The real C language

by Yannis Kiourktsoglou

12th Edition – December 2016

Index

C Language / Lesson 1

1. The rules of the game

Find in this lesson : Data Types Constants Scalars & aggregates sizeof

1.1. A first program in C

Let's write a simple program that will help us introduce the basic syntax rules of a C program.

```
// My First Program
#include <stdio.h>
void main()
{       printf("My First Program\n");
}
```

This program here is

- complete
- error free
- has no dependencies on other modules (apart from the standard C libraries)
- runs successfully

Let's now describe what we see inside this small piece of code :

Case sensitivity

To begin :
The code is **case sensitive :** capital (upper-case) letters cannot be interchanged with their corresponding small (lower-case) letters **.** This means that if I wrote VOID instead of void or Printf instead of printf , this would be an error. It is recommended that lower-case is used for variable names and function names for simplicity. Later we will see cases where traditionally capital letters (upper-case) are used.

Comments

`// My first program` in the above sample program is a comment line
There are two different ways to add comments to C code :
i. The text following two slashes // in one line is ignored by the compiler (this means that the two slashes do not have to be at the beginning of the line
ii. The other way to add comments to the source code is enclose them inside /* and */
Example:

```
        printf("Hello"); /* this is also a comment */
```

Note: The latter can be used for multi-line comments, it is therefore recommended to use the // for single line comments so that if later needed these can be commended out with a multi-line comment

Include files

An include file has the extension `.h` because it is also called a "header file" . It is a text file (it can be edited with any editor like Notepad). So an include file is not used during linkage like machine code modules and libraries (.lib) or execution of a program line dynamic linked libraries (.dll). Instead it is used as source code during compilation of the program.

The `#include` statement is a pre-processor directive like all statements in C beginning with a #. When a program is compiled every time it "sees" an #include statement it loads the file mentioned in the statement and treats it as if it was part of the program.
It is easily understood that the purpose of include files is to avoid repeating common pieces of code frequently used by typing it again and again in more than one programs. Instead we write the code in an include file and whenever we need it in our program we "include" it.
Such code is usually the function prototypes explained in the next paragraph.

Functions

Before introducing the functions that appear in the given code, here is a general discussion on functions :

A function is a named set of program statements that can be executed (called) from many different points in a program. A function based on its design, can accept 0 , 1 or more arguments (passing parameters is another equally correct term for arguments) but when its execution is completed it can return maximum <u>one</u> result (return value).

Examples :

1. the statement

   ```
   x = foo(a,b,c) ;
   ```

   executes the function foo which was designed to accept three arguments and the return value when the execution is completed is assigned to variable x .

2. the statement

   ```
   reset() ;
   ```

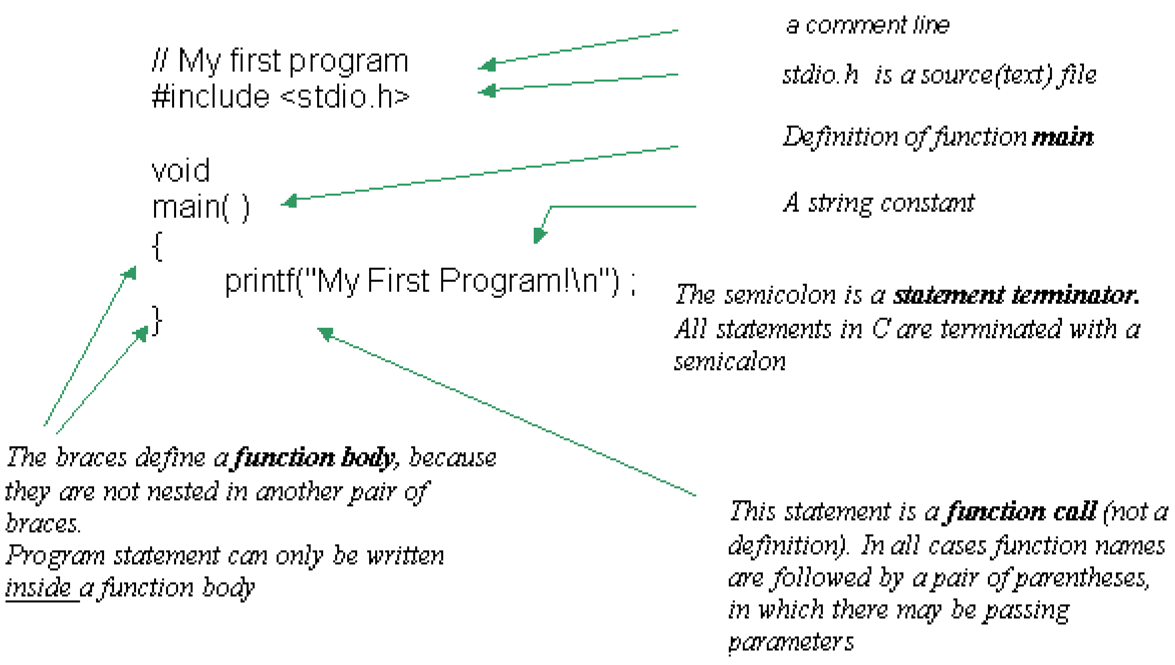
   executes the function reset <u>without</u> passing any arguments and no result is returned.

Functions is a term used in almost all programming languages but unfortunately it does not always have the same meaning. Other keywords like "Procedure", "Command", "Subroutine" etc. are combined with the keyword "function" to distinguish between reentrant code that has different behaviours. Some languages for instance use a different term for functions that <u>do not</u> return a result. In Visual Basic for example there are two types of procedures : Functions returning a result and Subroutines not returning anything.

In C language things are a little easier :

- Every program statement is in a function (see below)
- Unlike other languages it is called a function regardless of whether it returns or not a result – of course as in all languages a function can return no result or only one result value.
- In C every name that is directly followed by a pair of parentheses (empty or not) is the name of a function. For instance if someone asks you what is the meaning of `blabla()` found in a program, for one thing you can be sure : blabla is a function.

Only exception to this rule is the operator **sizeof(...)** which will be explained later.



```
// My first program
#include <stdio.h>

void
main( )
{
        printf("My First Program!\n") ;
}
```

a comment line

stdio.h is a source(text) file

*Definition of function **main***

A string constant

*The semicolon is a **statement terminator.** All statements in C are terminated with a semicalon*

*The braces define a **function body**, because they are not nested in another pair of braces.*
Program statement can only be written inside a function body

*This statement is a **function call** (not a definition). In all cases function names are followed by a pair of parentheses, in which there may be passing parameters*

In our first program however we see two names followed by parentheses:

a. main

b. printf

According to what has been already mentioned , these two must be function names.
Indeed they are function names, but with a very big difference :

main is created in this program while **printf** already exists and we don't know who created it (probably a good programmer working for the vendor of our C compiler). In other words a new function called main was written in this program and in this function we used the function printf.
How do we know that main is created here ? Simply because the word main is outside any pair of (curly) braces . Braces {} are used a lot in C and most of them are nested, i.e. inside other braces like this

```
{   blablabla ....
    {    blablabla ....
         blablabla ....
    }
}
. . .
```

When we create - build - write a new function then this cannot be inside another pair of braces.
Following is an intro and then a paragraph that you must make sure that you have fully understood because the terms "definition" and "declaration" will be around throughout this

manual not only regarding function names (as it is in this section) but also regarding variable names and their scope (see later Storage classes in another section).

* Defining a function means creating a function.
A **function definition** consists of
- a data type
- a function name and
- a body

Example:
```
  int foo(int x) { return x+x*2; }
```
Since the language has no column or line restrictions, this function definition can be written for readability reasons as follows :
```
 int
 foo(int x)
{ return x+x*2;
}
```

* **A function body** immediately follows the name of the function that is being defined and a pair of parentheses containing a list of arguments (can be empty). **Function bodies** are never nested (that's why every pair of nested braces does **not** define a function body and only pairs of braces <u>not enclosed</u> within other braces define a function body as in the above example)
All the code (run-time instructions) of a C program must be written inside function bodies. Data definitions and other declarations may exist outside function bodies

* **main( )** function demonstrated above is an agreed upon function that must always exist in a program as the start-up function [Program entry]

1.2 Defining vs. declaring vs. calling

These three very important concepts must be clear to the reader from the very beginning. Their importance will be visible again later in the "Storage Classes" topic.

In a C program the presence of an identifier followed by parentheses means ***"this is a function"***.
Only exception is the reserved keyword `sizeof( ... )` which is presented later.

A function name however appearing in the code may have three different roles :

a. **Definition** of a name means creation of the name. If you define a name then you create it and of course you cannot re-define it in its scope (the area where it is visible).

   ➢ ***A variable or function is defined only once***

   Talking about **function definitions** previously we discussed the details about the format of these definitions. A function defined by you is "your baby" and can be part of a program when it is used by that program only or part of another module or a library (=a collection of program modules) which is used by more that one programs
   ⇨ Function `main()` is always defined in every project because that's where the program start is.

b. **Declaration** of name means a statement that the name exists (has been created somewhere else) and a description of what it looks like .

   ➢ ***A variable or function can be declared many times***

   Function declarations are called **prototypes** and usually reside inside **include files** like the stdio.h mentioned in the first program above.

c. A **function call** is a request to execute an existing function
   A function cannot be called if it not previously prototyped
   That is why for every function that a program uses the corresponding include file must be included otherwise a "not declared" error will occur. For example :
   for the statement
   `n = getch() ;`
   to be executed
   i. `n` must be defined
      we will see how variables are defined in the next section
   ii. the prototype of `getch()` must be present
      this will be done with this line
      `#include <conio.h>`
      because the include file conio.h contains the prototype of getch()

1.3. Data Types

1.3.1 G e n e r a l

C type declarations have the general form:

```
[class][type] declarator [=initialiser] [, declarator [=initialiser] ...]
```

All variables must be declared before they can be used. (see Storage classes in a later chapter)

A variable name (identifier) can be modified with :
- a trailing * to signify a pointer
- [N] where N is a positive integer to signify an aggregate (array)
A declaration of an identifier may also be followed by (...) to signify function returning data type .

