before swapping the values in main a = %d, b = %d\n&quot;,a,b);
```

```
// printing the value of a and b in main
swap(a,b);
printf(&quot;After swapping values in main a = %d, b = %d\n&quot;,a,b);
// The value of actual parameters do not change by changing the formal
parameters in call by value, a = 10, b = 20
}
void swap (int a, int b)
{
int temp;
temp = a;
```

```
a=b;
b=temp;
printf(&quot;After swapping values in function a = %d, b = %d\n&quot;,a,b);
// Formal parameters, a = 20, b = 10
}
```

Output

```
Before swapping the values in main a = 10, b = 20
After swapping values in function a = 20, b = 10
After swapping values in main a = 10, b = 20
```

Call by reference in C

In call by reference, the address of the variable is passed into the function call as the actual parameter.

The value of the actual parameters can be modified by changing the formal parameters since the address of the actual parameters is passed.

In call by reference, the memory allocation is similar for both formal parameters and actual parameters. All the operations in the function are performed on the value stored at the address of the actual parameters, and the modified value gets stored at the same address.

Consider the following example for the call by reference.

```
#include&lt;stdio.h&gt;
void change(int *num)
{
printf(&quot;Before adding value inside function num=%d \n&quot;,*num);
(*num) += 100;
printf(&quot;After adding value inside function num=%d \n&quot;, *num);

}

int main()

{

int x=100;

printf(&quot;Before function call x=%d \n&quot;, x);

change(&amp;x); //passing reference in function

printf(&quot;After function call x=%d \n&quot;, x);

return 0;

}
```

Output

```
Before function call x=100

Before adding value inside function num=100

After adding value inside function num=200

After function call x=200
```

Call by reference Example: Swapping the values of the two variables

```
#include &lt;stdio.h&gt;

void swap(int *, int *); //prototype of the function

int main()

{

int a = 10;

int b = 20;

printf(&quot;Before swapping the values in main a = %d, b = %d\n&quot;,a,b); //

printing the value of a and b in main

swap(&amp;a,&amp;b);

printf(&quot;After swapping values in main a = %d, b = %d\n&quot;,a,b); // The values
```

```
of actual parameters do change in call by reference, a = 10, b = 20
}
void swap (int *a, int *b)
{
int temp;
temp = *a;
*a=*b;
```

```
*b=temp;
printf(&quot;After swapping values in function a = %d, b = %d\n&quot;,*a,*b); // Formal
parameters, a = 20, b = 10
}
```

Output

```
Before swapping the values in main a = 10, b = 20
After swapping values in function a = 20, b = 10
After swapping values in main a = 20, b = 10
```

Return Statement: The return statement is used to terminate the execution of a function and transfer program control back to the calling function.

In addition, it can specify a value to be returned by the function.

– A function may contain one or more return statements.

Syntax: return; or return expr;

– Execution of the return statement causes the expression expr to be evaluated and its value to be returned to the point from which this function is called.

– Only one expression can be included in the return statement.

Recursive Function:

1. Direct Recursion

Example: Fun()

```
{
—-------
—-------
Fun();
}
```

2. Indirect Recursion

Example:

```
Fun1()
{
—-------
—-------
Fun2();
}
Fun2()
{
—-------
—-------
Fun1();
}
```

Prog. 4: Program to find factorial of a given number.

```
#include&lt;stdio.h&gt;
```

```
long int multiplyNumbers(int n);
int main() {
int n;
printf(&quot;Enter a positive integer: &quot;);
scanf(&quot;%d&quot;,&amp;n);
printf(&quot;Factorial of %d = %ld&quot;, n, multiplyNumbers(n));
return 0;
}
long int multiplyNumbers(int n) {
if (n&gt;=1)
return n*multiplyNumbers(n-1);
else
return 1;
}
```

Scope of Variables:

```
#include&lt;stdio.h&gt;
int main()
{
int x = 10, y = 20;
{
printf(&quot;x = %d, y = %d\n&quot;, x, y);
{
int y = 40;
x++;
y++;
printf(&quot;x = %d, y = %d\n&quot;, x, y);
}
printf(&quot;x = %d, y = %d\n&quot;, x, y);
}
return 0;
}
```

```
x = 10, y = 20
x = 11, y = 41
x = 11, y = 20
```

## C Storage Class

Every variable in C programming has two properties: type and storage class.

Type refers to the data type of a variable. And, storage class determines the scope, visibility and lifetime of a variable.

There are 4 types of storage class:

1. automatic
2. external
3. static
4. register

### Local Variable

The variables declared inside a block are automatic or local variables.

The local variables exist only inside the block in which it is declared.

Let&#39;s take an example.

