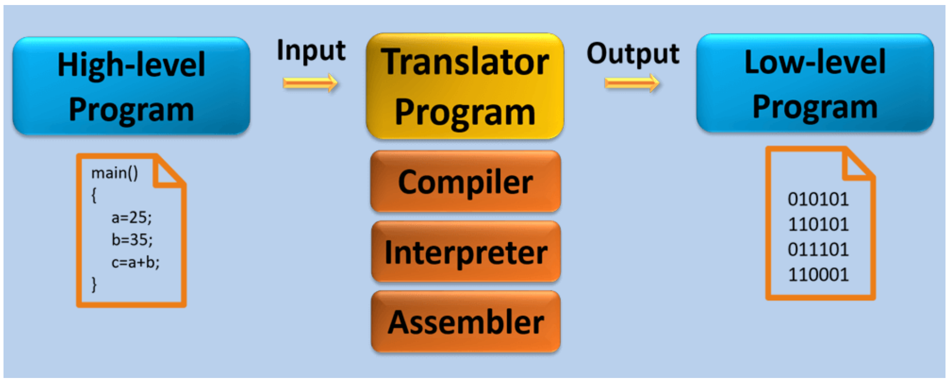
C PROGRAMMING LANGUAGE (INTRODUCTION)

Notes made by Soikot Shahriar
[t.me/soikot_shahriaar]
[github.com/soikot-shahriaar]

Introduction to Programming

Translator Programs

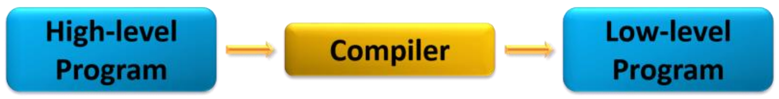


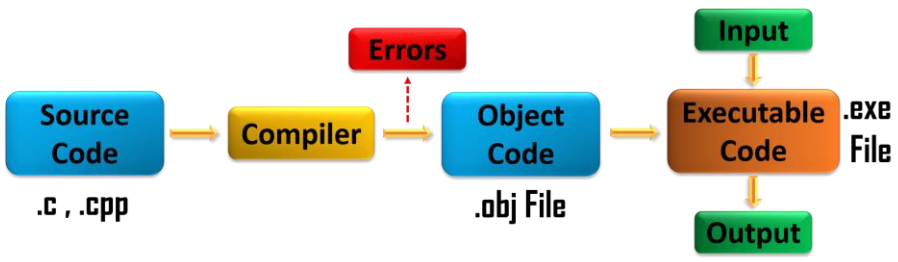
N.B: a program written in high-level language is called source program and a program written in machine language is called object program.

Compiler

A compiler is a type of translator program that Scans the entire program written in a high-level language and translates it as a whole into machine code.

Programming language like C, C++, Objective-C, C#, Pascal, COBOL, ADA, Visual Basic, Smalltalk, Scheme use compilers.





The compiler completes the task of translation in two steps –

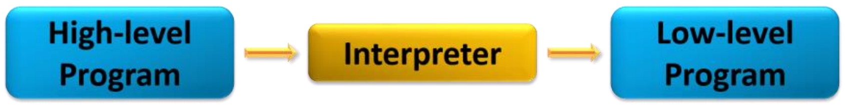
In the first step, the compiler reads the source program and converts into the object program.

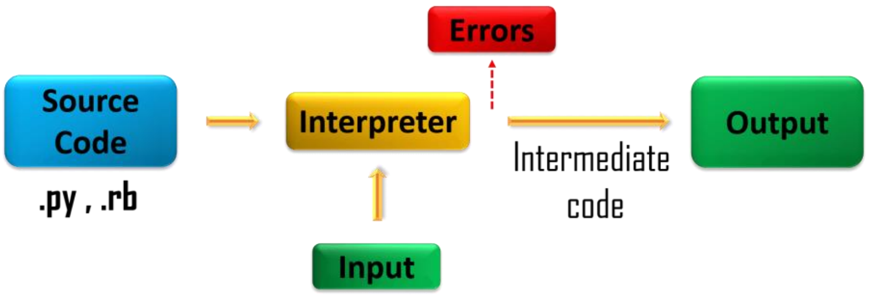
The second step is to execute the object program to display the results on the basis of input data.

Interpreter

An interpreter, like a compiler, is a kind of translator program that reads one line of a program written in a high-level language and converts it into mechanical language and executes it, displaying instant results.

Programming language like BASIC, Python, php, Perl, Ruby, JavaScript use interpreters.



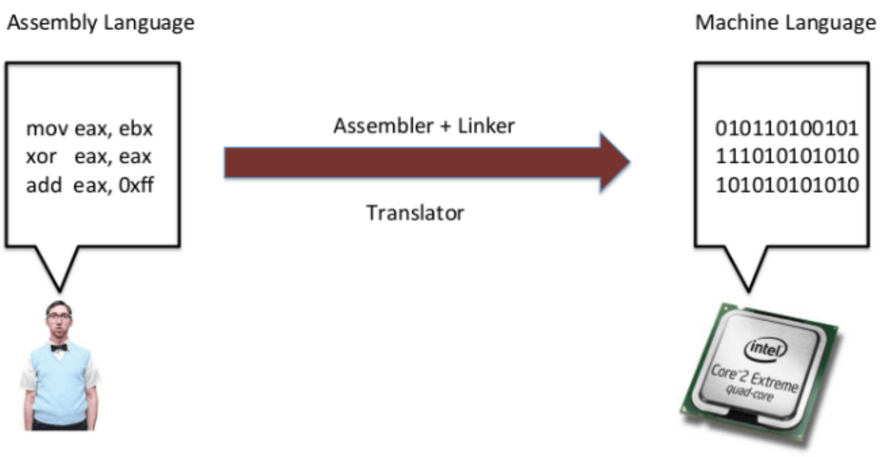


Interpreter translates line by line and reports the error once it encountered during the translation process.

Assembler

Assembler is a translator program which is used to translate the assembly language code into machine language code.

- Convert mnemonic codes into equivalent machine code.
- Convert symbolic operands or address to machine addresses.



C Programming

Follow this link for all the codes:

https://github.com/soikot-shahriaar/C-Programming

Program Structure:

Section	**Code**

Documentation	`/*`
	`file: sum.c`
	`author: Soikot Shahriar`
	`description: program to find sum of an integer and a constant value.`
	`*/`

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;`
	`}`

Subprograms	`int sum(int y) {`
	`return y + X;`
	`}`

Output	`Sum: 75`

Variables

In programming, a variable is a container (storage area) to hold data. To indicate the sorage area, each variable should be given a unique name (identifier).

```
int playerScore = 95;
```

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

```
Char ch = 'a';
// some code
ch = 'b';
```

The value of a variable can be changed, hence the name variable. But the variable type cannot be changed once it is declared.