It is too early to explain what pointers or aggregates are (we will talk about these later in this chapter) but some examples can clarify what the previous paragraph says :

In the definitions

```
char *last ;
int measure[30];
```

the words **last** and **measure** are variable names, but the trailing * and the leading [30] signify a pointer in the first case and an aggregate (array) in the latter. However the variable names will appear in the source code without these extra notations.

1.3.2 Binary Integer Types

char Character Data Type

The char type declares an object to be of "plain" binary type which is large enough to store an ASCII character. In ASCII character set the size of a character code is eight bits (1 Byte). It must be mentioned here that for multilingual purposes a "huge" comparing to ASCII character set called UNICODE has been introduced and is widely used. In Unicode a character occupies two bytes but still the type char always means ASCII character. A char may contain **any 8-bit** representation, not just the ASCII characters. The data type char is signed by default and therefore the range of values of a char is -128 to 127, inclusive. Usually however we prefer to use chars as unsigned integers. This can be done using the modifier **unsigned** (see below). Unsigned characters are >=0 and give a range from 0 to 255 (a total of $256=2^8$ different values)

In Microsoft C char variables may be defined by default as unsigned using the /J compile-time switch.

Conclusion #1: sizeof(char) is 1 (byte).
Conclusion #2: the range of an unsigned char value is 0 to 255.

Note : The reserved word `sizeof(... )` is called **'a pre-processor operator'** and is a syntax exception because it looks like a function i.e. it has a name like any identifier and is followed by parentheses.

sizeof(...) returns the allocated size in bytes by a defined variable or of a known data type as in the above example (this is a constant number and is known at compilation time)

Examples

```
char value, birth_date[6];
char *pointer_to_char;
char action_code = 12;
char selected_lang = 'G';
```

Note 1: Definition with/without initialisation

A variable can defined but not initialised as in the first two examples above**.** The third and fourth examples are definitions with initialisation.

Note 2: A first warning about string constants

A character constant may be assigned to a character variable as well as shown in the above example ('G'). It is very important to make clear that **a character constant in C is totally different from a string constant.**

- A **character constant** represents the value of one character which is written enclosed in single quotes as in 'G'
- A **string constant** represents an aggregate (a group) of characters and is always written with the characters enclosed in double quotes . Example **"Please press a key"**

Although we are in the middle of a presentation of data types, an introduction to constants will be done right here.

Read section 1.3.4 titled "More about constants" for the rest on constants.

Numeric constants

These are the basic rules for numeric constant notation in C

The 1st character of a numeric constant is always a digit. Any identifier beginning with a character other than a digit is a label or the name of a variable or function (i.e. an identifier)	
If the 1st digit is not a 0 , then the constant is a **decimal** constant	Example : 5467
If the 1st digit is = 0 , then the constant is an **octal** constant	Example : 033
If the two leading characters are 0x , then the constant is a **hex** constant (note : the x may be upper or lower case)	Example : 0xEFFF

Note : For those not familiar to numbering systems other than the decimal., it must be explained that hex numbers can contain digits (0 . . . 9) and / or the letters A,B,C,D,E and F. Some examples of erroneous syntax will help you understand :

- The figure 012F is incorrect because the number is octal (see above) and cannot contain the F .
- The figure 058 is incorrect because although the number is again octal , it contains an 8 which is not used in the octal system
- 54EA is incorrect because a decimal number cannot contain letters

short Short Integer Data Type

The keyword short may be applied to the int type. The short int type declares an object to be of type int and size 16-bits (2 bytes). **short int** must be the same or less in size than an int.

The keyword "unsigned" may be used as a prefix to short int, to identify the short integer as being unsigned.

The range of values of a **2-byte signed int** is -32768 to 32767 inclusive.

If short is used without the keyword int, the int is assumed.

Examples

```
short int value = 6;
short x,y,z;
short int counters[6];
```

Conclusion : sizeof(short) is 2.

long Long Integer Data Type

The keyword long applied to the int type, declares an object to be of integer binary type. The size of a long int object is four bytes and its range of values is -2147483648 to 2147483647, inclusive.

A long int is guaranteed to be the same or greater in size than an int. However, usually the int type is the same as short.
Therefore it is suggested that *declarations* of long ints are done with the type long

Long constants

An integral constant with the suffix L (or l) is interpreted as a long int constant, as is any integral constant that is too big to be represented in an int.

Examples

```
long int value = 6L;
long equip_info = 0X0040L;
long counters[6];
long *lptr;
```

`int` `Integer Data Type`

Some books "say" that "*An **int** is the same or greater in size than a short int, and the same or less than the size of a long int"*.

The truth : Originally `int` was (by definition) a binary integer of size **equal to the data bus size** of the processor, the *'word'* as they call it. For many years and in PC operating systems this happened to be = 16 bits (2 bytes) and that's how it ended up for many developers that **int** meant *"a 2 byte binary integer"*. In older DOS/Windows C compilers, an **int** was short int and thus the size of an int variable is two bytes and ints are signed. (That is, int is equivalent to signed short). Nevertheless you should not count on that ; you will encounter these days compilers treat the `int` data type as 'a 4 byte (long) binary integer' because their data buses are 32 bit wide (or the compiler is a 32 bit version).

If you feel that you have to be on the safe side , then you must use the keywords short and long explained below.

What is the idea behind using int if this means that we do not know its size since it depends on the size of the data bus . The answer is simple but also sophisticated : No matter if you want to transfer a value of less width than the data bus from the processor to the memory (or memory to processor) a full sized value will travel on the bus i.e. a value with the size of the data bus. Result: the speed of a transfer is the same for every value of the size of the data bus or shorter.

It is not in this manual's scope to explain what the Stack is but that's where the passing parameters are temporarily stored when a function is called. And ALL the values on the Stack must have the size of the accumulator. This means that on a 32 bit computer if a passing parameter is of size 16 bits (2 bytes) it will be passed on the stack as a 32 bit value.

Examples

```
int rec_count = 0;
int bit_pattern = 0X7FFF;
int counters[6], *ip;
```

Type Modifiers: Keywords that Modify Data Types

Type modifiers are keywords that can be used to prefix a data type in a declaration. They modify the type by changing its storage size (e.g. short and long) or the way in which it is interpreted (e.g. signed and unsigned.) etc.

Examples

```
unsigned long ul;
signed short int ssi;
```

1.3.3 Floating point Types

All types presented above are binary integer types that is they don't accept real numbers as values (containing a fractional part)

Following are the types that do this job. However it must be mentioned that there is nothing really special about floating point types in C : They are treated as in most languages (Pascal, Basic etc) and they are mainly used for numeric figures which must support the relevant characteristics in order to be involved in math calculations. That is why an emphasis must be given in C to the binary integer type where C does a lot more than any other language and in more optimised ways, starting from text processing and parsing to bit-wise manipulations.

`float` Single precision floating point Data Type

This type declares 4-byte floating point variables. It must be mentioned that unlike integer data types, ranges is not the important issue here since due to the presence of the exponent part of floating point numbers (see note below) the range is usually wider that required. What matters here is how may significant digits can a float variable hold. The answer is that single precision numbers can have approximately 6 significant digits. For example number 12345678 has 8 significant digits and doesn't therefore fit in a single precision variable, whereas number 123000000 although bigger than the former has only 3 significant digits and does fit in a single precision variable.

Examples of float variable definitions

```
float salary ;
float max_debit= 1260.0 ; // floating point constant must have a decimal point
float million = 1.0e6 ; // the e in the constant value indicates the exponent of 10
```

`double` Double precision floating point Data Type

This type is just another floating point type with all the features of the one previously described. However it is bigger (8-byte) and thus it can declare a variable which will contain values with more significant digits (approximately 15).

1.3.4 More about constants

Numeric constants were presented in the previous section because the timing was good since examples with such constants were already used and some explanation had to be given to the reader. Still, there is a lot more about constants because

- numeric constants are not the only constants
- C language handles in a very complicated manner the so called string constants

A first definition was given for character constants and string constants in paragraph "Binary integer types" before. Let's talk a little more about these.

Character constants

The notation is simple : an ASCII character enclosed in single quotes like `'a'`,`'$'`, `'R'` etc. These expressions represent the value that the ASCII character has. For example the letter Q in the ASCII table has the value 51 hex (or 81 decimal)

This means that given the definition

```
char w ;
```

the statement

```
w = 'Q';
```

is identical with

```
w = 0x51 ;   // use a hex constant
```

and also with

```
w = 81 ;   // use a decimal constant
```

It is <u>up to us</u> (humans) to use whichever of the above we like, in order to better document our program (make it readable).

It is time that you start seeing advantages that a language *"not strongly typed"* has :

```
w =  'Q' + 1 ;   // add two numbers
```

No conversion from type to type is needed in this example. Directly we have added a number to a character constant. It is as if the statement read :

```
w =  'R' ;
```

String constants

And now the hard part. In my opinion understanding string constants is the first big step in learning C (the second step comes later with pointers).

First the notation : It has been already mentioned that a string constant is a piece of text enclosed inside two <u>double</u> quotes. Example :

```
"Press a key to continue"
```

So far, all you have to remember is not to use single quotes for strings or double quotes for character constants.

Also it must be mentioned that this very same notation is used for initialising strings. Nevertheless this does not change the following discussion at all. The rules and behaviour is the same. Perhaps it is too early to extend our discussion on initialising strings. You can find references to this topic at the end of Lesson 7 and throughout the whole Lesson 11

Before we continue, here is a (long) comment

Programming languages are supposed to be human oriented and not machine oriented. This rule does not only apply to the linguistic style used for instance writing

```
IF  X > Y THEN DISPLAY "I am Sorry"
```

instead of

```
cmp ESI , [04518290]
jnc   EDI
push  msg023
call  displ
```

but also refers to the concepts that the (English looking) code describes. For example

```
WRITE PRINTER  FROM MESSAGE AFTER ADVANCING 1 LINE
```

looks more comprehensive than

```
prtObject.Print vbCrLf ; Message
```

For this purpose, some languages "hide" the concepts that are in reality understood by the computer in order to make the source code more friendly to the human. In this case the compiler undertakes the task of translating to the proper computer sequences that must be executed.

