BASIC C INTRODUCATION

C is a general-purpose, high-level language that was originally developed by Dennis M. Ritchie to develop the UNIX operating system at Bell Labs. C was originally first implemented on the DEC PDP-11 computer in 1972.

In 1978, Brian Kernighan and Dennis Ritchie produced the first publicly available description of C, now known as the K&R standard.

The UNIX operating system, the C compiler, and essentially all UNIX application programs have been written in C. C has now become a widely used professional

language for various reasons:

* Easy to learn
* Structured language
* It produces efficient programs
* It can handle low-level activities
* It can be compiled on a variety of computer platforms

**Data Types**

**1.Basic Types:**

They are arithmetic types and are further classified into: (a) integer

types and (b) floating-point types.

2 .**Enumerated types:**

They are again arithmetic types and they are used to define variables

that can only assign certain discrete integer values throughout the

program.

3.**The type void:**

The type specifier *void* indicates that no value is available.

4 **Derived types:**

They include (a) Pointer types,

(b) Array types,

(c) Structure types,

(d)Union types, and

(e) Function types.

TDATA **TYPESY**

**Variable:-E**

Svariable is nothing but a name given to a storage area that our programs can

manipulate. Each variable in C has a specific type, which determines the size

and layout of the variable's memory; the range of values that can be stored

within that memory; and the set of operations that can be applied to the

variable.

The name of a variable can be composed of letters, digits, and the underscore

character. It must begin with either a letter or an underscore. Upper and

lowercase letters are distinct because C is case-sensitive. Based on the basic

types explained in the previous chapter, there will be the following basic variable

types:

**Type**

* char :-Typically a single octet (one byte). This is an integer type.
* int :-The most natural size of integer for the machine.
* float :-A single-precision floating point value.
* Double:- A double-precision floating point value.
* Void:- Represents the absence of type.

**Operator:-**

An operator is a symbol that tells the compiler to perform specific mathematical

or logical functions. C language is rich in built-in operators and provides the

following types of operators:

* Arithmetic Operators
* Relational Operators
* Logical Operators
* Bitwise Operators
* Assignment Operators
* Misc Operators

**Arithmetic Operators**

* + :-Adds two operands.
* -:-Subtracts second operand from the first.
* \*:-Multiplies both operands.
* / :-Divides numerator by de-numerator.
* %:- Modulus Operator and remainder of after an integer division.
* ++:- Increment operator increases the integer value by one.
* --:- Decrement operator decreases the integer value by one.

**Relational Operators**

* ==:- Checks if the values of two operands are equal or not. If yes, then the condition becomes true.
* !=:- Checks if the values of two operands are equal or not. If the values are not equal, then the condition becomes true.
* >:-Checks if the value of left operand is greater than the value of right operand. If yes, then the condition becomes true.
* < :-Checks if the value of left operand is less than the value of right operand. If yes, then the condition becomes true.
* >= Checks if the value of left operand is greater than or equal to the value of right operand. If yes, then the condition becomes true
* <= Checks if the value of left operand is less than or equal to the value of right operand. If yes, then the condition becomes true

**Logical Operators.**

&&:- Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.

||:- Called Logical OR Operator. If any of the two operands is non-zero, then the condition becomes true.

!:- Called Logical NOT Operator. It is used to reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make it false.

**Bitwise Operators**

* &:- Binary AND Operator copies a bit to the result if it exists in both operands.
* |:- Binary OR Operator copies a bit if it exists in either operand.
* ^:- Binary XOR Operator copies the bit if it is set in one operand but not both.
* ~:- Binary Ones Complement Operator is unary and has the effect of 'flipping' bits. complement form.
* <<:- Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.
* >>:- Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.

**Assignment Operators**

* = :-Simple assignment operator. Assigns values from right side operands to left

side operand.

* += :-Add AND assignment operator. It adds the right operand to the left operand and assigns the result to the left operand.
* =:- Subtract AND assignment operator. It subtracts the right operand from the left operand and assigns the result to the left operand.
* \*=:- Multiply AND assignment operator. It multiplies the right operand with the left operand and assigns the result to the left operand.
* /=:- Divide AND assignment operator. It divides the left operand with the right

operand and assigns the result to the left operand.

