Chapter – 2

Variables:

An object, sometimes called a variable, is a location in storage, and its iterpretation depends on mainly two atrtributes:

- Its *Storage Class*, Storage Class determines the lifetime of the storage associated with the variable.
- Its *Type*, Type determines the meaning of the values found in the identified object.

Constant:

A Constant, can be simply defined as value given to an object. There are several kinds of constants.

- *Integer Constants*, An integer constant consists of sequence of digits. The type of integer constants depends upon its type (type specifiers), value and suffix. Integer constants can be suffixed with *u* or *U* to specify that it is unsigned. They can suffixed with *l* of *L* to specify that they are long. Prefixing with 0 or 0x specifies that the latter one is an octal constant and the former one is an hexadeecimal constants.

- *Character Constants*, A Character constant is an integer, written as one character within single quotes, such as '*a*'. The value of the character constant is the ASCII value of the character written in the machines character set.

- *Floating Constants*, A Floating constant consists of a integral part, a decimal point, a fractional part, an 'e' or 'E', an optionally assigned integer exponent and optional type suffix, one of f, F, l, or L.

- *Enumeration Constants,* Enumerations are unique types with values ranging over a set of named constants called enumerators. Enumerators provide convienient way to associate constant values to names. The first name in an enum has value 0, the next 1. Names in different enumerations must be distinct.

- *String Constants or String literals,* Strings Constant is a sequence of zero or more characters enclosed in doubles quotes. The quotes are not part of the string, but serve only to delimit the string. The internal representation has a null character ('*0*') at the end, so storage requred is one more than the length of the string.

Declaration:

A Declaration announces the properties of the variable, primarily its type. All variables are declared before their use. A declaratin specifies type and contains a list of one or more variables of that type. Declarations do not reserve the storage space for the variable. Variables are declared not constants.

A simple declaration can be,

data_type variable_name;

Definition:

A Defintion reserves the storage space for the variable. The storage space reserved for the variable depends on the size of the data type of the variable. The above declaration can sometimes be a definition or both. It depends on the place of declarations. All declarations are not definitions.

Initialization:

If in a declaration, a name is followed by equal to sign and an expression, or a function call, the expression serves as an initializer. This is initialization. Initialization is done once only, conceptually before the program stars executing, and the initializer must be a constant expression.

Data Type:

A Data Type defines the type of the variables, tells the storage space for the variable, identifies the operations that can be performed on the variable and how the data in represented in memory. There are few basic data types in C:

- *char,* a single byte, capable of holding one character.
- *int,* an integer, typicallyreflecting the natural size of integers on the host machines.
- *float,* single-precision floating point
- *double,* double-precision floating point.

Each compiler is free to chose appropriate sizes for its own hardware subject only to the restriction that *shorts* and *ints* are atleast 16 bits and *longs* are atleast 32 bits.

On *gcc (32 – bit machine),* size of *short int* is 16 bits, *int* and *long int* is 32 bits.

Identifiers:

An Identifier is a squence of letters and digits. The first character must be a letter, even _ is considered as a letter. Upper case and lower case letters are different. All variables are identifiers.

Keywords:

Some identifiers are reserved for use as keywords, and cannot be used as identifiers. Keywords define the functionality of the C language. Each keyword has a specific function. There are 32 keywords.

Operators:

An operator can be termed as a symbol. They allow us to perform different operations on operands. An opeand can be a variable or a constant. Operators in C can be categorized as:

- Arithmetic Operators,
  - ' + ', performs addition
  - ' - ', performs substraction
  - ' * ', performs multiplication
  - ' / ', performs division
  - ' % ', performs modulus operation, yeilds remainder

- Relational Operators,
  - ' > ', greater than operator
  - ' < ', less than operator
  - ' >=', grater than or equal to operator
  - ' <= ', less tham operator
  - ' == ' *and* ' *!*= ', equality operator

- Logical Operators,
  - ' || ', performs OR operation
  - ' &&', performs AND operation.
  - ' *!* ', performs NOT operation

- Bitwise Operators,
  - ' & ', performs bitwise AND operation
  - ' | ', performs bitwise OR operation
  - ' ^ ', performs butwise XOR operation
  - ' ~ ', performs negation operation
  - ' << ', performs bitwise left shift operation
  - ' >> ', performs bitwise right shift operation