```
#include &lt;stdio.h&gt;
int main(void)
{
for (int i = 0; i &lt; 5; ++i)
{
printf(&quot;C programming&quot;);
}
// Error: i is not declared at this point
printf(&quot;%d&quot;, i);
return 0;
}
```

When you run the above program, you will get an error undeclared identifier i. It&#39;s because i is declared inside the for loop block. Outside of the block, it&#39;s undeclared.

Let&#39;s take another example.

```
int main() {
int n1; // n1 is a local variable to main()
}
void func() {
int n2; // n2 is a local variable to func()
}
```

In the above example, n1 is local to main() and n2 is local to func().

This means you cannot access the n1 variable inside func() as it only exists inside main(). Similarly, you cannot access the n2 variable inside main() as it only exists inside func().

Global Variable

Variables that are declared outside of all functions are known as external or global variables. They are accessible from any function inside the program.

Example 1: Global Variable

```
#include &lt;stdio.h&gt;
void display();
int n = 5; // global variable
int main()
{
++n;
display();
return 0;
}
void display()
{
++n;
printf(&quot;n = %d&quot;, n);
}
```

Output

75

```
n = 7
```

Suppose, a global variable is declared in file1. If you try to use that variable in a different file file2, the compiler will complain. To solve this problem, keyword extern is used in file2 to indicate that the external variable is declared in another file.

Register Variable

The register keyword is used to declare register variables. Register variables were supposed to be faster than local variables.

However, modern compilers are very good at code optimization, and there is a rare chance that using register variables will make your program faster.

Unless you are working on embedded systems where you know how to optimize code for the given application, there is no use of register variables.

Static Variable

A static variable is declared by using the static keyword. For example;

```
static int i;
```

The value of a static variable persists until the end of the program.

Example 2: Static Variable

76

```
#include &lt;stdio.h&gt;
void display();
int main()
{
display();
display();
}
void display()
{
static int c = 1;
c += 5;
printf(&quot;%d &quot;,c);
}
```

Output

```
6 11
```

During the first function call, the value of c is initialized to 1. Its value has increased by 5. Now, the value of c is 6, which is printed on the screen.

During the second function call, c is not initialized to 1 again. It&#39;s because c is a static variable. The value c is increased by 5. Now, its value will be 11, which is printed on the screen.

**********************************************************************

**********************************************************************

**********************************************************************

# Ch. 5 C Arrays

Array in C is one of the most used data structures in C programming. It is a simple and fast way of storing multiple values under a single name. In this article, we will study the different aspects of array in C language such as array declaration, definition, initialization, types of arrays, array syntax, advantages and disadvantages, and many more.

What is Array in C?

Array can be defined as, “a collection of data items of same data types which are stored in consecutive memory locations and referred by common name.”

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.

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.

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

Example of Array Declaration C

```
// C Program to illustrate the array declaration

#include &lt;stdio.h&gt;

int main()

{

// declaring array of integers
```

```
int arr_int[5];

// declaring array of characters

char arr_char[5];

return 0;
```

```
}
```

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 b a comma.

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

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 &lt; N; i++) {
array_name[i] = valuei;
```

```
}
```

Example of Array Initialization in C

```
// C Program to demonstrate array initialization

#include &lt;stdio.h&gt;

int main()

{

// array initialization using initialier list

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

// array initialization using initializer list without

// specifying size

int arr1[] = { 1, 2, 3, 4, 5 };
```

82

```
// array initialization using for loop

float arr2[5];

for (int i = 0; i &lt; 5; i++) {

arr2[i] = (float)i * 2.1;

}

return 0;

}
```

Access 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.