C language does **not** do this, instead it addresses the experienced programmers who know the "secrets" of computer processing and although it is English looking it uses concepts that are **machine** oriented.

In the case of strings – and more specifically string constants - C does not accept strings as data units with a data type, because they consist of more than one elements. For instance the value "Sorry" is said in most languages to be a string – **one** string. But the truth is that this is a block of memory containing more that one things ; it contains an 'S' an 'o' , an 'r' , a second 'r' and a 'y'.

Whether you like it or not, that's how the computer 'sees' it. Kernighan and Ritchie the authors of C language say that since this is the truth we must **not** hide it. Of course one reason for not hiding it is to make use of the powerful concepts that a computer can apply to improve performance.

Generally **C doesn't support composite data types** and the reason is that computers don't support such types. When you instruct the computer to display "Hello" it is not like saying *"take the Hello and display it"* because the processor cannot take a whole string at once (it doesn't fit in the CPU). You must give the letters one by one to the processor and because you would hate to do this, there's a better way :

- Store the string in memory,
- give **the address** of the string to the program and
- let it pick one by one the characters and use them

Back to String constants. A string constant like "Hello" is of course stored somewhere in memory but the new thing (a little confusing in the beginning) is that in the program it represents **the address** of itself. So when we write

```
printf ("Hello") ;
```

we give to the function printf( ) the address of Hello and not the text.

It is as if we say *"go to that memory location and print the text that is stored there".*

At this point let's introduce a fundamental I/O function that is 'hidden' behind printf ():

The **putchar(char)** function says : "Give my *one character* and I will display it on the screen". Compare this to the previous statement that involves an address. Using putchar( ) we could display the "Hello" equally well as with printf( )

```
putchar('H') ;  putchar('e');  putchar('l'); putchar('l'); putchar('o');
```

[Imagine what would happen if we had to write such statements just to display a string!]

Now we 'tell' the computer to take one by one the letters we give it with separate statements while with printf("Hello") we instruct the computer to access the memory address where Hello is stored and pick up the characters one by one.

1.3.5 The string terminator

The above raises the second matter to be settled : How does the computer know where to stop picking letters like 'H', 'e' etc. ? It doesn't know. That's why we have to put a sign – a terminator. All strings are terminated with the **Null character** and you can read about it in the next paragraph.

Conclusion

- **'H' is not the same as "H"**

'H' is a character constant
"H" is a string constant

- **All string constants are enclosed in double quotes like "Hello".**
- **A string constant in the program code is treated by the C compiler as having a Null terminator**

See next section (Scalars and aggregates) also.

1.3.6 Escape sequences

In the beginning of this lesson is a string constant "`My First Program!\n`" containing the character sequence `\n` .

A sequence of a \ followed by a character is called an escape sequence and though this is confusing (since there is a key on the keyboard call 'Escape' and of course this key corresponds to the Escape character of the ASCII table) it is easily accepted by the C student since these sequences are in everyday usage in C and secondly the word sequence tells the difference.

An escape sequence gives the C programmer the ability to include Control characters in string constants. A control character is not printable, it simply controls the behaviour of the output device. For instance if I want to send a phrase to a printer and then before the next phrase I wish to start a new line how do I instruct my printer to start a new line ? With a control code. The ASCII table at the end of this document is very informative.

The ASCII code which contains 32 control codes and has been around from the early years of computers and that is why it has handled the problem of starting a new line just like the old typewriters.

To start a new line with a typewriter you have to

a. return the carriage (cylinder where the paper is wrapped) back
b. feed a line : push the paper up for the size of one line

This is done in one step because when pushing the carriage from the left to the right, there is a button that is pushed also when the carriage stops and feeds a new line. As you can see in the ASCII table there is a Carriage Return (or simply Return) character and a Line Feed .
The question with computers was :
Must we contain both a Carriage Return (CR) and a Line Feed (LF) in a text block in order to cause a new line at the output device ?
UNIX (and Linux) have adopted the convention that if a Line Feed is there then the operating system will add a Carriage Return. In MS-DOS and Windows both characters are included. But from the keyboard where the Enter key sends a (Carriage) Return to the computer it works the other way : a return is sent and the LF is added by the operating system.
Since C language is the sibling of UNIX , it followed the former method : a Line Feed in the text is enough to make the operating system send both control codes to the output device.

Here are some popular escape sequences

\n	newline	[LineFeed]
\t	tab	[Tab]
\r	return	[Carriage Return]
\\	\	
\b	backspace	[backspace]
\0	Null character	

A \n in a string will cause the start of a new line or as we usually say we have an auto CR behaviour. See the examples of page 20 where it is obvious that a \n is all you need to start a new line and that you can have as many of them in a single string.

1.4. Scalars and aggregates

General

The word 'scalar' was borrowed from Physics where a scalar is a measure with no dimension or direction i.e. it doesn't have a direction - it is not a vector. The same definition can be used in C language with the difference that the words **dimension** and **direction** are used with a different meaning.
An array is a vector in the sense that it doesn't have only one measure(one value). For example the area of a surface may be expressed with two figures to denote the width and length from which the area is calculated. This could be written in some other language as

area = Array (12.3 , 3.26)

In C, arrays are called **aggregates.** An aggregate is therefore a variable with multiple values. It can also have multiple dimensions. A variable containing *only one value* is not an aggregate but it is a **scalar.**

Previously in this page definitions like these

```
short int myvalue = 6;

short int counters[6];
```

defined scalars and aggregates. Variable `myvalue` is a scalar and variable `counters` is an aggregate.

How do I know? the definition of counters has a pair of square brackets following the name of the variable.

What about multidimension aggregates ? The above variable `counters` is a one dimension aggregate: It contains only one row which can contain 6 values. Let's see the syntax for multi-dimension definitions in C

```
char grade[12][26] ;

int y[100][300][140] ;
```

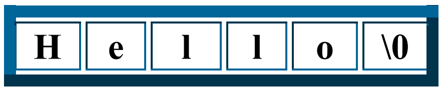
The above are definitions of a 2D character aggregate and a 3D integer aggregate. In the example of the grade variable , this contains 12 rows of 26 **elements** each.

The 0 base headache

In C the first element of an aggregate for example x[12] is x[0]. Obviously the last element is x[11]. One must be very careful because these subfixes confuse the newcomer. And the situation becomes more desperate when we learn that ***a string is stored in a character aggregate*** since there is no data type in C called *string* or so.
And also that a string stored in a char aggregate is ***null terminated*** with the Null character which is written in C like this: '\0' (read the note below).

char message[6]= "Hello" ;

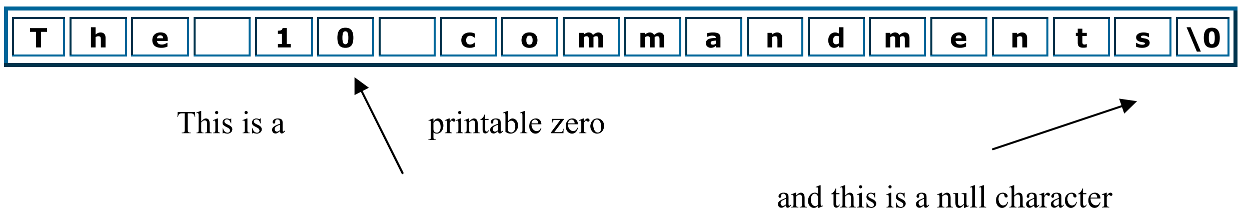
is stored in memory like this



This means that

- To store the 5 byte string value "Hello" we must allocate 6 bytes
- The Null character is implied to be there in string literal definitions like "Hello"
- The last (printable) character of the 6 byte string message[ ] is in message[4]

Note : \0 is one character and is written like this to distinguish from the printable 0 which as you can see in the ASCII table corresponds to the value 48 (30 hex) and not to 0 .



Suppose that this string had to be the initial value of a string (i.e. a character aggregate).

QU : Do I have to count the characters in order to declare the size of the string ?

ANS: No !

Here is the way around it :

```
char  title[] = "The 10 commandments" ;
```

Empty brackets are introduced here for the first time. Later you will find a paragraph (Lesson 7) discussing empty brackets which happen to have other usages and meanings also.

In this case (definition and initialisation of a string) the empty brackets "say" to the compiler :

"Find yourself how may bytes are needed to store this string value"

Common sense says that a definition like :

```
char nuts[] ;
```

is impossible because how will the compiler find the required size when there is no initial value !

1.5 Operators

1.5.1 Some basic operators

Arithmetic

As expected the four operations of arithmetic are treated in C in the same manner as in any other language and there hierarchy is also the usual. An example is "a thousand words" :

```
        x = y + 3 * ( a+b )
```

So let's mention the things that are not expected mainly because they are unique in C :

The modulus operator Let's add a fifth operation (binary operator) to the four well known operations , The modulus in C can be found using the symbol %. An example

```
        a =  x % 3 ;
```

Assignments

It is not strange that the = sign is used for assignments as in this statement

```
        a = b ;
```

What is strange at first sight is that **the = sign is only the assignment sign and can't be used for comparisons** (read more in Lesson 3)

The other new thing is that you can write multiple assignment statements. In the following example

```
        a = b = c = d = -1 ;
```

the statement says *"assign -1 to a, b, c and d"*

Also, specific to C language is the combination of an operation with the assignment = symbol. The statement

```
        x + = 5 ;
```

says ***"add 5 to x"*** . In another language you would write this as `x = x + 5`
One more example :

```
        x * = y ;
```

says ***"multiply x by y"*** .

Comparisons

Perhaps you already have the curiosity : how do we compare for equality since the = sign can't be used? The answer is simple : we use two = signs.

Example :

```
        if  (  a == b ) . . .
```

Later it will be explained that conditions in C are always enclosed in parentheses.