Data Types

In C programing, data types are declarations for variables. This determines the type and size of data associated with variables.

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.

```
int id, age;
```

The size of `int` is usually 4 bytes (32 bits). It can take 2^{32} 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.

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.

```
char test = 'h';
```

The size of the character variable is 1 byte.

void

`void` is an incomplete type. It means "nothing" or "no type".

If a function is not returning anything, its return type should be `void`. We cannot create variables of `void` type.

short and long

To use a large number, type specifier `long` is used.

```
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.

`short` can be used only for a small integer (`[−32,767, +32,767]` range)

```
short d;
```

signed and unsigned

In C, `signed` and `unsigned` are type modifiers.

`signed` - allows for storage of both positive and negative numbers.

`unsigned` - allows for storage of only positive numbers.

Operators

Arithmetic Operators

An arithmetic operator performs mathematical operations.

Operator	Meaning of Operator
+	addition or unary plus
-	subtraction or unary minus
*	multiplication
/	division
%	remainder after division (modulo division)

```
int a = 9, b = 4, c;
    c = a+b;
printf("a+b = %d \n",c);
    c = a-b;
printf("a-b = %d \n",c);
    c = a*b;
printf("a*b = %d \n",c);
    c = a/b;
printf("a/b = %d \n",c);
    c = a%b;
    // Remainder when a divided by b
printf("Remainder = %d \n",c);
```

```
a+b = 13

a-b = 5

a*b = 36

a/b = 2


Remainder = 1
```

```
int num1 = 5;
int num2 = 2;
float sum = (float) num1 / num2;

printf("%.1f", sum);
```

```
2.5
```

Increment and Decrement Operators

Increment ++ increases the value by 1 whereas decrement −− decreases the value by 1. These two operators are unary operators, meaning they only operate on a single operand.

Increament Operators (++)

Post Increament	Pre Increament
Operand Value is assigned first then gets increamented.	Operand Value gets increamented first and then assigned.

`int a = 20;`
 	
`a ++;`
 	
`printf("%d" , a);`
	`int a = 20;`
	
	`++a;`
	
	`printf("%d" , a);`

21	21

`int a = 20, b;`
 	
`b = a ++;`
 	
`printf("%d" , a);`
 	
`printf("%d" , b);`
	`int a = 20, b;`
	
	`b = ++a;`
	
	`printf("%d" , a);`
	
	`printf("%d" , b);`

21 20	21 21

Decreament Operators (--)

Post Decreament	**Pre Decreament**
Operand Value is assigned first then gets decreamented.	Operand Value gets decreamented first and then assigned.

`int a = 20;`
 	
`a --;`
 	
`printf("%d" , a);`
	`int a = 20;`
	
	`--a;`
	
	`printf("%d" , a);`

19	19

`int a = 20, b;`
 	
`b = a --;`
 	
`printf("%d" , a);`
 	
`printf("%d" , b);`
	`int a = 20, b;`
	
	`b = --a;`
	
	`printf("%d" , a);`
	
	`printf("%d" , b);`

19 20	19 19

Assignment Operators

Assignment operators are used for assigning a value to a variable.

Operator	**Example**	**Same as**
=	a = b	a = b
+=	a += b	a = a+b
-=	a -= b	a = a-b

*=	a *= b	a = a*b
/=	a /= b	a = a/b
%=	a %= b	a = a%b

```
int a = 5, c;
c = a;
printf("c = %d\n", c);
c += a;
printf("c = %d\n", c);
c -= a;
printf("c = %d\n", c);
c *= a;
printf("c = %d\n", c);
c /= a;
printf("c = %d\n", c);
c %= a;
printf("c = %d\n", c);
```

```
c = 5
c = 10
c = 5
c = 25
c = 5
c = 0
```

Relational Operators

Relational operators check the relationship between two operands. If the relation is true, it returns 1; if the relation is false, it returns value 0.

Operator	Meaning of Operator	Example
==	Equal to	5 == 3 is evaluated to 0
>	Greater than	5 > 3 is evaluated to 1
<	Less than	5 < 3 is evaluated to 0
!=	Not equal to	5 != 3 is evaluated to 1
>=	Greater than or equal to	5 >= 3 is evaluated to 1
<=	Less than or equal to	5 <= 3 is evaluated to 0

```
int a = 5, b = 5, c = 10;
printf("%d == %d is %d \n", a, b, a == b);
printf("%d == %d is %d \n", a, c, a == c);
printf("%d > %d is %d \n", a, b, a > b);
printf("%d > %d is %d \n", a, c, a > c);
printf("%d < %d is %d \n", a, b, a < b);
printf("%d < %d is %d \n", a, c, a < c);
printf("%d != %d is %d \n", a, b, a != b);
printf("%d != %d is %d \n", a, c, a != c);
printf("%d >= %d is %d \n", a, b, a >= b);
printf("%d >= %d is %d \n", a, c, a >= c);
printf("%d <= %d is %d \n", a, b, a <= b);
printf("%d <= %d is %d \n", a, c, a <= c);
```

```
5 == 5 is 1
5 == 10 is 0
5 > 5 is 0
5 > 10 is 0
5 < 5 is 0
5 < 10 is 1
5 != 5 is 0
5 != 10 is 1
5 >= 5 is 1
5 >= 10 is 0
5 <= 5 is 1
5 <= 10 is 1
```

Logical Operators

An expression containing logical operator returns either 0 or 1 depending upon whether expression results true or false.

Operator	Meaning	Example
&&	Logical AND. True only if all operands are true	If c = 5 and d = 2 then, expression `((c==5) && (d>5))` equals to 0.
||	Logical OR. True only if either one operand is true	If c = 5 and d = 2 then, expression `((c==5) || (d>5))` equals to 1.
!	Logical NOT. True only if the operand is 0	If c = 5 then, expression `!(c==5)` equals to 0.

`int a = 5, b = 5, c = 10;`	

`result = (a == b) && (c > b);`
`printf("(a == b) && (c < b) is %d \n", result);`
	(a == b) && (c > b) is 1

`result = (a == b) && (c < b);`
`printf("(a == b) && (c < b) is %d \n", result);`
	(a == b) && (c < b) is 0

`result = (a == b) || (c < b);`
`printf("(a == b) || (c < b) is %d \n", result);`
	(a == b) || (c < b) is 1

`result = (a != b) || (c < b);`
`printf("(a != b) || (c < b) is %d \n", result);`
	(a != b) || (c < b) is 0

`result = !(a != b);`
`printf("!(a != b) is %d \n", result);`
	!(a != b) is 1

`result = !(a == b);`
`printf("!(a == b) is %d \n", result);`
	!(a == b) is 0

Bitwise Operators

Operators	Meaning of operators
&	Bitwise AND
|	Bitwise OR
^	Bitwise Exclusive OR
~	Bitwise Complement
>>	Shift Right
<<	Shift Left