* %=:- Modulus AND assignment operator. It takes modulus using two operands and assigns the result to the left operand.
* <<= :-Left shift AND assignment operator.
* >>=:- Right shift AND assignment operator.
* &=:- Bitwise &= Bitwise AND assignment operator.
* ^= :-Bitwise exclusive OR and assignment operator.
* |=:- Bitwise inclusive OR and assignment operator.

**Misc Operators** ↦ **size of & ternary**

* sizeof() :-Returns the size of a variable
* & :-Returns the address of a variable.
* \*: Pointer to a variable.
* ? ::- Conditional Expression

**Operators Precedence in C**

* Postfix ():- [] -> . ++ - - Left to right
* Unary:- + - ! ~ ++ - - (type)\* & sizeof Right to left
* Multiplicative:- \* / % Left to right
* Additive + -:- Left to right
* Shift:- << >> Left to right
* Relational:- < <= > >= Left to right
* Equality:- == != Left to right
* Bitwise AND:- & Left to right
* Bitwise XOR:- ^ Left to right
* Bitwise OR :-| Left to right
* Logical AND:- && Left to right
* Logical OR:- || Left to right
* Conditional :-?: Right to left
* Assignment :-= += -= \*= /= %=>>= <<= &= ^= |= Right to left
* Comma:- , Left to right

**Statement**

* if statement:- An **if statement** consists of a boolean expression followed by one or more statements.
* if...else statement:- An **if statement** can be followed by an optional **else statement**, which executes when the Boolean expression is false.
* nested if statements:- You can use one **if** or **else if** statement inside another **if** or **else if** statement(s).
* switch statement :-A **switch** statement allows a variable to be tested for equality against a list of values.
* nested switch statements :-You can use one **switch** statement inside another **switch** statement(s).

**if Statement**

An **if** statement consists of a Boolean expression followed by one or more

statements.

**Syntax**

The syntax of an ‘if’ statement in C programming language is:

if(boolean\_expression)

{

/\* statement(s) will execute if the boolean expression is true \*/

}

**if…else Statement**

An **if** statement can be followed by an optional **else** statement, which executes

when the Boolean expression is false.

**Syntax**

The syntax of an **if...else** statement in C programming language is:

if(boolean\_expression)

{

/\* statement(s) will execute if the boolean expression is true \*/

}

else

{

/\* statement(s) will execute if the boolean expression is false \*/

}

**if...else if...else Statement**

An **if** statement can be followed by an optional **else if...else** statement, which is

very useful to test various conditions using single if...else if statement.

When using if…else if…else statements, there are few points to keep in mind:

 An if can have zero or one else's and it must come after any else if's.

 An if can have zero to many else if's and they must come before the else.

 Once an else if succeeds, none of the remaining else if's or else's will be

tested.

**Syntax**

The syntax of an **if...else if...else** statement in C programming language is:

if(boolean\_expression 1)

{

/\* Executes when the boolean expression 1 is true \*/

}

else if( boolean\_expression 2)

{

/\* Executes when the boolean expression 2 is true \*/

}

else if( boolean\_expression 3)

{

/\* Executes when the boolean expression 3 is true \*/

}

else

{

/\* executes when the none of the above condition is true \*/

}

**Nested if Statements**

It is always legal in C programming to **nest** if-else statements, which means you

can use one if or else if statement inside another if or else if statement(s).

**Syntax**

The syntax for a **nested if** statement is as follows:

if( boolean\_expression 1)

{

/\* Executes when the boolean expression 1 is true \*/

if(boolean\_expression 2)

{

/\* Executes when the boolean expression 2 is true \*/

}

}

**switch Statement**

A **switch** statement allows a variable to be tested for equality against a list of

values. Each value is called a case, and the variable being switched on is

checked for each **switch case**.