The **NOT operator** in C is the exclamation character (!) and thus the not equal is !=
Example:

```
        if  (  a !=b ) . . .
```

The following simple program will help you understand the difference between the assignment = and the comparison ==

```
#include <stdio.h>

int main()
{
   int x , y ;
        x = 0;          // ASSIGNMENT
        y = 0;          // ASSIGNMENT

        if (x==0)       // COMPARISON
          printf("x is = 0\n");

        if (y=0)                // 0 is assigned to y and  y is tested
          printf("y is = 0");   // INCORRECT CONCLUSION
        else
          printf("y is not =0 ; it is =%d\n", y);

        x = 3 ;
        if (x != 0)
          printf("x is not =0 ; it is =%d\n", x);
        y = 4 ;
        if (y)
          printf("y is not =0 ; it is =%d\n", y);

        return 0;
}
```

The rest is the same as in all other languages

```
        >    >=    <    <=
```

are the symbols you will use for what they seem to be used for.

Logical operator that can be used in conditions are

&&	the AND operator
||	the OR operator

example

```
        if  (a>0  &&  b<a) . . .
        if  (x==y  ||  y>10) . . .
```

1.5.2 Increment and Decrement Operators

The increment and decrement operators **++** and **--** respectively increase or decrease an lvalue (i.e. a value that can be modified -see lesson 3) by 1.
In the special case of pointers the pointer is increased or decreased by the **size** of the pointed element. You will better understand this, when you reach section 2.3 .

Prefix - Postfix

```
++v  First increase v then use the value of v
v++  First use the value of v then increase v
```

The above definitions do not clarify what exactly the verb "use" means. The following cases where the pre fix usage of the increment decrement operators has a different application that the post fix, will try to make it clear.

All examples following use the ++ operator. Similar are the usages of the -- operator.

Case I: Assignments a=++b ; vs. a=b++;

```
// First increase
b= 3;
a=++b ;
 → b is =4 and a is also = 4
            - - - - - - - - -
// First assign

b=3;
a=b++;
 → b is =4 but a is = 3
```

Case II: Indices a[++i] vs. a[i++]

```
// First increase
     char s[]="abcdefg";
     char c;
     int i;
i= 0;
c= s[++i];
→ i is = 1  and c = 'b'
            - - - - - - - - -
// First assign
char s[]="abcdefg";
char c;
     int i;
i= 0;
c= s[i++];
→ i is = 1  and c is = 'a'
```

Case III : Conditions if (... ++x ...) vs. if (... x++...)

```
i=0;
if (i++ == 1)           // first compare then increase
   printf("Hello");     // not displayed
printf("i=%d" , i);
- - - - - - - - -
i=0;
if (++i == 1)           // first increase then compare
  printf("Hello");      // it is displayed
```

Countdown : Two very special examples	
`n=10 ;` `while (n--)` `putchar('*');  // 10 asterisks displayed`	To repeat a piece of code a fixed number of times , it is better to do it counting down that up. Adding to this the advantage of C in using the result in the condition parenthesis as a true false value, the code at the left is a great moment for C.
`n=10 ;` `while (--n)` `putchar('*'); // 9 asterisks displayed`	

Case IV : Passing parameters func(++a) vs. func(a++)

```
i = 1;
printf ("i=%d", i++);  // OUTPUT: i=1
- - - - - - - - -
i = 1;
printf ("i=%d", ++i);  // OUTPUT: i=2
```

Case V : Pointers (indirection involved)

Pointers will be introduced in the next chapter.

This special case which is more complicated thanks to involvement of *indirection* also to be introduced in next chapter.

So let's leave this for later. You can find it in section 2.3.

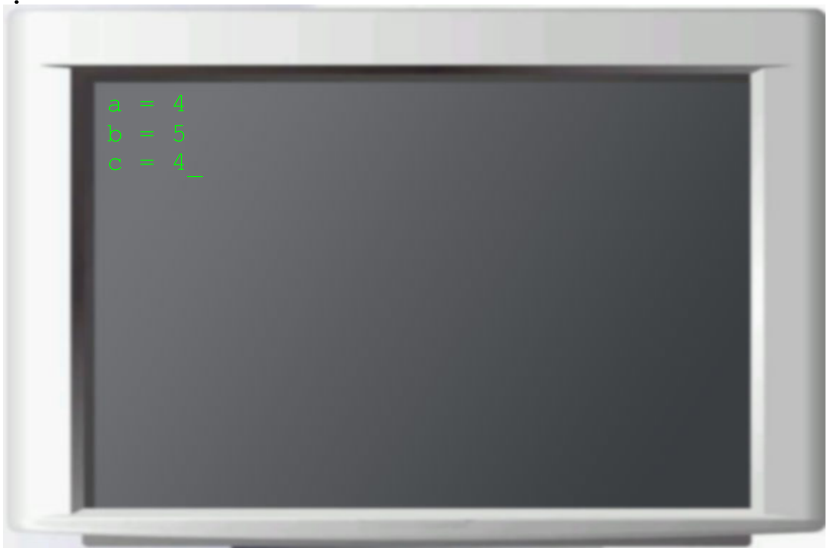
Example 5.1

```
void            // Fill a string with asterisks
fill_stars(char *s)
{
      while (*s!='\0')
       *s++='*' ; // First assign the asterisk then increase the pointer
}
```

Example 5.2

```
// Demo of the above case 1 [assignments]
void     main( )
{  int a , b , c;
      a = 3 ;
      b = ++a ;  // First increase a then assign to b
      c = b++ ;  // First assign b to c then increase b
      printf("a = %d\n" , a);
      printf("b = %d\n" , b);
      printf("c = %d\n" , c);
}
```

OUTPUT :



```
a = 4
b = 5
c = 4_
```

C Language / Lesson 2

2. Understanding Pointers and Addresses

When the finger points at the moon, the idiot looks at the finger [Confucius]

2.1 The address operator

The address of a block of memory or a variable in memory is important to very powerful languages like C (in time you will realize the reasons why). The address of a block is the address of the first byte of the block. For instance if "Hello" is somewhere in memory then its address is the address of 'H' . Variables or expressions which refer to **a block of memory** by default return their address in C. On the contrary those variables containing one single value -called **scalars -** or elementary expressions referencing one single value by default return the value What if we want to have the address of a scalar ? Herein we introduce the **address operator** . It is the character **&** (ampersand) preceding an expression. Be careful :

x = &a	This is an address operator
y = k & m	This is **not** an address operator

Using the **&** with Scalars and Aggregates

Given are :
```
     int n;      // a scalar integer
     char s[20]; // a character aggregate
```

	the value of	the address of
Scalar	n	&n
Aggregate	s[n] [1]	s

1. the value of the nth element of s

```
// A sample program

   int x = 147 ;              //  definition  of a scalar
   char s[] = "Demo";         // definition of an aggregate

   void main ( )
   {
      printf ("x=%d\n" , x );
      printf ("the address of x is %d\n" ,  &x );
      printf ("the address of s is %d\n",  s );       // the & is not required before s
   }
```

2.2 Pointers

2.2.1 Why pointers

Regarding address processing, there are two disadvantages that pointers came to solve

Disadvantage 1 :

An address is a binary number of size equal to the size of the address bus. This is not the place to discuss what the address bus is however for those who don't know it : For every transfer of a value from memory to CPU or CPU to memory , the processor has to present the address which is involved in the operation. This is done on the special interface of the processor called "the address bus" . It is easily understood that in order for an address to be "presented" on the address bus it must not have a size bigger that the bus.
The problem here is that the size of the address bus in not known in advance when we write a program but even if it is , what will happen if our program is used with a computer with a smaller or bigger address bus !

Disadvantage 2 :

When we store an address in a variable for later usage, we don't care for the address itself but for what is stored at that address. So the question is how are we going to "tell" our computer to use the value at that address an not the address itself ?

These two problems above are solved with the introduction of a special type of variable the pointer and a new operator the address operator .

2.2.2 Pointers and indirection

Pointers are variables like all other variables. They can be scalars or aggregates (like all other variables) and have all the characteristics of a variable. Only they contain addresses. And following the discussion of the previous paragraph they have a size equal to that of the address bus. This difference would not be so interesting if an address (in memory) wasn't the address of another variable. This second variable (the *pointed* variable) has a value and this is exactly what we are usually interested in. Thus we have a value (in the pointer variable) which leads to another value (the pointed value). This is called **indirection**.

Result : While all other variables have an address (where they are stored in memory) and a value , pointers in addition to these also have **a value pointed by the pointer**. That is why knowing that a variable is a pointer is not enough. We must declare what is the type of the pointed value. Obviously this is the reason why pointer are declared with the type of the pointed value followed by an asterisk. The asterisk says "this is a pointer" and the data type before it declares that "this pointer points to (contains the address of) a value of this type.

Examples

```
long *plng ;
char *current_message ;
```

int *n[12] ; // This is an aggregate (array) of 12 pointers and not a pointer to an aggregate
char msg[ ]= "Sample message" ; // this isn't a pointer - it is a string (character aggregate)
char *p = msg ;   // This pointer is initialised to the address of the previous string

In the previous example the expression **msg** represents the address of variable msg because, as explained earlier, msg is a block of memory containing more than one values .