```c
#include <stdio.h>
int main() {
    int a = 12, b = 25;
    printf("Output = %d", a & b);
    return 0;
}
// Output = 8
```

```
12 = 00001100 (In Binary)
25 = 00011001 (In Binary)
Bit Operation of 12 and 25:
   00001100
& 00011001
---------------
   00001000  = 8 (In decimal)
```

```c
#include <stdio.h>
int main() {
    int a = 12, b = 25;
    printf("Output = %d", a | b);
    return 0;
}
// Output = 29
```

```
12 = 00001100 (In Binary)
25 = 00011001 (In Binary)
Bit Operation of 12 and 25:
   00001100
|  00011001
---------------
   00011101  = 29 (In decimal)
```

```c
#include <stdio.h>
int main() {
    int a = 12, b = 25;
    printf("Output = %d", a ^ b);
    return 0;
}
// Output = 21
```

```
12 = 00001100 (In Binary)
25 = 00011001 (In Binary)
Bit Operation of 12 and 25:
   00001100
^ 00011001
---------------
   00010101  = 21 (In decimal)
```

```c
#include <stdio.h>

int main() {

 printf("Output = %d\n", ~35);

return 0;
}
```

```
35 = 00100011 (In Binary)
Bitwise complement Operation of 35:
~ 00100011
---------------
  11011100  = 220 (In decimal)
```

Decimal	**Binary**	**2's complement**
220	11011100	-(00100011+1) = -00100100 = -36(decimal)

Shift Operators

```c
#include <stdio.h>

int main() {

int num=212, i;

for (i = 0; i <= 2; ++i) {
 printf("Right shift by %d: %d\n", i, num >> i);
}

printf("\n");

for (i = 0; i <= 2; ++i) {
 printf("Left shift by %d: %d\n", i, num << i);
}

return 0;
}
```

Output:

```
Right Shift by 0: 212
Right Shift by 1: 106
Right Shift by 2: 53

Left Shift by 0: 212
Left Shift by 1: 424
Left Shift by 2: 848
```

Explanation:
212 = 11010100 (In binary)
212 >> 1 = 01101010 (106)
[Right shift by one bit]
212 >> 2 = 00110101 (53)
[Right shift by two bits]

212<<1 = 110101000 (424)
[Left shift by one bit]
212<<2 = 1101010000 (848)
[Left shift by two bits]

Conditional Operator / Ternary Operator

Operator	Syntax
? :	condition ? expression1: expression2

```c
int a,b,g;
printf("Enter the values:\n");
scanf("%d %d",&a,&b);
g = a>b?a:b;
printf("Greatest = %d", g);
```

```
Enter the values:
10
30
Greatest = 30
```

```c
#include <stdio.h>
int main() {
int M, N, O, highest;
scanf("%d %d %d", &N, &M, &O);
highest = (M>N && M>O) ? (M) : (
(N>O)?(N):(O) );
printf("%d\n", highest);

if (highest >= 40) {
printf("Pass\n");
} else {
printf("Fail\n");
}
return 0;
}
```

```
18 80 42
80
Pass
```

Comma Operator

Comma operators are used to link related expressions together.

```c
int a,b,c;

scanf("%d%d%d",&a,&b,&c);

printf("%d%d%d",a,b,c);
```

Sizeof Operator

```c
char m;
int a;
float b;
double c;
printf("Size of char is %d\n",sizeof(m));
printf("Size of int is %d\n",sizeof(a));
printf("Size of float is %d\n",sizeof(b));
printf("Size of double is %d\n",sizeof(c));
```

```
Size of char is 1
Size of int is 4
Size of float is 4
Size of double is 8
```

Precedence and Associativity of Operators

Precedence determines **the order** in which different operators in a complex expressions are evaluated. Operators are evaluated from **Highest precedence to Lowest precedence**.

Associativity determines how **operators with same precedence** are evaluated in an expression.

Left Associativity	**Right Associativity**
Operators evaluated form Left to Right	Operators evaluated from Right to Left
Denoted by L→R	Denoted by R→L
Example: Arithmetic Operators	Example: Unary, Assignment Operators

OPERATOR	**DESCRIPTION**	**ASSOCIATIVITY**
( )	parentheses (function call)	L→R
[ ]	brackets (array subscript)	
.	member selection via object name	
->	member selection via pointer	
++ --	postfix increment, postfix decrement	
++ --	prefix increment, prefix decrement	**R→L**
+ -	unary plus, unary minus	
! ~	logical NOT/bitwise Negate	
(type)	cast	
*	dereference	
&	address (of operand)	
sizeof	determine size in bytes	
* / %	multiplication, division, modulus	L→R
+ –	plus, minus	L→R
<<>>	bitwise shift left, bitwise shift right	L→R
<<=	relational less than, less than or equal to	L→R
>>=	relational greater than, greater than or equal to	
== !=	relational is equal to, is not equal to	L→R
&	bitwise AND	L→R
^	bitwise exclusive OR	L→R
|	bitwise OR	L→R
&&	logical AND	L→R
||	logical OR	L→R
? :	conditional	**R→L**
=	assignment	**R→L**
+= -=	addition assignment, subtraction assignment	
*= /=	multiplication assignment, division assignment,	

%= &= ^= |= <<= >>=	modulus assignment, bitwise AND assignment bitwise exclusive OR assignment, OR assignment bitwise shift left assignment, bitwise shift right assignment	
,	comma (separate expressions)	L→R

Category	**Operators**	**Associativity**
Postfix	() [] -> . ++ —	L→R
Unary	– + ! ~ – – ++ (type)* &sizeof	**R→L**
Multiplicative	/ * %	L→R
Additive	– +	L→R
Shift	>><<	L→R
Relational	<>>= <=	L→R
Equality	!= ==	L→R
Bitwise	& ^ |	L→R
Logical	&& ||	L→R
Conditional	?:	**R→L**
Assignment	= -= += /= *= %=>>= &= <<= |= ^=	**R→L**
Comma	,	L→R

Comments

There are two ways to add comments in C:

1. // - Single Line Comment
2. /*...*/ - Multi-line Comment

Input Output (I/O)

Output

In C programming, `printf()` is one of the main output function.