- Increment and Decrement Operators,
  - ' ++ ', increases the value by 1
  - ' -- ', decreases the value by 1

- Assignment Operators, ' = ', ' += ', ' -= ', ' *= ', ' /= '
- Ternary Operator, ' ? ' and ' : ', can be used as an conditional statement
- Comma Operator ' , ' seperates variables, constants
- sizeof operator ' *sizeof* ' , gives the size of the operand
- ' *&* ', Address operator, it gives the address of the operand

Type Conversions:

When an operator has operands of different types, they are converted to common type according to a small set of rules. Automatic conversions are done on narrow operands, will be converted to wider operands, without losing information.

If an operator like ' + ' or ' * ', that takes two operands, has operands of different types, the operand with lower type is promoted to the higher type. The result is of the higher type.

These are the formal set of rules:

> *“ If either operand is long double, convert the other to long double.*
> *Otherwise, if either operand is double, convert the other to double.*
> *Otherwise, if either operand is float, convert the other to float.*
> *Otherwise, convert the char and short to int.*
> *Then, if either operand is long, convert the other to long. “*

Precedence and Order of Evaluation:

Operator precedence in C Language specifies the priority of operators in expressions. Order of Evaluation tells how an expression should be evaluated. Associativity defines the of an operator is a property that determines how operators of the same precedence are grouped in the absence of parentheses. C language do not specify the order in which the operands of an operator are evaluated. Similarly, the order in which function parameters are evaluated are not specified.

Macro v/s Enum:

S. No	MACRO	ENUM
1	*#define* replaces the subsequent occurances of token names with a replacement text.	*enum* is a list of names with constant integer values.
2	Macros do not have any storage space aassociated with them	Enumerations have storage space associated. Typically 4 bytes
3	Macros are processed in preprocessor stage, so, programming errors are not identified.	*enum* is processed in compilation stage and programming errors may occur
4	Programmers need to define macros for each name associated	*enum* is a convinient way to associate constant values to names, with values generated for you,

Type Qualifiers:

Types may be qualified to indicate special properties of the objects being declared. There are two type qualifiers:

1. *const,* A const object may be initialized but cannot be assigned to there after. It announces that the objects will be placed in read-only memory once it is initialized.

2. *volatile,* The purpose of volatile is to force an implemetation to suppress optimization that could otherwise occur. By specifying volatile with an object declaration, it tells the compiler that the value of the variable may change at any time. It also specifies to compiler fetch the value of object from main memory always. Practically, volatile keyword is very useful in embedded applications.

   Consider this example:
   Embedded systems contain real hardware, usually with sophisticated peripherals. These peripherals contain registers whose values may change asynchronously to the program flow. As a very simple example, consider an 8-bit status register that is memory mapped at address 0x1234. It is required that you poll the status register until it becomes non-zero. The naive and incorrect implementation is as follows:

```
uint8_t *pReg = (uint8_t *) 0x1234;

// Wait for register to become non-zero
while (*pReg == 0) { } // Do something else
```

   This will almost certainly fail as soon as you turn compiler optimization on, since the compiler will generate assembly language that looks something like this:

```
mov ptr, #0x1234
mov a, @ptr

loop:
      bz loop
```

   The rationale of the optimizer is quite simple: having already read the variable's value into the accumulator (on the second line of assembly), there is no need to reread it, since the value will always be the same. Thus, in the third line, we end up with an infinite loop. To force the compiler to do what we want, we modify the declaration to:

```
uint8_t volatile * pReg = (uint8_t volatile *) 0x1234;
```

The assembly language now looks like this:

```
mov ptr, #0x1234
loop:
 mov a, @ptr
 bz loop
```

The desired behavior is achieved.

Type Specifiers:

Type specifiers specify the length and type of the variable. They are:

- *void*
- *char*
- *short*
- *int*
- *long*
- *float*
- *double*
- *signed*
- *unsigned*
- *struct or union*
- *enum*
- *typedef*

At most one of the word *long* or *short*, can be specified with *int.* The word *long* may be specified with *double.* Either of *signed* or *unsigned* can be specified with either of *long* or *short* of *int and char.*

Tokens:

Identifiers are classied into six groups known as tokens. The six types of tokens are:

1. Keywords
2. Constants
3. String Literals
4. Operators
5. Seperators
6. Identifiers

If the input stream has been seperated into tokens up to a given character, the next token is the longest string of characters that could constitute a token.