**Syntax**

The syntax for a **switch** statement in C programming language is as follows:

switch(expression){

case constant-expression :

statement(s);

break; /\* optional \*/

case constant-expression :

statement(s);

break; /\* optional \*/

/\* you can have any number of case statements \*/

default : /\* Optional \*/

statement(s);

}

**Nested switch Statements**

It is possible to have a switch as a part of the statement sequence of an outer

switch. Even if the case constants of the inner and outer switch contain common

values, no conflicts will arise.

**Syntax**

The syntax for a **nested switch** statement is as follows:

switch(ch1) {

case 'A':

printf("This A is part of outer switch" );

switch(ch2) {

case 'A':

printf("This A is part of inner switch" );

break;

case 'B': /\* case code \*/

}

break;

case 'B': /\* case code \*/

}

**The ? : Operator:**

We have covered **conditional operator ? :** in the previous chapter which can be

used to replace **if...else** statements. It has the following general form:

Exp1 ? Exp2 : Exp3;

Where Exp1, Exp2, and Exp3 are expressions. Notice the use and placement of

the colon.

The value of a ? expression is determined like this:

1. Exp1 is evaluated. If it is true, then Exp2 is evaluated and becomes the

value of the entire ? expression.

2. If Exp1 is false, then Exp3 is evaluated and its value becomes the value of

the expression.

LOOP

A loop statement allows us to execute a statement or group of statements

multiple times.

* while loop :-Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
* for loop:- Executes a sequence of statements multiple times and abbreviates the
* do...while loop:- It is more like a while statement, except that it tests the condition at the end of the loop body.
* nested loops:- You can use one or more loops inside any other while,

for, or do..while loop.

**while Loop**

A **while** loop in C programming repeatedly executes a target statement as long

as a given condition is true.

**Syntax**

The syntax of a **while** loop in C programming language is:

while(condition)

{

statement(s);

}

**for Loop**

A **for** loop is a repetition control structure that allows you to efficiently write a

loop that needs to execute a specific number of times.

**Syntax**

The syntax of a **for** loop in C programming language is:

for ( init; condition; increment )

{

statement(s);

}

**do…while Loop**

Unlike **for** and **while** loops, which test the loop condition at the top of the loop,

the **do...while** loop in C programming checks its condition at the bottom of the

loop.

A **do...while** loop is similar to a while loop, except the fact that it is guaranteed

to execute at least one time.

**Syntax**

The syntax of a **do...while** loop in C programming language is:

do

{

statement(s);

}while( condition );

**Nested Loops**

C programming allows to use one loop inside another loop. The following section

shows a few examples to illustrate the concept.

**Syntax**

The syntax for a **nested for loop** statement in C is as follows:

for ( init; condition; increment )

{

for ( init; condition; increment )

{

statement(s);

}

statement(s);

}

The syntax for a **nested while loop** statement in C programming language is as

follows:

while(condition)

{

while(condition)

{

statement(s);

}

statement(s);

}

The syntax for a **nested do...while loop** statement in C programming language

is as follows:

do

{

statement(s);

do

{

statement(s);

}while( condition );

}while( condition );

**Loop Control Statements**

Loop control statements change execution from its normal sequence. When

execution leaves a scope, all automatic objects that were created in that scope

are destroyed.

C supports the following control statements.

break statement:- Terminates the **loop** or **switch** statement and transfers execution to the statement immediately following the loop or switch.

continue statement:- Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.

goto statement :-Transfers control to the labeled statement

**break Statement**

The **break** statement in C programming has the following two usages:

 When a **break** statement is encountered inside a loop, the loop is

immediately terminated and the program control resumes at the next

statement following the loop.

 It can be used to terminate a case in the **switch** statement (covered in

the next chapter).

If you are using nested loops, the break statement will stop the execution of the

innermost loop and start executing the next line of code after the block.

**Syntax**

The syntax for a **break** statement in C is as follows:

break;

**continue Statement**

The **continue** statement in C programming works somewhat like the **break**

statement. Instead of forcing termination, it forces the next iteration of the loop

to take place, skipping any code in between.

For the **for** loop, **continue** statement causes the conditional test and increment

portions of the loop to execute. For the **while** and **do...while** loops, **continue**

statement causes the program control to pass to the conditional tests.