```
#include<stdio.h>
int main(){
// Displays the string inside quotations
printf("C Programming");
return 0; }
```

```
C Programming
```

- 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 <stdio.h>` statement.
- The return 0; statement inside the `main()` function is the "Exit Status" of the program. It's optional.

Integer Output

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

```c
#include<stdio.h>
int main()
{
int testInteger = 5;
printf("Number = %d", testInteger);
return 0;
}
```

```
Number = 5
```

Float and Double Output

`%f` format specifier is used to print `float`

`%lf` format specifier is used to print `double`

```c
#include<stdio.h>
int main(){
float number1 = 13.5;
double number2 = 12.4;
printf("number1 = %f\n", number1);
printf("number2 = %lf", number2);
return 0;
}
```

```
number1 = 13.500000
number2 = 12.400000
```

Print Characters

`%c` format specifier is used to print `char`

```c
#include<stdio.h>
int main()
{
char chr = 'a';
printf("character = %c", chr);
return 0;
}
```

```
character = a
```

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.

Integer Input/Output

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

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

Here, `%d` format specifier is used 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 `&testInteger` inside `scanf()`. It is because `&testInteger` gets the address of `testInteger` and the value entered by the user is stored in that address.

Float and Double Input/Output

`%f` and `%lf` format specifier is used for `float` and `double` respectively.

```
#include <stdio.h>
int main()
{
float num1;
double num2;

printf("Enter a number: ");
scanf("%f", &num1);
printf("Enter another number: ");
scanf("%lf", &num2);

printf("num1 = %f\n", num1);
printf("num2 = %lf", num2);

return 0;
}
```

```
Enter a number: 12.523
Enter another number: 10.2
num1 = 12.523000
num2 = 10.200000
```

Character Input/Output

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

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

N.B: 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's ASCII value is printed.

ASCII Value

```c
#include <stdio.h>
int main(){
char chr;
printf("Enter a character: ");
scanf("%c", &chr);

// When %c is used, a character is displayed
printf("You entered %c.\n",chr);

// When %d is used, ASCII value is displayed
printf("ASCII value is %d.", chr);
return 0;
}
```

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

I/O Multiple Values

```c
#include <stdio.h>
int main(){
int a;
float b;

printf("Enter a integer and a float:\n");

// Taking multiple inputs
scanf("%d%f", &a, &b);

printf("You entered %d and %f", a, b);
return 0;
}
```

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

Conditionals

1. if Statement

The syntax of the if statement in C programming is:

```
if (test expression)
{
   // code
}
```

2. if...else Statement

The if statement may have an optional else block. The syntax of the if..else statement is:

```
if (test expression) {
    // run code if test expression is true
}
else {
    // run code if test expression is false
}
```

3. if...else Ladder

The if...else statement executes two different codes depending upon whether the test expression is true or false. Sometimes, a choice has to be made from more than 2 possibilities.

The if...else ladder allows us to check between multiple test expressions and execute different statements.

Syntax of if...else Ladder is:

```
if (test expression1) {
   // statement(s)
}
else if(test expression2) {
   // statement(s)
}
else if (test expression3) {
   // statement(s)
}
.
.
else {
   // statement(s)
}
```

4. Nested if...else

It is possible to include an if...else statement inside the body of another if...else statement. Example:

```
#include <stdio.h>
int main() {
   int number1, number2;
   printf("Enter two integers: ");
   scanf("%d %d", &number1, &number2);

   if (number1 >= number2) {
     if (number1 == number2) {
       printf("Result: %d = %d",number1,number2);
     }
     else {
      printf("Result: %d > %d", number1, number2);
     }
   }
   else {
       printf("Result: %d < %d",number1, number2);
   }

   return 0;
}
```

Loops

In programming, a loop is used to repeat a block of code until the specified condition is met.C programming has three types of loops:

- for loop
- while loop
- do...while loop

1. for Loop

The syntax of the for loop is:

```
for (initializationStatement; testExpression; updateStatement)
{
    // statements inside the body of loop
}
```

How for loop works?

a. The initialization statement is executed only once.

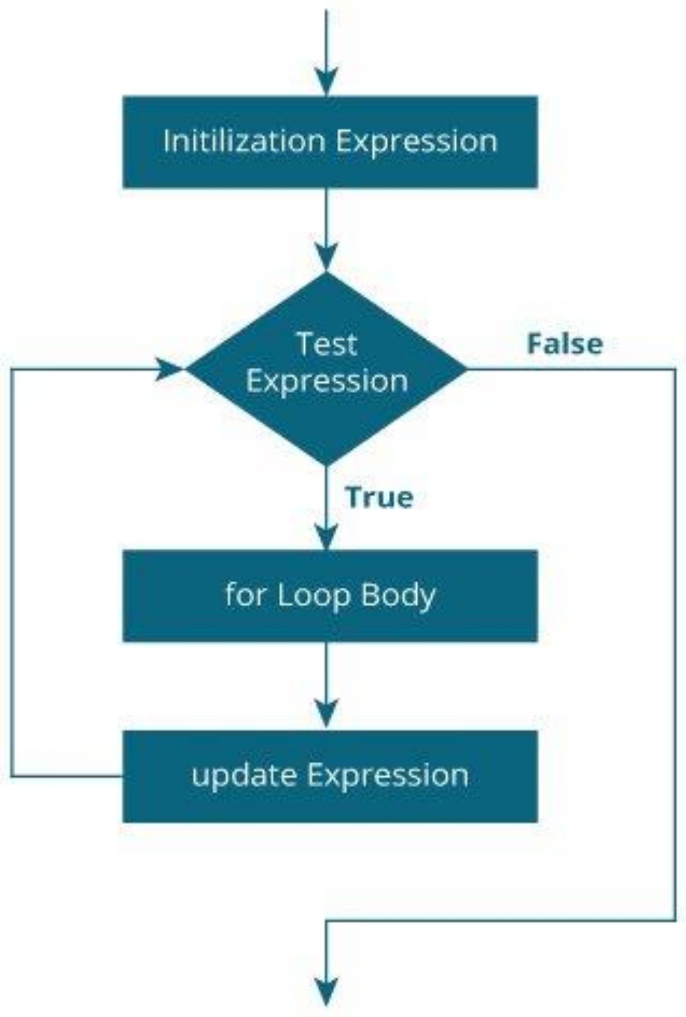
b. Then, the test expression is evaluated. If the test expression is evaluated to false, the for loop is terminated.

c. However, if the test expression is evaluated to true, statements inside the body of the for loop are executed, and the update expression is updated.

d. Again, the test expression is evaluated.

e. This process goes on until the test expression is false. When the test expression is false, the loop terminates.

Flowchart of for loop:



2. While Loop

The syntax of the while loop is:

```
while (testExpression) {
     // the body of the loop
}
```

How while loop works?

a. The while loop evaluates the testExpression inside the parentheses ().

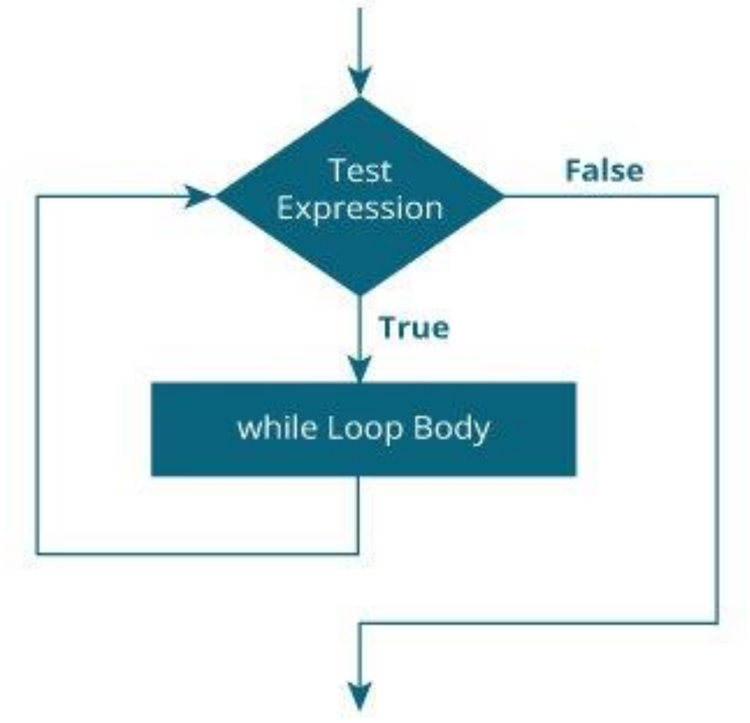
b. If testExpression is true, statements inside the body of while loop are executed. Then, testExpression is evaluated again.

c. The process goes on until testExpression is evaluated to false.

d. If testExpression is false, the loop terminates (ends).

e. To learn more about test expressions (when testExpression is evaluated to true and false), check out relational and logical operators.

Flowchart of while loop:



3. do...while loop

The do..while loop is similar to the while loop with one important difference. The body of do...while loop is executed at least once. Only then, the test expression is evaluated.

The syntax of the do...while loop is:

```
do {
  // the body of the loop
}
while (testExpression);
```

How do...while loop works?

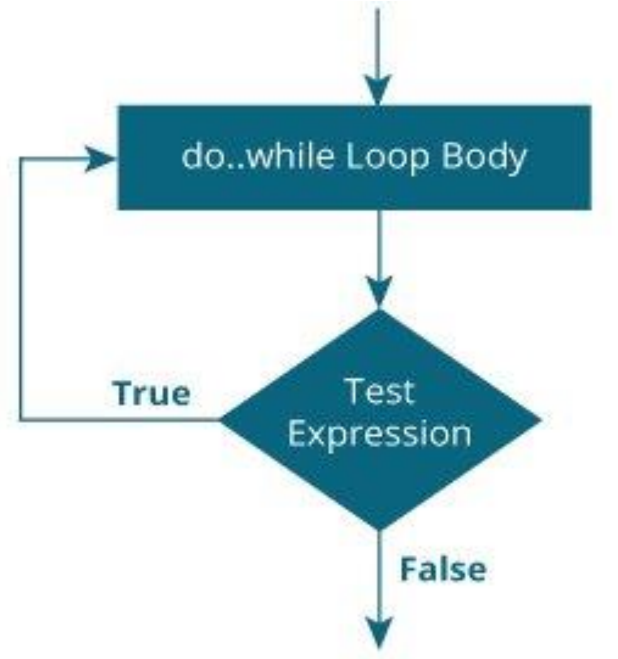
a. The body of do...while loop is executed once. Only then, the testExpression is evaluated.

b. If testExpression is true, the body of the loop is executed again and testExpression is evaluated once more.

c. This process goes on until testExpression becomes false.

d. If testExpression is false, the loop ends.

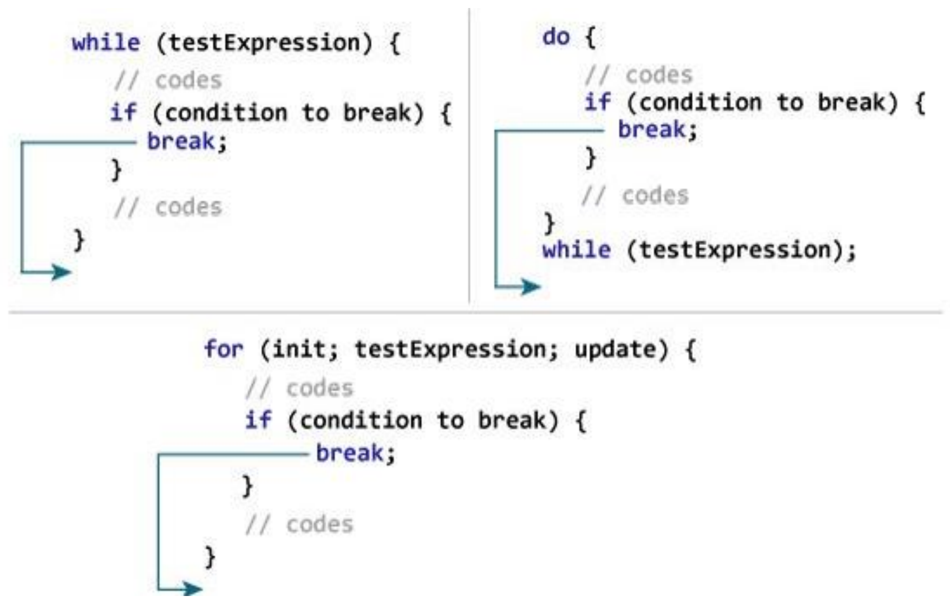
Flowchart of do...while Loop:



Break and Continue

break

The break statement ends the loop immediately when it is encountered. Its syntax is: `break;` This statement is almost always used with if...else statement inside the loop. **How break statement works?**



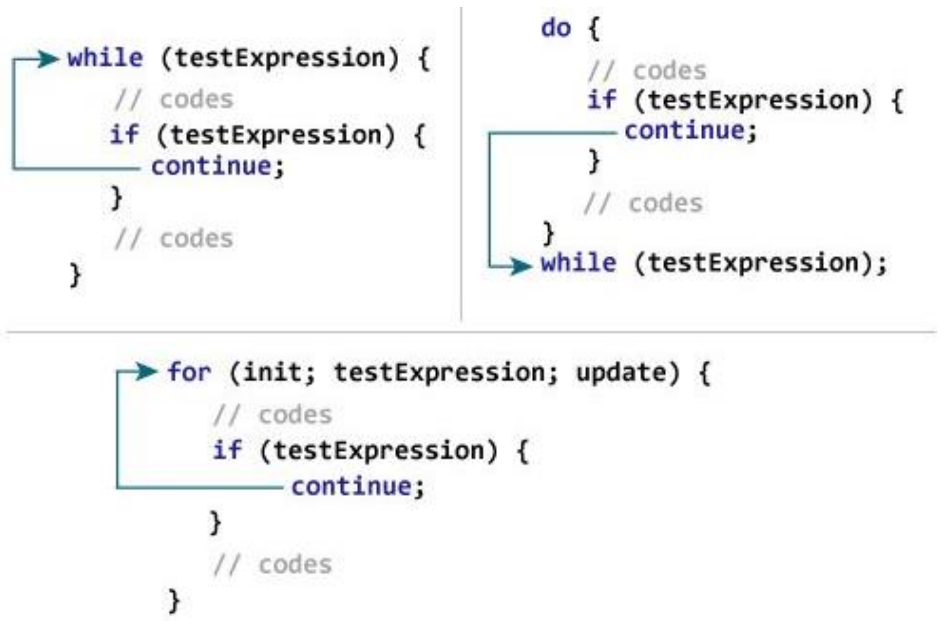
```
while (testExpression) {
    // codes
    if (condition to break) {
        break;
    }
    // codes
}
```