Note : In C a string is not one single value but a composite variable (consisting of more than one values) - every character in the string is a value and thus the string is an aggregate of characters.

char max_files ; // This is only a one byte variable -a scalar character variable

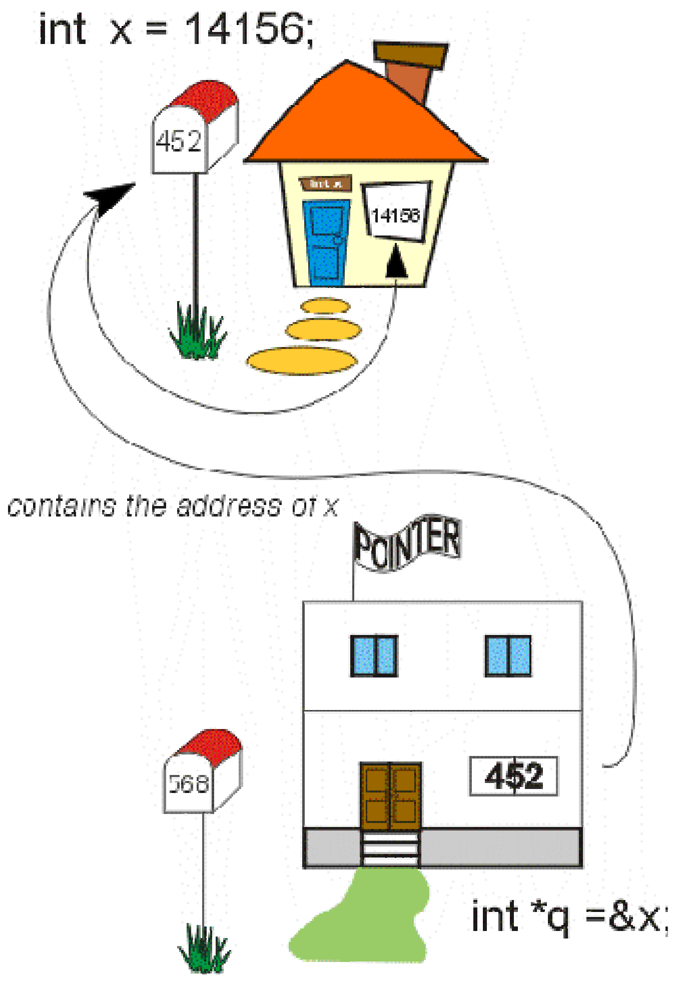
char *p = &max_files ; // This pointer is initialised to the address of max_files

Note : This time max_files is one single value and if we didn't use the address operator, the value of max_files would initialise the pointer and not the address.

The following example attempts to describe this situation graphically .

Two variables are defined :

int x and int *q



&x == 452   &q == 568

q is initialised (=&x) to the address of x , i.e. q == 452 (this is the value in q) The expression *q means the contents of the integer (since q is an int pointer) , which is stored at the address pointed by q i.e. the values of q. q==452 => at 452 is the 'home' of x and the contents are 14156. Therefore, the expression *q is == 14156. This addressing mode is called **indirection** because the access to 14156 was indirect through 452.

Now that pointers have been introduced, let's enrich the table of section 2.1 :

Given are
```
  int n;
  char s[20];
  char *p;
```

	the value of	the address of	indirection
Scalar	n	&n	▨
Aggregate	s[n] [1]	s	▨
Pointer	p	&p	*p

2.2.3 More on pointers and indirection

Given are :

```
char salut[]="Hello";       // a string
char *p=salut;              // a pointer pointing to salut[]
```

1. **salut[2]** is <u>the value</u> of the 2nd (3rd) character of salut
2. the same value can be accessed via **p** using the indirection operator ➔ ***(p+2)**

   B U T
3. there is a better way ➔ **p[2]**
   meaning : the value of the 2nd element of the string pointed by p

In other words although expressions with scalars do not contain square brackets, a scalar pointer is an exception because in this case the square brackets do not imply an element of this variable but an element of the pointed aggregate.

2.3 The increment and decrement operators revisited (see Lesson 1)

Continuing the discussion of page 20 about "first increasing/decreasing then using" or "first using then increasing/decreasing, now that pointers have been introduced, one more case is presented

Expression	First	Then
*p++	use value	increase pointer
*++p	increase pointer	use value
++*p	increase value	use value
(*p)++	use value	increase value

the last statement adds 4 to p because p points to a long value and long means 4 byte size.
For instance given is

```
long *p;
. . .
. . .
  ++p;
```

the last statement adds 4 to p because p points to a long value and long means 4 byte size.

Sample function using indirection and increment operators

```
short
strlen(char *s)
{ short i;
   i=0;
   while (*s++)   // use the value pointed by s for the condition and then increase s
      ++i;
   return i;
}
```

In Lesson 4 you will have the opportunity to see common usages of pointers

2.4 Formatted output to the screen - Keyboard input

Now that we know a few things, can we write a small program ?

Yes and No. Because usually even a very simple program requires console input/output i.e. we must be able to input data from the keyboard and display results on the screen (at least).

Officially input/output commands do not exist in C in the sense that they are not built-in. We must use the C standard library which contains a lot of functions to process console input and output. These functions are very powerful and flexible and I should say very impressive. The principles related to them are explained later (mainly in lesson 8). On the other hand we can't wait until we become gurus in C before we write a simple program to read two numbers from the keyboard and display their sum on the screen.
The following program uses the most important functions for console I/O and will give us the opportunity to start using them without asking questions.

printf( ) was already used in the very beginning of this text. It was used to simple display a message like "Hello world" or so. This function however is a very rich one in capabilities and doesn't only display string constants. It's real definition is approximately like this

```
printf(formatstring , arg1, arg2 , . . . )
```

where the first passing parameter is a string containing the format of the output : explaining how the arguments must be printed,
A format string contains text and (optionally) one or more **%** symbols called **format modifiers** followed by a letter explaining how the next in order argument (pass.parameter must be printed)
This means that the number of arguments following the format string in the parenthesis must be equal to the number of % symbols in the formatstring.

Example :

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

There are 3 arguments (a,b, and c) and therefore there are 3 % format modifiers in the format string.

What do the letters next to the % symbol mean ?
Here are some letter codes which when following the % give useful combinations (you can find a detailed list in the K & R book) :

code	type	format
d	int	decimal number
o	int	octal number
x or X	int	hexadecimal number
u	unsigned	decimal number
c	char	single character
s	char pointer	string
f	float	floating point number
g	float	- " -

```
// A simple example
#include <stdio.h>
char myname[7]= "Yannis" ;
void main()
{printf("My name is %s\n" , myname);
}
```

Format modifiers will be used in the following program.

gets(*string* **)** can be used for text input **BUT be careful !!!!!** You can't use this function to input numbers to numeric variables. gets( ) must always have a string as a passing parameter and because strings are stored in character aggregates (as explained before in paragraph "The 0 base headache"), we can rephrase this to " *gets() must have a char.aggregate as a passing parameter*"

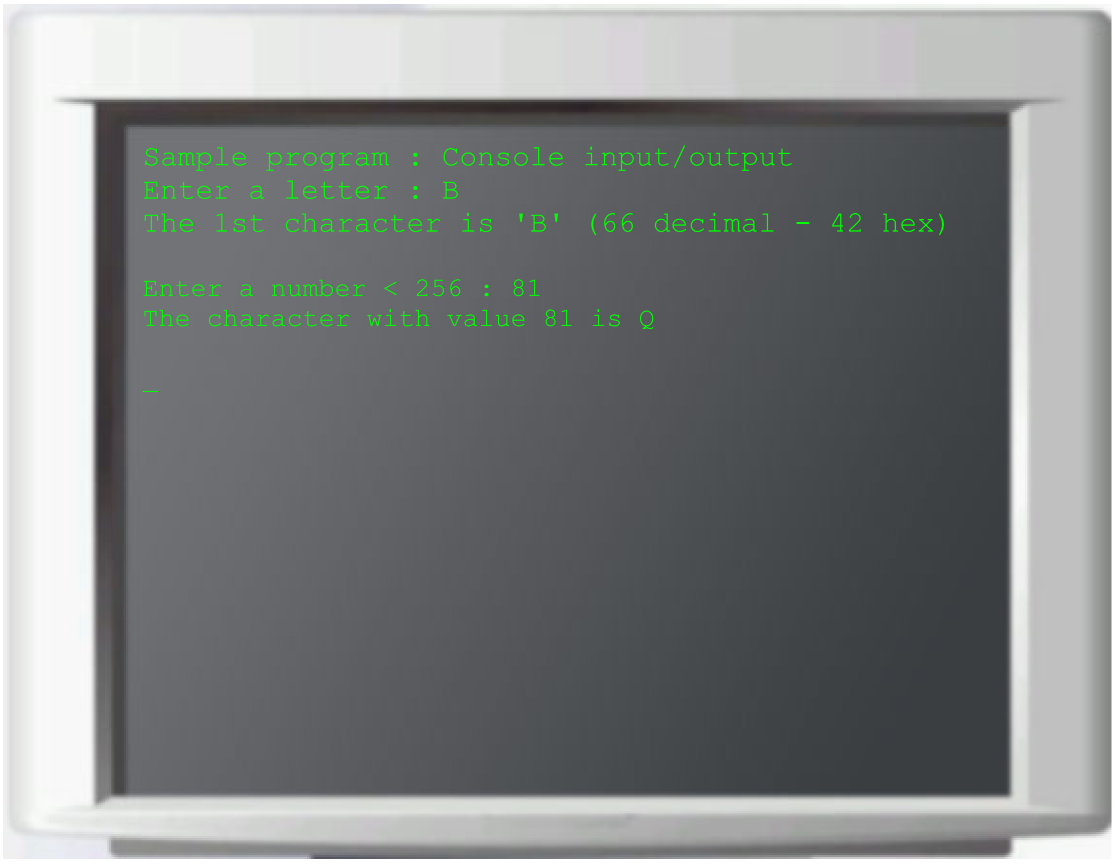
Finally **scanf(. . .)** can be used to input numbers. Don't ask much , from day to day it will look more familiar to you. For the time use it as in the example .

```c
// C language sample program
// Console I/O functions

#include <conio.h>
#include <stdio.h>
#include <stdlib.h>

//// Variable definitions ////
char c;
char buff[60];
int n;

/////// Main Program //////////
void main()
{   printf("Sample program : Console input/output\n\n");
    printf("Enter a letter : ");
    gets(buff);
    if (buff[0]=='\0') // If the 1st character of buff is NULL
     exit(0); // then the string is empty
    c=buff[0];
    printf("The 1st character is '%c' (%d decimal - %x hex) \n\n", c,c,c);
    printf("Enter a number < 256 : ");
    scanf("%d", &n); // Use scanf to input numbers
    if (n>255)
     printf("Out of range !\n");
    else
     printf("The character with value %d is %c\n\n", n,n);
}
```



```
Sample program : Console input/output
Enter a letter : B
The 1st character is 'B' (66 decimal - 42 hex)

Enter a number < 256 : 81
The character with value 81 is Q


_
```

See also : The ASCII code

Exercise 2.1

What is the output of the following program ?

```
#include <stdio.h>
char c = 'C' ;
void main()
{       printf("%c%c ROM" , c , c+1 );
}
```

Exercise 2.2

Study the previous sample program and write a program to

- ask the user to enter an upper case (capital) letter
- check that the character is an upper case one (use statements like if (c>='A' && c<='Z'))
- taking a look at the ASCII table you can easily see that lower case letters are 32 positions after their corresponding upper case characters and therefore given an upper case character we can find it lower case pair by adding 32
- display the lower case version of the upper case character typed by the user

C Language / Lesson 3

3. Lvalues / The Assignment =

In C values are assigned to variables or other elements using the = sign exactly as in most languages
Example :

```
 int x ;
   x = 15 ;
```

But unlike most other languages the = sign is **not** used for other purposes like comparisons.
For instance in Basic A=B is an assignment but IF A=B is a comparison (A and B are compared for equality)

In C

a. if we want to compare **a** and **b** , then the condition will look like this

```
    if ( a == b )  ....
```

b. if on purpose or accidentally we write

```
    if ( a = b )  ...
```

then b will be assigned to a and the truth of the condition will be the value of **a.**
(Remember that a number obtained as a logical expression ***is always true unless it is =0***)

Although there is a mysterious confusion in several C language books regarding the definition of an lvalue we can define it easily with two similar statements.

a. an expression which is allowed to appear at the left side of an assignment = is an **lvalue**
b. an expression describing a place somewhere in memory where we can store a value is an **lvalue** .