**Syntax**

The syntax for a **continue** statement in C is as follows:

continue;

**continue Statement**

The **continue** statement in C programming works somewhat like the **break**

statement. Instead of forcing termination, it forces the next iteration of the loop

to take place, skipping any code in between.

For the **for** loop, **continue** statement causes the conditional test and increment

portions of the loop to execute. For the **while** and **do...while** loops, **continue**

statement causes the program control to pass to the conditional tests.

**Syntax**

The syntax for a **continue** statement in C is as follows:

continue;

**The Infinite Loop**

A loop becomes an infinite loop if a condition never becomes false. The **for** loop

is traditionally used for this purpose. Since none of the three expressions that

form the ‘for’ loop are required, you can make an endless loop by leaving the

conditional expression empty.

#include <stdio.h>

int main ()

{

for( ; ; )

{

printf("This loop will run forever.\n");

}

return 0;

}

**Function**

A function is a group of statements that together perform a task. Every C

program has at least one function, which is **main()**, and all the most trivial

programs can define additional functions.

You can divide up your code into separate functions. How you divide up your

code among different functions is up to you, but logically the division is such

that each function performs a specific task.

A function **declaration** tells the compiler about a function's name, return type,

and parameters. A function **definition** provides the actual body of the function.

The C standard library provides numerous built-in functions that your program

can call. For example, **strcat()** to concatenate two strings, **memcpy()** to copy

one memory location to another location, and many more functions.

**Defining a Function**

The general form of a function definition in C programming language is as

follows:

return\_type function\_name( parameter list )

{

body of the function

}

**Function Declarations**

A function **declaration** tells the compiler about a function name and how to call

the function. The actual body of the function can be defined separately.

A function declaration has the following parts:

return\_type function\_name( parameter list );

For the above defined function max(),the function declaration is as follows:

int max(int num1, int num2);

Parameter names are not important in function declaration, only their type is

required, so the following is also a valid declaration:

int max(int, int);

**Calling a Function**

While creating a C function, you give a definition of what the function has to do.

To use a function, you will have to call that function to perform the defined task.

When a program calls a function, the program control is transferred to the called

function. A called function performs a defined task and when its return

statement is executed or when its function-ending closing brace is reached, it

returns the program control back to the main program.

**Function Arguments**

If a function is to use arguments, it must declare variables that accept the

values of the arguments. These variables are called the **formal parameters** of

the function.

Formal parameters behave like other local variables inside the function and are

created upon entry into the function and destroyed upon exit.

While calling a function, there are two ways in which arguments can be passed

to a function:

**Call Type Description**

Call by value This method copies the actual value of an argument

into the formal parameter of the function. In this case,

changes made to the parameter inside the function

have no effect on the argument.

Call by reference This method copies the address of an argument into

the formal parameter. Inside the function, the address

is used to access the actual argument used in the call.

This means that changes made to the parameter affect

the argument.

**Call by Value**

The **call by value** method of passing arguments to a function copies the actual

value of an argument into the formal parameter of the function. In this case,

changes made to the parameter inside the function have no effect on the

argument.

By default, C programming uses *call by value* to pass arguments. In general, it

means the code within a function cannot alter the arguments used to call the

function. Consider the function **swap()** definition as follows.

/\* function definition to swap the values \*/

void swap(int x, int y)

{

int temp;

temp = x; /\* save the value of x \*/

x = y; /\* put y into x \*/

y = temp; /\* put temp into y \*/

return;

}

**Call by Reference**

The **call by reference** method of passing arguments to a function copies the

address of an argument into the formal parameter. Inside the function, the

address is used to access the actual argument used in the call. It means the

changes made to the parameter affect the passed argument.

To pass a value by reference, argument pointers are passed to the functions just

like any other value. So accordingly, you need to declare the function

parameters as pointer types as in the following function **swap()**, which

exchanges the values of the two integer variables pointed to, by their

arguments.

/\* function definition to swap the values \*/

void swap(int \*x, int \*y)

{

int temp;

temp = \*x; /\* save the value at address x \*/

\*x = \*y; /\* put y into x \*/

\*y = temp; /\* put temp into y \*/

return;

}