```
do {
    // codes
    if (condition to break) {
        break;
    }
    // codes
}
while (testExpression);
```

```
for (init; testExpression; update) {
    // codes
    if (condition to break) {
        break;
    }
    // codes
}
```

continue

The continue statement skips the current iteration of the loop and continues with the next iteration. Its syntax is: `continue;` The continue statement is almost always used with the if...else statement. **How continue statement works?**



```
while (testExpression) {
    // codes
    if (testExpression) {
        continue;
    }
    // codes
}
```

```
do {
    // codes
    if (testExpression) {
        continue;
    }
    // codes
}
while (testExpression);
```

```
for (init; testExpression; update) {
    // codes
    if (testExpression) {
        continue;
    }
    // codes
}
```

Switch Case

The switch statement allows us to execute one code block among many alternatives. We can do the same thing with the if...else..if ladder. However, the syntax of the switch statement is much easier to read and write.

Syntax of switch...case:

```
switch (expression)
{
    case constant1:
      // statements
      break;

    case constant2:
      // statements
      break;
    .
    .
    default:
      // default statements
}
```

How does the switch statement work?

a. The expression is evaluated once and compared with the values of each case label.

b. If there is a match, the corresponding statements after the matching label are executed. For example, if the value of the expression is equal to constant2, statements after case constant2: are executed until break is encountered.

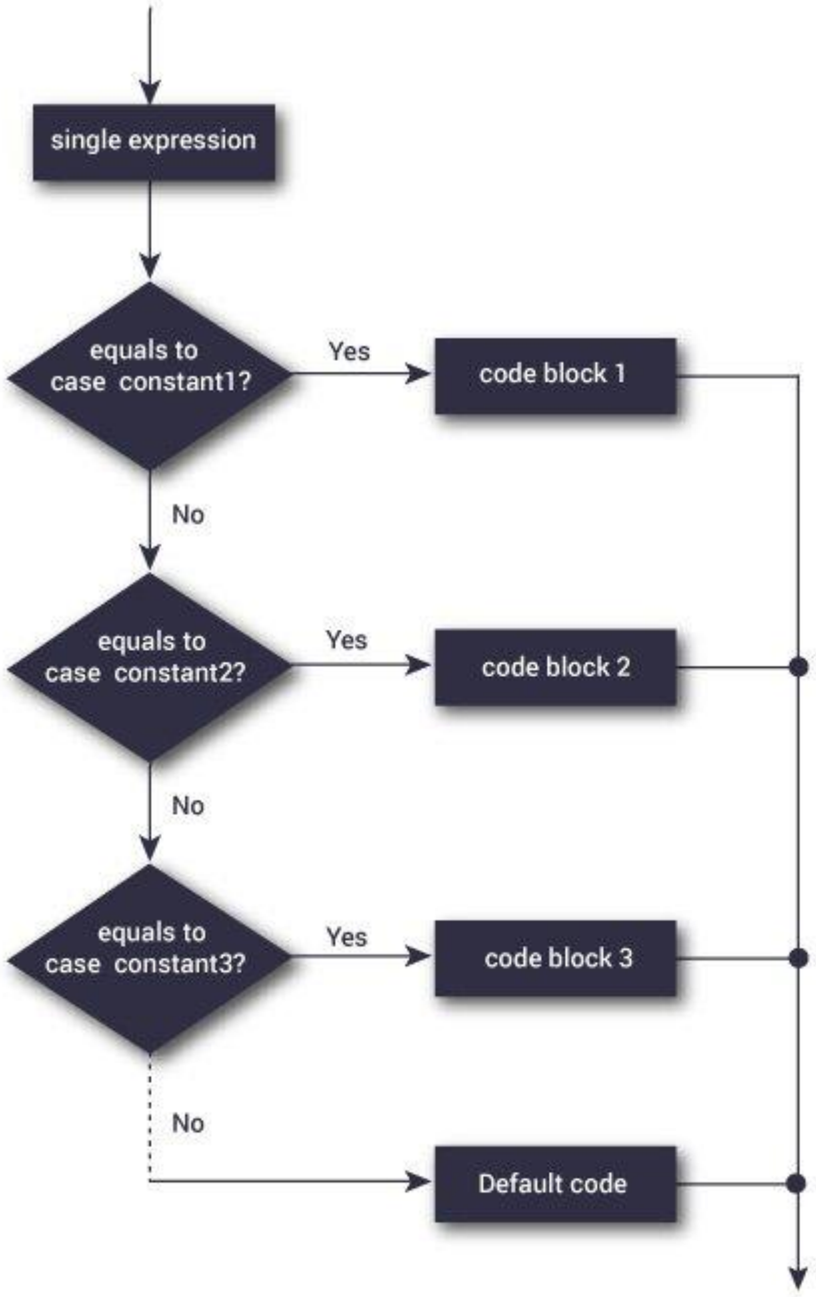
c. If there is no match, the default statements are executed.

Notes:

a. If we do not use the break statement, all statements after the matching label are also executed.

b. The default clause inside the switch statement is optional.

Flowchart of switch Statement:



Functions

A function is a block of code that performs a specific task. Dividing a complex problem into smaller chunks makes our program easy to understand and reuse.

There are two types of function in C programming:

- Standard library functions
- User-defined functions

Standard library functions:

The standard library functions are built-in functions in C programming. These functions are defined in header files. For example,

- The printf() is a standard library function to send formatted output to the screen (display output on the screen). This function is defined in the stdio.h header file.
- Hence, to use the printf()function, we need to include the stdio.h header file using #include <stdio.h>.
- The sqrt() function calculates the square root of a number. The function is defined in the math.h header file.

User-defined functions:

We can also create functions as per our need. Such functions created by the user are known as user-defined functions.

How user-defined function works?