83

Example of Accessing Array Elements using Array Subscript Operator

C

```
// C Program to illustrate element access using array

// subscript

#include &lt;stdio.h&gt;

int main()

{

84

// array declaration and initialization

int arr[5] = { 15, 25, 35, 45, 55 };

// accessing element at index 2 i.e 3rd element

printf(&quot;Element at arr[2]: %d\n&quot;, arr[2]);

// accessing element at index 4 i.e last element

printf(&quot;Element at arr[4]: %d\n&quot;, arr[4]);

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

return 0;

}
```

Output

```
Element at arr[2]: 35

Element at arr[4]: 55

85

Element at arr[0]: 15
```

Update Array Element

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;
```

C Array Traversal

Traversal is the process in which we visit every element of the 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 &lt; N; i++) {
array_name[i];
}
```

86

How to use Array in C?

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

C

```
// C Program to demonstrate the use of array

#include &lt;stdio.h&gt;

int main()

87

{

// 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(&quot;Elements in Array: &quot;);

for (int i = 0; i &lt; 5; i++) {

printf(&quot;%d &quot;, arr[i]);

}

return 0;

}
```

Output

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

Types of Array 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

```
// C Program to illustrate the use of 1D array

#include &lt;stdio.h&gt;

int main()

{

// 1d array declaration

int arr[5];

// 1d array initialization using for loop

for (int i = 0; i &lt; 5; i++) {

arr[i] = i * i - 2 * i + 1;

90

}
printf(&quot;Elements of Array: &quot;);

// printing 1d array by traversing using for loop

for (int i = 0; i &lt; 5; i++) {

printf(&quot;%d &quot;, 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

```
// C Program to illustrate strings

#include &lt;stdio.h&gt;
```

```
int main()

{

// creating array of character

char arr[6] = { &#39;G&#39;, &#39;e&#39;, &#39;e&#39;, &#39;k&#39;, &#39;s&#39;,
&#39;\0&#39; };

// printing string

int i = 0;

while (arr[i]) {

92

printf(&quot;%c&quot;, arr[i++]);

}

return 0;

}
```

Output

```
Geeks
```

To know more about strings, refer to this article – Strings in C

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

93

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

Here,

● size1: Size of the first dimension.

● size2: Size of the second dimension.

Example of 2D Array in C

C

94

```
// C Program to illustrate 2d array
```

```
#include &lt;stdio.h&gt;

int main()

{

// declaring and initializing 2d array

int arr[2][3] = { 10, 20, 30, 40, 50, 60 };

printf(&quot;2D Array:\n&quot;);

// printing 2d array

for (int i = 0; i &lt; 2; i++) {

for (int j = 0; j &lt; 3; j++) {

printf(&quot;%d &quot;,arr[i][j]);

95

}

printf(&quot;\n&quot;);

}
return 0;

}
```

Output

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

// Prog 1. Program to take 5 values from the user and store them in an

array and Print the elements stored in the array.

```
#include &lt;stdio.h&gt;
int main()
{
int values[5];
printf(&quot;Enter 5 integers: &quot;);
// taking input and storing it in an array
for(int i = 0; i &lt; 5; ++i)
{
scanf(&quot;%d&quot;, &amp;values[i]);
}
printf(&quot;Displaying integers: &quot;);
// printing elements of an array
for(int i = 0; i &lt; 5; ++i)
{
printf(&quot;%d\n&quot;, values[i]);
}
return 0;
}
```

97

```
//Prog. 2. Program to find the average of n numbers using arrays
#include &lt;stdio.h&gt;
int main()
{
int marks[10], i, n, sum = 0;
double average;
printf(&quot;Enter number of elements: &quot;);
scanf(&quot;%d&quot;, &amp;n);
for(i=0; i &lt; n; ++i)
{
printf(&quot;Enter number %d: &quot;,i+1);
scanf(&quot;%d&quot;, &amp;marks[i]);
// adding integers entered by the user to the sum variable
sum += marks[i];
}
// explicitly convert sum to double then calculate average
average = (double) sum / n;
printf(&quot;Average = %.2lf&quot;, average);
return 0;
}
```