Both definitions are trying to say the same thing, since a single = sign in C always means an assignment.

Some reasons why the lvalue definition is important in C :

FIRST : The address operator (&) gives the ability for expressing addresses in addition to values. But an
address cannot change. You can change the value at an address in memory but the address is not modifiable. Thus if we attempt to add the following line in a C program

```
            &x  =  7  ;
```

where x is a variable previously defined in the program
then we will get a compilation error : **Not an lvalue**
Of course &x is not somewhere to assign a value. If we wanted to assign a value to x then

x = 7 ;

is what we need.

SECOND : Aggregates (arrays) in C are composite types i.e. they don't receive values with a simple assignment (see lesson 1).
Let's mention the case of strings which can be assigned a value with the use of the **strcpy( )** function example

```
    strcpy ( name , "George") ;
```

An aggregate's name in a statement means the address of the aggregate and not its value. That's why a statement like this

```
      name = "George" ;
```

where name is defined as a character aggregate (string) will cause a compilation error : **Not an lvalue**

What else ?

Although in traditional C the lvalue issue is raised usually in cases similar to the above 'FIRST' example i.e. when the programmer accidentally attempts to store to a destination that is an address, which is logically impossible, a few more cases of **Not lvalues** will be presented here :

Constants are NOT lvalues. This is of course obvious : A constant is by definition something that doesn't change. Trying to assign a value to the constant means "change the constant". That is , the computer is instructed to change an expression that can't change. Do you want to try it? Then enter this statement in your program :

```
int  x;
     5 = x ;
```

Of course this is obvious. Whenever I ask an inexperienced young programmer with no knowledge of C language to discuss this, I always get the answer "this is wrong!". But then, when I ask about this :

```
int  x;
     if (5 = x) ....
```

most people say "this is OK !". Well it's not OK in C , because this condition doesn't compare 5 and x for equality. Instead due to the assignment operator it attempts to assign x to 5 which as explained is crazy.

More composite cases :

The advantage of C from the usage of two different operators (the = for assignments and the == for comparisons) allows for sophisticated code like this

```
int  x, y ;
      . . .
     if (x=y==5)
     { // this block is executed if the condition is true
         ...
     }
```

This statement 'says' : Assign the logical result of the comparison 'y==5' to x and then use it as a condition (i.e. if it is true proceed to the execution of the flow control block that follows) . The reason it is interpreted like that is that the = sign has the lowest priority of all operators and so the == must be executed first.

The above style of a condition is legal but not really useful. There is a slight variation though, which is really useful and very popular to C programmers

```
int  x, y ;
      . . .
    if ( (x=y)==5)
    {    ...
    }
```

The parenthesis around the `x=y` makes all the difference. Now the priority goes to the = operator and the statement says : "Assign the value of y to x and then compare x and 5 for equality". In other words we have a ***2 in 1*** statement since in another language this would be done in two steps

```
x=y; if (x==5) etc ...
```

Lvalues that are ... Not lvalues

As it was shown in the previous paragraph an expression is an lvalue if it is physically and logically possible to assign a value to it. The obvious was demonstrated above : numeric constants is impossible to change. But what about **string constants ?**

In the following example

```
 . . .
 char *p;
      p="Hello" ;
```

the "Hello" is a string constant. However this is stored somewhere in memory as it is shown in Lesson 1 (scalars and aggregates) and therefore can be physically overwritten. For instance if this statement

```
      strcpy(p,"Bye") ;
```

followed the above code, then the "Hello" would be overwritten.
Result : string constants are modifiable but **we don't want this to be so** (for various serious reasons that are not explained here).
That's why the rules of C language dictate to a C compiler not to accept the modification of a string constant. This means that

- ***string constants are <u>not</u> lvalues***

C Language / Lesson 4

4. Controlling the flow of a program

The following code :

```
char *salutation, *title, gender;
 . . .
if (gender == 'F')
{ salutation = "Madame";
  title= "Ms." ;
}
else
{ salutation = "Sir" ;
  title = "Mr.";
}
```

speaks for its self. If you ask someone with no knowledge of C language, you will get an explanation of what is the purpose of the above code. However there are details that one would not notice for sure if there were no reason for learning how to write blocks of this type. So let's start :

- Conditions are always written inside parentheses
- Following the parenthesis of a condition in an `if` or `while` statement there is NO semicolon
- the block following the condition (and the `else` statement) is enclosed in braces. Such a block can never be confused with a function body because it is nested <u>inside</u> a function body (read again lesson 1)
- if the block contains only one statement then the braces may be omitted as in the following variation

```
char *salutation, *title, gender ;
if (gender == 'F')
   salutation = "Madame";
else
   salutation = "Sir" ;
```

```
// Sample program
#include <stdio.h>
char your_name[40] ;
void main()
{ int n ;
      printf("Enter your name : ");
      gets(your_name);
      n=strlen(your_name);
      if (n>1)
      { printf("Hello %s\n", your_name);
        printf("Length of name=%d\n", n ) ;
      }
      else
        printf("Unacceptable answer !\n");
}
```

return

This statement, although familiar as a command requires special treatment in the way it is used in code. First of all, let's remember that to return we have to call first :
Calling a block of code generally means by definition that the code "knows" how to return back where the block was "called", in other words calling means jumping to another point of the code while "remembering" the address of the next instruction after the 'call' .
In C language , calls are only done to functions. You can call a function by simply mentioning its name followed by a pair of parentheses containing the required passing parameters. In the following example:

```
void say_hello()
{    printf("Hello\n");
}
void main()
{
     say_hello();
     printf("End\n");
}
```

the statement `say_hello();` **calls** the function say_hello(), that is it branches to the function (and executes the statements in the function's body) remembering that at the function's end it must **return** to the next statement after the call [printf("End\n");] .

Now it's time to present the variations of the return statement.

- It has been mentioned in Lesson 1 that a function is **void** if it returns a meaningless value. Some novice programmers would say that the function returns nothing. This is not correct because there is always an agreed place in the CPU where a function is expected to 'put' a return value before it returns. The functions that don't leave a value in that place (because they have no reason for doing so) are called void.
- It has been clear in all previous examples that a function terminates its execution (returns) automatically when it reaches the end of its block.

Based on these two remarks, we can easily tell that a void function will return at the end of the block. But what about a function which returns a result ? There is a variation of the syntax for a function returning a value :

```
          return  value ;
```

Later in this lesson you will have the chance to see examples of functions returning a value.
Conclusion :

- A function returning a return value must always use the `return` statement

There is however another reason for using the `return` statement . Functions (void or not) do not return always at the end of the function body. The need for returning pre-maturely from a function is very frequent and in such a case the return statement must be used for any type of function.

Example : Returning from a void function

```
     if (condition)
          return;
```

This time the return statement did not have an operant next to it.
Conclusion :

- The `return` statement is used without an operant only if the function is `void` and must return before the end of the body.

while

Similar (or I should say identical) to those of the if statement are the rules for the while statement. An example is all you need

```
void display(char *s)
{     while (*s)                  // while the pointed character is not null
      {     putchar (*s);         // display the (pointed) character
            ++s;                  // and increase the pointer
      }
}
```

Note : the ++ operator is explained in the following lesson.

```
// Example II
#include <stdio.h>
char m[]="\tDemo:\tControl characters\n" ;
void main()
{ int i=0 ;
      while (m[i] )
      {   if (m[i]<' ')   // if the ith char is a control char
              putchar('*');  // then display an asterisk
          else
              putchar(m[i]) ;
          ++i ;
      }
}
```

OUTPUT

```
*Demo:*Control characters*
```

do - while

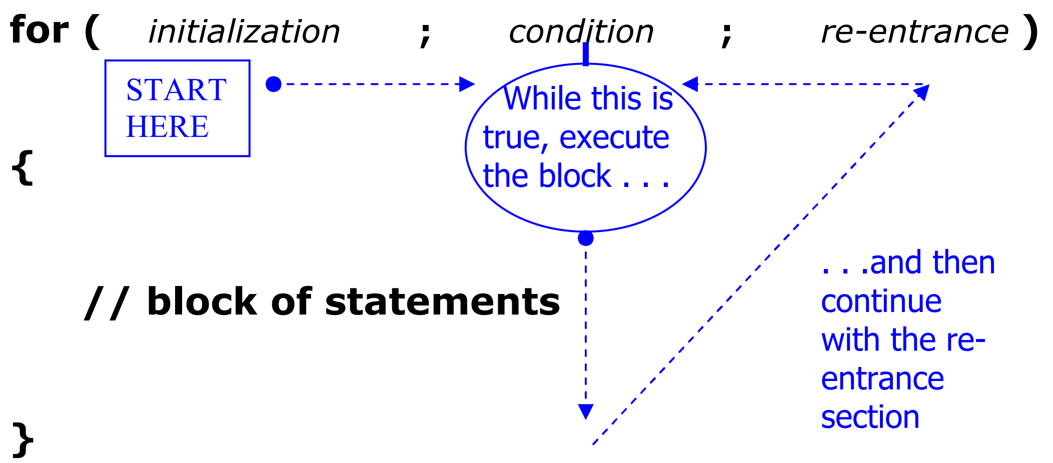
Here we have to be careful. Look at this do – while loop :

```
char c ;
printf ("Press Enter");
do
    c = getch();        // get a character from keyboard
while ( c!='\r') ;      // keep looping while the character is not a Carriage Return
```

the `do` <u>has no semicolon</u> while the condition parenthesis **IS** followed by a semicolon.
Generally you have to be very careful with semicolons, because incorrect usage does not always generate an error (which would at least be an alarm in case of mistyping).
There are cases where even a single semicolon is accepted with the meaning 'do nothing'. Such a semicolon is called **"a null statement"** and later we will use some of these.