```
#include <stdio.h>
void functionName()
{
    ... .. ...
    ... .. ...
}

int main()
{
    ... .. ...
    ... .. ...
    functionName();
    ... .. ...
    ... .. ...
}
```

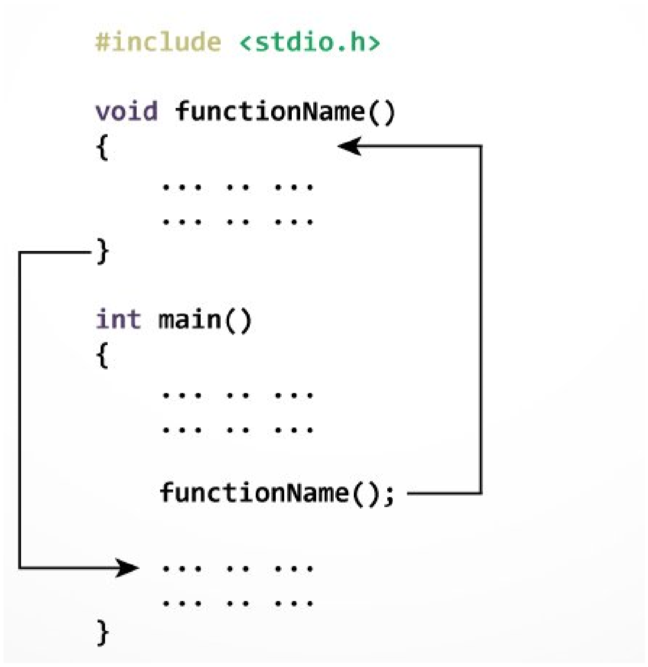
a. The execution of a C program begins from the main() function.

b. When the compiler encounters functionName();, control of the program jumps to void functionName()

c. And, the compiler starts executing the codes inside functionName().

d. The control of the program jumps back to the main() function once code inside the function definition is executed.

How function works in C programming?



```
#include <stdio.h>

void functionName()
{
    ... .. ...
    ... .. ...
}

int main()
{
    ... .. ...
    ... .. ...

    functionName();

    ... .. ...
    ... .. ...
}
```

Function prototype:

A function prototype is simply the declaration of a function that specifies function's name, parameters and return type. It doesn't contain function body. A function prototype gives information to the compiler that the function may later be used in the program.

Syntax of function prototype:

```
returnType functionName(type1 argument1, type2 argument2, ...);
```

In the above example, `int addNumbers(int a, int b);` is the function prototype which provides the following information to the compiler:

a. name of the function is addNumbers()

b. return type of the function is int

c. two arguments of type int are passed to the function

* The function prototype is not needed if the user-defined function is defined before the main() function.

Calling a function:

Control of the program is transferred to the user-defined function by calling it.

Syntax of function call:

```
functionName(argument1, argument2, ...);
```

In the above example, the function call is made using `addNumbers(n1, n2);` statement inside the main() function.

Function definition:

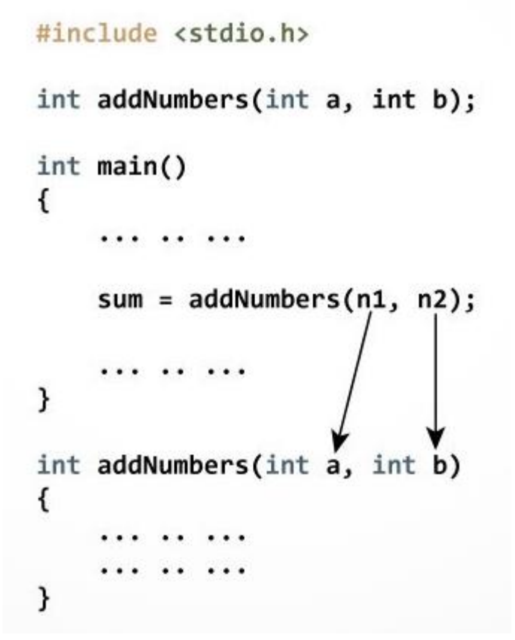
Function definition contains the block of code to perform a specific task. In our example, adding two numbers and returning it.

Syntax of function definition:

```
returnType functionName(type1 argument1, type2 argument2, ...)
{
    //body of the function
}
```

When a function is called, the control of the program is transferred to the function definition. And, the compiler starts executing the codes inside the body of a function.

Passing arguments to a function:



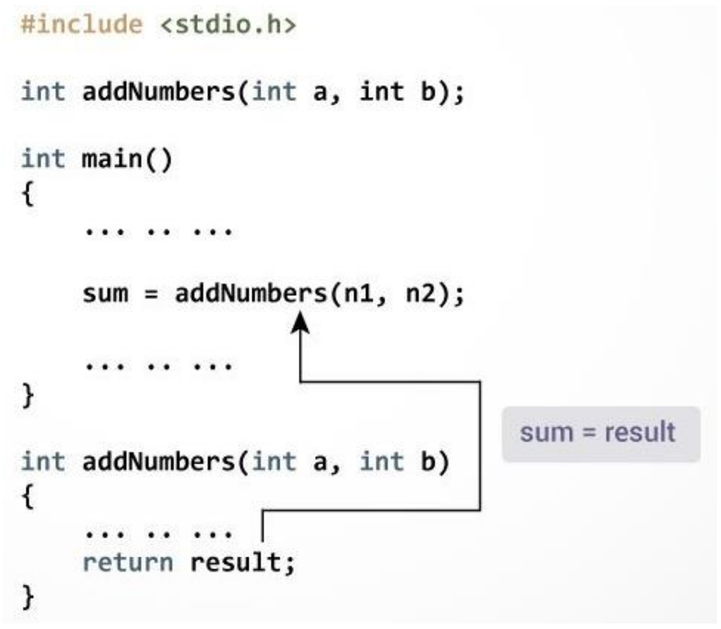
```
#include <stdio.h>

int addNumbers(int a, int b);

int main()
{
    ... .. ...

