Variables

A ***Variable*** is an identifier for a value. It refers to a space (location) in the memory that holds a value.

Ex -

```
#include<stdio.h>
int main()
{
    int a = 4;
    // here, the 'a' is a variable
    // the data type of the variable is 'int' or Integer.
    // this means, the variable can only hold values that are defined in the integer
range
    // the value that is held by the variable a is '4'. This 4 is also known as a
literal.
    // there can be any number of variables in a C program and of any data type that is
supported by C.
    char c = 'A';
    float f = 4.0f;
    double = 3.14;
    return 0;
}
```

The variables follow the rules of identifiers for their naming convention.

That means,

1. The variable name ***can not*** have a special symbol in it. **[!,@,#,%,^,&,*]**
2. The variable name ***can*** have an underscore in the variable name **[_]**
3. The name of a variable ***can not*** start with a number, however, the variable name can have a number in it. Ex -

   ```
   int 1a; // this is an illegal variable name. writing this in a program will lead to
   syntax error (Compilation Error)
   int a1; // this is a legal variable name.
   ```

Declaring a variable

In C, before you can use a variable to store some value, you will have to declare it. Declaring variables is the way in which a C program shows the number of variables it needs, what they are going to be named, and how much memory they will need.

Within the C programming language, when managing and working with variables, it is important to know the *type* of variables and the *size* of these types. A type's size is the amount of computer memory required to store one value of this type. Since C is a fairly low-level programming language, the size of types can be specific to the hardware and compiler used – that is, how the language is made to work on one type of machine can be different from how it is made to work on another.

Here is an example of declaring an integer, which we've called `some_number`. (Note the semicolon at the end of the line; that is how your compiler separates one program *statement* from another.

```
int some_number;
```

This statement means we're declaring some space for a variable called some_number, which will be used to store integer (`int`) data. Note that we must specify the type of data that a variable will store. There are specific keywords to do this `int`, `char`, `float`, `double` etc.

Multiple variables can be declared with one statement, like this:

```
int anumber, anothernumber, yetanothernumber;
```

We can also declare *and* assign some content to a variable at the same time.

```
int some_number = 3;
```

This is called ***initialization***.

In early versions of C, variables had to be declared at the beginning of a block. In C99 it is allowed to mix declarations and statements arbitrarily – but doing so is not usual, because it is rarely necessary, some compilers still don't support C99 (portability), and it may, because it is uncommon yet, irritate fellow programmers (maintainability).

After declaring variables, you can assign a value to a variable later on using a statement like this:

```
some_number = 3;
```

You can also assign a variable the value of another variable, like so:

```
anumber = anothernumber;
```

Or assign multiple variables the same value with one statement:

```
anumber = anothernumber = yetanothernumber = 3;
```

This is because the assignment `x = y` returns the value of the assignment. `x = y = z` is really shorthand for `x = (y = z)`.

Naming Variables

Variable names in C are made up of letters (upper and lower case) and digits. The underscore character ("_") is also permitted. Names must not begin with a digit.

Some examples of valid (but not very descriptive) C variable names:

```
foo
Bar
BAZ
foo_bar
_foo42
QuUx
_
```

Some examples of invalid C variable names:

```
2foo    (must not begin with a digit)
my foo  (spaces not allowed in names)
$foo    ($ not allowed -- only letters, and _)
while   (language keywords cannot be used as names)
```

Literals

Anytime within a program in which you specify a value explicitly instead of referring to a variable or some other form of data, that value is referred to as a **literal**.

Ex -

```
int value = 4;  //  here, 4 is a literal
```

The Four Basic Data Types

In Standard C there are four basic data types. They are `int`, `char`, `float`, and `double`.

The `int` type

The `int` type stores integers in the form of "whole numbers". Examples of literals are whole numbers (integers) such as 1,2,3, 10, 100... When `int` is 32 bits , it can store any whole number (integer) between -2147483648 and 2147483647.

If you want to declare a new int variable, use the `int` keyword. For example:

```
int numberOfStudents, i, j=5;
```

In this declaration we declare 3 variables, numberOfStudents, i and j, j here is assigned the literal 5.

The `char` type

The `char` type is capable of holding only a single character. It stores the same kind of data as an `int` (i.e. integers), but typically has a size of one byte. A variable of type `char` is most often used to store character data, hence its name. Most implementations use the ASCII character set as the execution character set, but it's best not to know or care about that unless the actual values are important.

Examples of character literals are 'a', 'b', '1', etc., as well as some special characters such as '`\0`' (the null character) and '`\n`' (newline, recall "Hello, World"). Note that the `char` value must be enclosed within single quotations.

When we initialize a character variable, we can do it two ways. One is preferred, the other way is **bad** programming practice.

The first way is to write

```
char letter1 = 'a';
```

This is *good* programming practice in that it allows a person reading your code to understand that letter1 is being initialized with the letter 'a' to start off with.

The second way, which should *not* be used when you are coding letter characters, is to write

```
char letter2 = 97; // in ASCII, 97 = 'a'
```

This is considered by some to be extremely **bad** practice, if we are using it to store a character, not a small number, in that if someone reads your code, most readers are forced to look up what character corresponds with the number 97 in the encoding scheme. In the end, `letter1` and `letter2` store both the same thing – the letter 'a', but the first method is clearer, easier to debug, and much more straightforward.

One important thing to mention is that characters for numerals are represented differently from their corresponding number, i.e. '1' is not equal to 1. In short, any single entry that is enclosed within 'single quotes'.

There is one more kind of literal that needs to be explained in connection with chars: the **string literal**. A string is a series of characters, usually intended to be displayed. They are surrounded by double quotations (" ", not ' '). An example of a string literal is the "Hello, World!\n" in the "Hello, World" example.

The string literal is assigned to a character **array**, arrays are described later. Example:

```
char name[] = "John Doe\n";
```

The `float` type

`float` is short for **floating point**. It stores inexact representations of real numbers, both integer and non-integer values. It can be used with numbers that are much greater than the greatest possible `int`. `float` literals must be suffixed with F or f. Examples are: 3.1415926f, 4.0f, 6.022e+23f.

It is important to note that floating-point numbers are inexact. Some numbers like 0.1f cannot be represented exactly as `float`s but will have a small error. Very large and very small numbers will have less precision and arithmetic operations are sometimes not associative or distributive because of a lack of precision. Nonetheless, floating-point numbers are most commonly used for approximating real numbers and operations on them are efficient on modern microprocessors.Floating-point arithmetic is explained in more detail on Wikipedia.

`float` variables can be declared using the `float` keyword. It is used when less precision than a double provides is required.

The `double` type

The `double` and `float` types are very similar. The `float` type allows you to store single-precision floating point numbers, while the `double` keyword allows you to store double-precision floating point numbers – real numbers, in other words. Its size is 8 bytes on most machines. Examples of `double` literals are 3.1415926535897932, 4.0, 6.022e+23 (scientific notation). If you use 4 instead of 4.0, the 4 will be interpreted as an `int`.

The distinction between floats and doubles was made because of the differing sizes of the two types. When C was first used, space was at a minimum and so the judicious use of a float instead of a double saved some memory. Nowadays, with memory more freely available, you rarely need to conserve memory like this – it may be better to use doubles consistently. Indeed, some C implementations use doubles instead of floats when you declare a float variable.

If you want to use a double variable, use the `double` keyword.