The **for** statement

The parenthesis in a **for** statement has three sections separated with **;** .
The diagram below explains how these sections are involved in the execution of the loop


for (*initialization* ; *condition* ; *re-entrance*)
START HERE
While this is true, execute the block . . .
{
// block of statements
}
. . .and then continue with the re-entrance section

```
// Example II re-written  using for (...)
#include <stdio.h>
char m[]="\tDemo:\tControl characters\n" ;
void main()
{ int i ;
      for (i=0 ; m[i] ; ++i  )
      {   if (m[i]<' ')   // if the ith char is a control char
              putchar('*');  // then display an asterisk
          else
              putchar(m[i]) ;
      }
}
```

A function written in three different ways

strlen(char *s) returns the length (in characters) of a string

Using an index

```
int
strlen(char s[])
{ int count;
     count=0 ;
     while (s[count])
        ++count;
     return count;
}
```

Using pointer s

```
int
strlen(char *s)
{ int count;
     count=0;
     while (*s)
     {   ++count;
         ++s ;
     }
     return count;
}
```

Using for(. . .)

```
int
strlen(char *s)
{ int count;
      for (count=0; *s ; ++s)
         ++count;
      return count;
}
```

Note 1 : the notation **char s[]** in a passing parameter only , is the same as **char *s** : the former means "a string" and the latter "a pointer to a string", since passing a string to a function is done by passing its address (a pointer) these are therefore identical.

Special cases of **for** blocks

You can omit the statement of any section in the `for` parenthesis. The initialisation and re-entrance sections when empty are simply not executed. An empty condition section declares a condition which is always true.
In such a case there must be an other way out of the loop. This is the statement **break;** which terminates the execution of the loop and continues with the first statement following the loop.

```
for ( ; ; )
{ . . . .
      if (condition)
        break;
  . . . .
}
```

A loop like this one has all sections empty and because the condition is empty such a loop is called an "infinite loop" .

On the other hand we can have more than one statements in the initialisation and re-entrance sections. We cannot yet use the ; symbol because inside the for parenthesis this is used as a section separator . That's why here we have an exception to the rules that have been explained before :

- Multiple statements in a for section are separated by comma

Example:

```
for ( i=0,k=20 ; i<30 ; i+=2, k-=3)

    printf ("%d",k);
```

the **switch** statement

Unlike the mentioned statements where inside the parenthesis there is a condition , here we introduce a statement where inside the parenthesis is an expression . This may be a variable , a formula, a function returning a result etc.
Here is the syntax of the switch statement :

```
switch (expression)
{ case const1 :
        . . .
        break;               //  optional
  case const2 :
        . . .
        break;               //  optional
  case const3 :
        . . .
        break;               //  optional
  default:
        . . .
}
```

The program continues execution at the case where the expression matches the constant. It must be mentioned that a disadvantage at first sight is that the case keyword can only be followed be a constant and not a variable. But once again this is due to the effective way that a processor can be programmed with matching constants.

The program then does not exit the block unless a break is found. In other words next cases are also executed if there is no break; at the end of one case's code.
See the next example .

Example 1

```
#define ESC 27
. . .
void add_cr()
{    . . .
     switch (c=getch())
     { case '\n':
         putchar(c);
        case '\r':
         putchar('\r');
         break;
        case ESC:
         return;
        default:
         putchar(c);
     }
}
```

There is no break here. This means that if a '\n' is returned by getch() , then both putchar(c); and putchar('\r'); will be executed

Example 2

```
. . .
     printf("Press (A)dd ,  (M)odify,  (D)elete , e(X)it  ");
again:
     switch (toupper(getch())
     {
        case 'A':
           addnew();
           break;
        case 'M':
           modify();
           break;
        case 'D':
           delete();
        case 'X':
           return;
        default:
            printf("Illegal option");
           goto again;
     }
     . . . .
```

C Language / Lesson 5

5. #define

A multi-purpose pre-processor keyword

5.1 Symbolic names (symbolic constants)

```
#define BUFFER_SIZE 512
#define ESC 27
```

The defined names are constants i.e. they don't occupy memory and they can't change at run-time.
That's what exactly a pre-processor statement (like the ones above) is :
a statement processed at compilation time and not at run time.
Things to watch :

a. There is no = sign after the symbolic name. Instead there is a space. This first space separates the symbolic name defined at its left with the rest of the line that contains the code that replaces every occurrence of the new name in the program.
b. The statement is <u>not</u> terminated with a semicolon

given the two #define statements here above, we can have in the program statements like the following ones :

```
char filename[BUFFER_SIZE];
. . .
  if  (c==ESC)
. . .
```

5.2 Conditional compilation

```
#define LINUX
. . .
#ifdef LINUX
#define  PRG_BREAK 4
#else
#define  PRG_BREAK 3
#endif
```

5.3 Macros

A macro is a pre-processor function looking expression.

```
#define  cls()  printf("\f");
#define spc(s)  memset(s,' ' , sizeof(s))
```

Given these macros, you can write in your code :

```
char my_buffer[80];
. . .
spc(my_buffer);
cls();
```

Read about macros in the rest of this lesson.

5.4. A trilogy : From functions to macros

Three phases for the definition of a short function

5.4.1 General

The process of calling a function (and returning from it) is an overhead to the execution time of the function, which though insignificant when the function body is very rich (contains a lot of statements) , becomes very significant if the body is very short.

Take this example :

```
int absval(int x)
{  if (x<0)
      return -x;
   else
      return x;
}
```

This function obviously returns the absolute value of an integer. The work that the function does is very little. So little that the overhead of calling and returning is a "serious delay".
Of course we are talking about milliseconds but remember that C is the language of the languages i.e. system software , utilities and other important pieces of software that run continuously on a computer and milliseconds are eternity, the code therefore must be optimized for speed to the maximum.

- ***What can we do about this delay ?***

5.4.2 The ternary operator

Here is a tool that really opens the way to an optimizing solution :
First what does ternary mean ? It is a three-value expression (in our case operator), like binary is a two-value expression. And because there are no other operators that have three operands – the operator introduced here, although some papers call it "the condition operator", is traditionally called "the ternary operator".
Let's take a look at the syntax involving the three operands :

result = *condition* **?** *expression1* **:** *expression2* ;

The 3 operands are : a condition and two expressions .
The previous statement is equivalent to :

. . .
if (*condition*)
    result=*expression1* ;
else
    result=*expression2* ;

In other words the ternary operator replaces and an "if … then .. else " block of code.
The benefit of the presence of this operator will be clear at the end of this discussion, but at first sight we can speak about optimization at least of spaces since the term result does not appear twice in the statement as it does in the if… then…else block and believe me this already is reflected in the machine code.

Another comment that we must also emphasize on is :
The ternary operator is an operator and as such it gives a result. This means that in the if . . . else version the statements cannot be imperative, instead they must assign a value to the very same variable (in our example the variable is 'result'). For instance this block :

```
. . .
if (amount>1000)
        return ;
else
        bal = 0 ;
```

cannot be written with a ternary operator.

➢ ***Can we write the* absval(. . .) *body using the ternary operator ?***

Yes ! and this leads us to the second phase

```
int absval(int x)
{   if (x<0)
        return –x;
    else
        return x;
}
```

➔

```
int absval(int x)
{
    return x<0 ? –x : x;
}
```

Phase I **Phase II**

Popular misuses of the ternary operator

a. x<0? return –x : return x ;
the return is a command (imperative) and does not evaluate to a result as an expression does – we have said that the 2nd and 3rd operands are expressions

b. x<0 –x : x ;
this version does not assign the operator's result to anywhere : the result is lost

Getting rid of the function call

In the beginning we said that the overhead is the function call. In the 2nd phase example above we made some improvement regarding optimization but we did not avoid the function call.

a program statement calling the function like this

```
. . .
n = absval(k) ;
. . .
```

will "waste time" to call the function and return from it in both versions

One solution would be (for such small functions) not to call the function but write explicitly the contents of its body every time you have to calculate an absolute value :

In other words instead of

```
n = absval(k) ;
```

write

```
n = k<0? =-k : k ;
```

BUT this is not a happy solution because

- we gained something : speed
- we lost something else : readability

indeed the latter is not very readable.

Then, what is that would make us happy at last ?
Answer : Writing `n=absval(k);` and meaning `n=k<0?-k:k;`

5.4.3 Macros

The final step that leads to the perfect solution [optimization and readability] comes with the involvement of macros

- ➢ A macro is a pre-processor expression that looks like a function call i.e. it is replaced at compilation time instead of calling an existing function

Macros are similar to symbolic names / constants with the difference that they accept passing parameters. These parameters of course are **dummy arguments** because they are pre-processor and therefore have no data type. The #define keyword is used to declare them too

Example

```
#define  cls() printf("\033[2J")
```

Here we suppose that our terminal clears the screen with the character sequence ESC,[,2 and J . As a result every time we wish our code to clear the screen we must write
`printf("\033[2J")`
But this is unacceptable because a. we must remember the sequence by hart b. it is a long sequence to type. Using the macro we declare that whenever we write

```
cls() ;
```

we mean

```
printf("\033[2J") ;
```

So it looks as if we are executing the function cls() but the is <u>no such function</u>. Instead our code is replaced by the pre-processor and there is no function call to cls() which anyway doesn't exist.