    sum = addNumbers(n1, n2);

    ... .. ...
}

int addNumbers(int a, int b)
{
    ... .. ...
    ... .. ...
}
```

- 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 accepts the passed arguments in the function definition. These arguments are called formal parameters of the 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.

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.



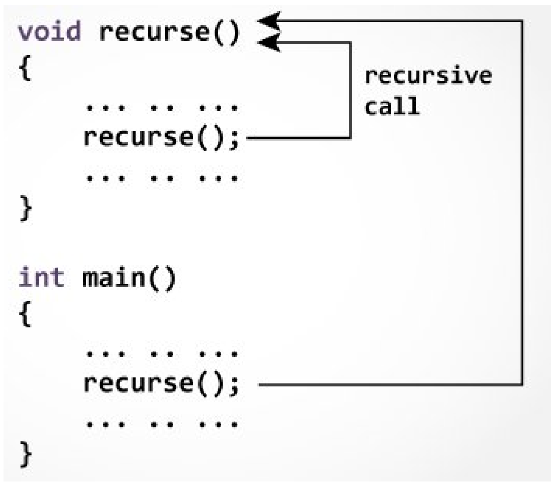
```
#include <stdio.h>

int addNumbers(int a, int b);

int main()
{
    ... .. ...

    sum = addNumbers(n1, n2);

    ... .. ...
}

int addNumbers(int a, int b)
{
    ... .. ...
    return result;
}
```

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

Recursion

A function that calls itself is known as a recursive function. And, this technique is known as recursion.

How recursion works?



```
void recurse()
{
    ... .. ...
    recurse();
    ... .. ...
}

int main()
{
    ... .. ...
    recurse();
    ... .. ...
}
```

The recursion continues until some condition is met to prevent it. To prevent infinite recursion, if...else statement (or similar approach) can be used where one branch makes the recursive call, and other doesn't.

Example: Sum of Natural Numbers Using Recursion:

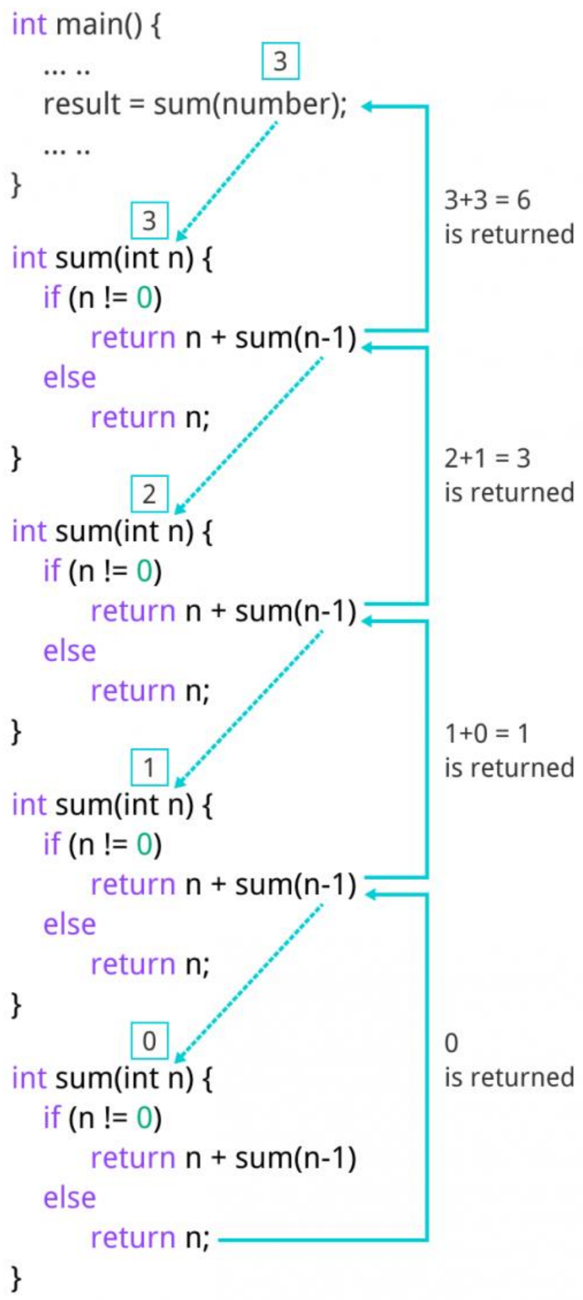
```
#include <stdio.h>
int sum(int n) {
    if (n != 0)
        // sum() function calls itself
        return n + sum(n-1);
    else
        return n;
}
int main() {
    int number, result;

    printf("Enter a positive integer: ");
    scanf("%d", &number);

    result = sum(number);

    printf("sum = %d", result);
    return 0;

}
```

Explanation:

a. Initially, the sum() is called from the main() function with number passed as an argument.

b. Suppose, the value of n inside sum() is 3 initially. During the next function call, 2 is passed to the sum() function. This process continues until n is equal to 0.

c. When n is equal to 0, the if condition fails and the else part is executed returning the sum of integers ultimately to the main() function.



```
int main() {
   ... ..
   result = sum(number);
   ... ..
}

int sum(int n) {
   if (n != 0)
      return n + sum(n-1)
   else
      return n;
}

int sum(int n) {
   if (n != 0)
      return n + sum(n-1)
   else
      return n;
}

int sum(int n) {
   if (n != 0)
      return n + sum(n-1)
   else
      return n;
}

int sum(int n) {
   if (n != 0)
      return n + sum(n-1)
   else
      return n;
}
```

References:

1. https://www.programiz.com/c-programming
2. https://www.w3schools.com/c
3. https://www.tutorialspoint.com/cprogramming
4. https://github.com/soikot-shahriaar/C-Programming
5. https://youtube.com/playlist?list=PLxgZQoSe9cg1drBnejUaDD9GEJBGQ5hMt&si=051RqSUFezTmsXMH