```
//Prog. 3. C program to find the sum of two matrices of order 2*2

#include &lt;stdio.h&gt;

int main()

{

float a[2][2], b[2][2], result[2][2];

// Taking input using nested for loop

printf(&quot;Enter elements of 1st matrix\n&quot;);

for (int i = 0; i &lt; 2; ++i)

for (int j = 0; j &lt; 2; ++j)

{

printf(&quot;Enter a%d%d: &quot;, i + 1, j + 1);

scanf(&quot;%f&quot;, &amp;a[i][j]);

}

// Taking input using nested for loop

printf(&quot;Enter elements of 2nd matrix\n&quot;);

for (int i = 0; i &lt; 2; ++i)

for (int j = 0; j &lt; 2; ++j)

{

printf(&quot;Enter b%d%d: &quot;, i + 1, j + 1);

scanf(&quot;%f&quot;, &amp;b[i][j]);

}

// adding corresponding elements of two arrays

for (int i = 0; i &lt; 2; ++i)

for (int j = 0; j &lt; 2; ++j)

{

result[i][j] = a[i][j] + b[i][j];

}
```

```
// Displaying the sum
printf(&quot;\nSum Of Matrix:&quot;);
for (int i = 0; i &lt; 2; ++i)
```

```
for (int j = 0; j &lt; 2; ++j)
{
printf(&quot;%.1f\t&quot;, result[i][j]);
if (j == 1)
printf(&quot;\n&quot;);
}
return 0;
}

//Prog. 4. Print the minimum of the array and maximum of the array
values.
#include &lt;stdio.h&gt;
int main()
{
int a[1000],i,n,min,max;
printf(&quot;Enter size of the array : &quot;);
scanf(&quot;%d&quot;,&amp;n);
printf(&quot;Enter elements in array : &quot;);
for(i=0; i&lt;n; i++)
{
scanf(&quot;%d&quot;,&amp;a[i]);
}
```

```
min=max=a[0];

for(i=1; i&lt;n; i++)

{

if(min&gt;a[i])

min=a[i];

if(max&lt;a[i])

max=a[i];

}
```

```
printf(&quot; Minimum of array is : %d&quot;,min);

printf(&quot;\n Maximum of array is : %d&quot;,max);

return 0;

}
```

```
//Prog. 5. Program to Find the Transpose of a Matrix

#include &lt;stdio.h&gt;

int main()

{

int a[10][10], transpose[10][10], r, c;

printf(&quot;Enter rows and columns: &quot;);

scanf(&quot;%d %d&quot;, &amp;r, &amp;c);

// asssigning elements to the matrix

printf(&quot;\nEnter matrix elements:\n&quot;);

for (int i = 0; i &lt; r; ++i)

for (int j = 0; j &lt; c; ++j)

{
```

```
printf(&quot;Enter element a%d%d: &quot;, i + 1, j + 1);

scanf(&quot;%d&quot;, &amp;a[i][j]);

}

// printing the matrix a[][]

printf(&quot;\nEntered matrix: \n&quot;);

for (int i = 0; i &lt; r; ++i)

for (int j = 0; j &lt; c; ++j)

{

printf(&quot;%d &quot;, a[i][j]);

if (j == c - 1)

printf(&quot;\n&quot;);

}

// computing the transpose

for (int i = 0; i &lt; r; ++i)

for (int j = 0; j &lt; c; ++j)
```

```
{

transpose[j][i] = a[i][j];

}

// printing the transpose

printf(&quot;\nTranspose of the matrix:\n&quot;);

for (int i = 0; i &lt; c; ++i)

for (int j = 0; j &lt; r; ++j)