Comments:

The characters ' /* ' include a comment, which terminates with the characters ' */ '. Comments do not nest, and they donot occur within string or character literals.

Void:

The non exixtent value of a *void* object may not be used in any way, and neither explixit nor implicit conversion to any non-void type may be applied. Because a *void* expressions denotes a non-exixtent value, such as an expression may be used only where the value is not required.

Chapter 3

Control Flow:

The control flow statements of a language specify the order in which the computations are performed.

Statements:

An expression such as *x = 0* or *printf ("....")* becomes statement when it is followed by a semicolon.

```
X = 0;
printf ("....");
```

where, ' ; ', is a statement terminator.

Blocks:

Braces { and } are used to group declarations and statements together into a compound statement or block, so that multiple statements which combinlet perform a specific task are enclised within braces.
There is no semicolon after the right brace that ends the block.

```
{

        .......;
        .......;
        .......;

}
```

if – else:

It is used to express decisions. It takes the form,

```
if ( expression )
        statement1;
else
        statement2;
```

The expression is evaluated, if the result is a non-zero value, then statement1 is executed, otherwise statement2 is executed.

else – if:

It takes the form,

```
if ( expresseion1)
        statement1;
else if ( expression2)
        statement2;
else if ( expression3)
        stetement3;
else
        statement4;
```

This sequence of if statements is the most general way of writing a mulit-way decision. The *expressions* are evaluated in order; if any *expression* is true, the *statement* associated with it is executed; and this terminates the whole chain. The *else* part handles the *"none of the above"* or default case where none of the other conditions is satisfied.

switch:

The switch statement is a multi-way decision that tests weather an expression matches one of a number of constant integer values and branches accordingly.
It takes the form,

```
switch (expression) {

case const_expr:        statements;
case const_expr:        statements;
case const_expr:        statements;
default:                statements;

}
```

Each case is labelled by one or more integer-valued or constant expressions. If a case matches the expression value, execution starts at that case. The case *default* is executed if none of the other cases are executed. The *break* statement causes immediate exit from the *switch.* Because cases serve as just labels, after the code for one case is done, execution falls through to the next unless you take explicit action to escape. *break* and *return* are the most common ways to leave a *switch.* Falling through, sometimes allow several cases to be attached to a single action. Falling through from one case to other is not robust, being prone to disintegration when the program is modified.

Loops – *while* and *for:*

while takes the following form,

```
while (expression) {

        statements;

}
```

If the expression evaluatedis *true,* i.e, if it yeilds a non-zero result the *statament* is executed and the *expression* is re-evaluated. This as a cycle until the *expression* becomes zero.

For takes the following form,

```
for (expr1; expr2; expr3 ) {

        statements;

}
```

It is equivalent to,

```
expr1;
while (expr2) {

        statements;
        expr3;

}
```

Gramatically, three components of the *for* loop are *expressions.* Most commonly *expr1* and *expr3* are assignments or function calls and *expr2* is a relational expression. You can ommit the expressions but semicolon's must be remained.

do – while Loop:

while and *for* loops test the termination condition at top. The third loop in C, the *do – while*, tests the bottom after making each pass through the loop body, the body is executed at least once.

```
do {

        statements;

} while ( expression );
```

The *statement* is executed, then the *expression* is evaluated. If it is true, *statement* is evaluated again and so on. When the expression becomes false the loop terminates.

break and *continue*:

It is sometimes convinient to be able to exit from a loop other than by testing at the top or bottom. The break statement provides an early exit from enclosing *loop* or *switch*. A *break* causes the innermost enclosing *loop* or *switch* to be exited immediatley.

The *continue* statement is related to *break,* but less often used; it causes the next iteration of the enclosing *for, while or do – while* to begin. The *continue* statement applies only to loops, not *switch.*

***Goto* and *Labels*:**

C provides *goto* statements and *labels* to branch to. *goto* can be used to abandon processing in some deeply nested structures, such as breaking out of two or more loops at once.

A *label* has the same form as a variable name, and is followed by a colon. It can be attached too any statement in the same function as the *goto*. The scope of label is the entire function.