{ printf(&quot;%d &quot;, transpose[i][j]);

if (j == r - 1)

printf(&quot;\n&quot;);
}
return 0;
}
```

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 &lt;stdio.h&gt;
void printArray(int arr[])
```

```
{
printf(&quot;Size of Array in Functions: %d\n&quot;, sizeof(arr));
printf(&quot;Array Elements: &quot;);
for (int i = 0; i &lt; 5; i++)
{
printf(&quot;%d &quot;,arr[i]);
}
}
// driver code
int main()
{
int arr[5] = { 10, 20, 30, 40, 50 };
printf(&quot;Size of Array in main(): %d\n&quot;, 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
```

Return 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 &lt;stdio.h&gt;
// function
int* func()
{
static int arr[5] = { 1, 2, 3, 4, 5 };
```

```
return arr;
}
// driver code
int main()
{
int* ptr = func();
```

```
printf(&quot;Array Elements: &quot;);
for (int i = 0; i &lt; 5; i++) {
printf(&quot;%d &quot;, *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 a segmentation fault while trying to access the array in the main function.

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. The following are the main properties of an array in C:

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 &lt;stdio.h&gt;
```

```
int main()
{
```

```
int arr[2];
printf(&quot;%d &quot;, arr[3]);
printf(&quot;%d &quot;, arr[-2]);
return 0;
}
```

Output

```
211343841 4195777
```

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 &lt;stdio.h&gt;
int main()
{
// Array declaration by initializing it
// with more elements than specified size.
int arr[2] = { 10, 20, 30, 40, 50 };
return 0;
}
```

Warnings:

```
prog.c: In function &#39;main&#39;:
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 &#39;arr&#39;)
```

```
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 &#39;arr&#39;)
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 &#39;arr&#39;)
```

Examples of Array in C

Example 1: C Program to perform array input and output.

In this program, we will use scanf() and print() function to take input and print output for the array.

```
// C Program to perform input and output on array
#include &lt;stdio.h&gt;
int main()
{
// declaring an integer array
int arr[5];
// taking input to array elements one by one
for (int i = 0; i &lt; 5; i++) {
scanf(&quot;%d&quot;, &amp;arr[i]);
}
// printing array elements
printf(&quot;Array Elements: &quot;);
for (int i = 0; i &lt; 5; i++) {
printf(&quot;%d &quot;, arr[i]);
}
return 0;
}
```

Input

```
5 7 9 1 4
```

Output

```
Array Elements: 5 7 9 1 4
```

Example 2: 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 &lt;stdio.h&gt;
// 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 &lt; 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 };

108

// size of array using sizeof operator
int n = sizeof(arr) / sizeof(float);
// printing array elements
printf(&quot;Array Elements: &quot;);
for (int i = 0; i &lt; n; i++) {
printf(&quot;%.0f &quot;, arr[i]);
}
// calling getAverage function and printing average
printf(&quot;\nAverage: %.2f&quot;, getAverage(arr, n));
return 0;
}
```

Output

```
Array Elements: 10 20 30 40 50
Average: 30.00
```

Example 3: C Program to find the largest number in the array.

```
// C Program to find the largest number in the array.
#include &lt;stdio.h&gt;
// function to return max value
int getMax(int* arr, int size)
{
int max = arr[0];
for (int i = 1; i &lt; size; i++) {
if (max &lt; arr[i]) {
```

```
max = arr[i];
}
}
return max;
}
int main()
{
int arr[10]
```

```
= { 135, 165, 1, 16, 511, 65, 654, 654, 169, 4 };
printf(&quot;Largest Number in the Array: %d&quot;,
getMax(arr, 10));
return 0;
}
```

Output

```
Largest Number in the Array: 654
```

Advantages of Array in C

The following are the main advantages of an array:

Random and fast access of elements using the array index.

Use of fewer lines of code as it creates a single array of multiple elements.

Traversal through the array becomes easy using a single loop.

Sorting becomes easy as it can be accomplished by writing fewer lines of code.

Disadvantages of Array in C

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.

Insertion and deletion of elements can be costly since the elements are needed to be rearranged after insertion and deletion.

## Conclusion

The array is one of the most used and important data structures in C. It is one of the core concepts of C language that is used in every other program. Though it is important to know about its limitation so that we can take advantage of its functionality.