<u>IMPORTANT</u>

a. whatever follows the space after cls() will replace the macro
b. **no semicolon** follows the macro definition
c. Note: this example doesn't use a dummy argument, but another one following does

There are several good reasons for using macros. They are not presented here but we can summarize them in two words

- readability
- portability

➢ *Back to our problem*

We want to get rid of a function call , remember ? Ok, let's use a macro !

```
int absval(int x)
{
    return x<0 ? -x : x;
}
```

$\rightarrow$

```
#define absval(x)  x<0 ? -x : x
```

P h a s e II **P h a s e III**

Having this macro, whenever we write `n = absval(k) ;`
this will be replaced by the pre-processor with `n = k<0? -k : k ;`

6. C standard string functions

6.1 Introduction

String functions are a standard part of most standard C library exactly as they were proposed by Kernighan and Ritchie. Unfortunately some C compiler vendors have made variations that deviate from the standard thus creating compatibility and other problems to C code authors. Nevertheless it must be mentioned that big software houses very often develop their own string functions for in-house use – in most cases totally irrelevant with the standard and finally that some compromising situations is writing macros that use standard functions giving this way the option to the user of the code to modify the macro without having to modify the code.
An example :

```
#define mycopy(srce, dest)  strcpy(dest,srce)
```

6.2 strcpy

6.2.1 the reference

Include:	<string.h>
Syntax:	char ***strcpy**(char *dest, char *srce);
Returns:	the destination string.
Description:	copies from srce to dest until a NULL terminator is found

6.2.2 Beautiful things now that we can understand the code:

Here is the code of a void version for strcpy

```
char *strcpy(char *dest, char *srce)
{ char *p;
      for (p=dest ; *dest++=*srce++ ; )
        ;
      return dest;
}
```

A whole paper could be written describing the details of this little piece of code. But since this is beyond the scope of this document, I will only give some comments for those who want to connect this code with past sections of this document.

- DO NOTHING ! The 3rd section (the re-entrance section) of the `for` parenthesis is empty, also there is a semicolon under the for that has no statement in front of it (a null statement). In both cases the meaning is "do nothing"
- The condition in the `for` is a demo of
  - the "first assign, then increase" story that is discussed in 2.3
  - the difference between an assignment = and an equality ==
- returning dest is what we call "a transparent function" : we return a value that was given to us ! Why? Because it can be used directly when the function is called . For example :

```
printf("%s\n", strcpy(b,a));
```

This is an extreme demonstration of the magic of C language. However if you don't understand what happens don't panic. Some day you will !

6.3 A reference to other string functions

Include:	<string.h>
Syntax:	char ***strchr**(char *string, char c);
Returns:	a pointer to the 1st occurrence of character c in string. If not successful then it returns NULL .
Description:	searches for char **c** in NULL terminated string

Include:	<string.h>
Syntax:	int **strcmp**(char *string1, char *string2);
Returns:	a negative value if is less than , 0 if is equal to , or a positive value if is greater than .
Description:	The strcmp function compares and returns a value indicating their relationship, as follows: Value Meaning < 0 is less than = 0 is identical to > 0 is greater than The strcmp function operates on null-terminated strings. The arguments to this function are expected to contain a null character (\0) marking the end of the string.

Include:	<string.h>

Syntax:	char ***strlwr**(char *string);
	char ***strupr**(char *string);

Returns:	a pointer to the converted string.
Description:	The strlwr function converts any uppercase letters in the given null-terminated string to lowercase. The strupr function converts any lowercase letters to uppercase. Other characters are not affected.

Include:	<string.h>
Syntax:	char *strcat(char *string1, char *string2);
Returns:	a pointer to the concatenated string.
Description:	The strcat function operates on null-terminated strings.

Include:	<string.h>
Syntax:	char ***strstr**(char *string1, char *string2);
Returns:	a pointer to the substring in string1 that matches string2
Description:	this function searches in a mainstring for the presence of a substring. If the substring is not found then it returns NULL

C Language / Lesson 7

7. Storage Classes

General

The storage class of a variable (or function) name describes the visibility (scope) of the name and the run-time life cycle of this name. In a program module the declared names have a scope within this module but also they may or may not be available to other modules (other functions written in different modules or libraries used by the program).

Within a module : Two rules that are never violated

I. A name declared in a module is known only after its declaration line

II. A name declared inside a function body is only known within that function body

a. External variables

An external name is a name available to all modules (other languages use the words global or public for this).

However, there is a confusion in declarations of external variables in C because a variable available to other modules, in addition to its **definition** (which can only be done **once**) , must be **declared** in every module where it will be used, because otherwise the compilation of that module will generate a 'Not declared' error.

Definition of an external variable	All variable declarations done <u>outside</u> function bodies and not beginning with a storage class declarator are (by default) **external definitions**
Declaration of an external variable	A variable declaration beginning with the storage class declarator 'extern' declares external variables example : `extern int max_files ;` declarations merely announce that 'somewhere else the name has been <u>defined</u>' This 'somewhere' may even be in the very same module. This is really useful when declarations are inside header files, which can be included in the module which contains the definition of the name

b. Auto variables

Auto variables can only be local to a function and the 'auto' storage class is the default class inside function bodies i.e. auto variables are declared inside function bodies An auto variable is temporary: it is created (**auto**matically) when the function is called and 'dies' when the function terminates.

example:

```
int
strlen (char *s)
{
    int n;    // auto variable

     for (n=0 ; *s ; ++n)
       ++s;
     return n ;

}
```

A special sub class of auto

Register variables

An auto variable is local and temporary and it is a practice to most languages to store this type of variables in the Stack. Of course the Stack is a memory area and as all other variables in memory have to travel back and forth whenever they are used in expressions or receive new values. Stack addressing is a little worse in speed than other variables.

That's why a brilliant idea called Register variables was invented : One or two auto variables in every function can be stored in **registers** in the CPU thus avoiding the transfers over the Data bus. In other words register variables are auto variables with a significant privilege : they are faster.
BUT :

- **A register variable does not have an address**
- **A register variable must be a scalar (because an aggregate can't fit in the CPU)**
- **A register variable (as auto) cannot be an external or static variable**

c. Static variables

Static names (variables and functions) are for some authors "the opposite of external" that is **not available outside the module** in which they are defined. For some others they are "the opposite of auto" that is **permanent** (they continuously exist during run-time).

This is not confusing because in fact static variables are both of the above. Let's put it in this way : Static variables are permanent variables not available to other modules. They can be declared outside or inside function bodies. But thus a static variable inside a module <u>retains its value from call to call</u> and this makes it an alternative to auto variables which are temporary and lose their value from call to call. On the other hand a static variable defined outside a function body would be permanent anyway (even if it were not static) and so what really changes is that the variable is not available to other modules.

```
// C language sample program
//  Storage Classes


void        // this prototype declares that somewhere else a function
subfunc() ; //  called subfunc  is defined

char buff[80];  // External (global) variable defined here
          // Allocates 80 bytes in this module and is globally visible.

static int j;      // This variable is accessible only in this module,
                  //  but has a lifetime  equal to the run-time


////  All variables below this line are declared inside function bodies
////  and are therefore available only inside the body the are declared

void
main()
{
 register char *p;      // p is declared in  main() i.e. it is auto
                  // and because it is also scalar it is legally
                  // declared as register

 extern char name[];    // External declared here defined elsewhere
                  // Size is not specified because the variable
                  // allocates memory via a definition in another
                  // module.

      /*  Here goes the code .....*/
}

void
subfunc()
{
  char *p;  // This auto variable is not the same as the register
            //  variable declared in main()
      ;
      /*  Here goes the code .....*/
}
```

<u>DISCUSSION</u>

Empty Brackets

By now we have encountered all the cases where empty brackets ([ ]) are legal in C. Let's review them

a. Definition of an **initialised** string   `char name[] = "Dennis Ritchie" ;`
  It is obvious that the string <u>must</u> be initialised

b. Declaration of an external string   `extern char wrk[] ;`
  (See previous sample program)
  the size here is "not our job" because the name is not created here

c. Character pointers as Passing parameters   `void foo(char s[]) { . . . }`
  <u>Warning !</u> The argument in the parenthesis is <u>not</u> a string
  but a **pointer** (see Lesson 4 – the for statement)

C Language / Lesson 8

8. Files and Devices

8.1 Standard devices and redirection

In the C standard library there are two basic functions which read and write respectively from/to the user's terminal :
getchar() returns one character every time it is called from the so called standard input
putchar(c) sends c to the standard output.
In most environments a file may be substituted for the terminal using the symbol > for the output and < for the input. If the program myprog for example uses getchar() to read from the terminal then the command

```
myprog < infile
```

will read input from infile instead of the keyboard. This is called **redirection**.

The keyboard and the screen which are the peripheral devices related to the above standard input and standard output (logical) devices respectively plus two or three other standard devices which are explained below are treated for the first time under UNIX (and of course "C") and later under DOS and other console oriented operating systems as files.

Although file functions are not discussed here it must be mentioned that when opening a file in C using one of the two existing standard functions a **File Descriptor** is returned. This is a value which is used as the reference to the opened file. The functions previously mentioned getchar() and putchar() are actually file functions which read and write respectively from two special files opened automatically when a program starts.
In other words these standard devices are treated as files. This is a great feature since it is related with other concepts which give a great flexibity to a programmer's strategic. For example a program made to write text to a file can be used to write to the standard printer file i.e. to print instead of storing the text on disk. Here is a table showing all the standard devices and other important names related to them. It must be mentioned that Windows and DOS do not support all common features of standard devices; there are differences in redirection :

- UNIX commands can redirect the error device also
- the name at the command line are not the same (here DOS names are shown)

Device	**Name in C** (ANSI descriptor)	**DOS Device**	**Mode**	**System file descriptor**
Keyboard	stdin	CON	Input	0
Screen	stdout	CON	Output	1
Error	stderr	(N/A)	Output	2
T/C port	stdaux	COM1	I/O	3
Printer	stdprn	LPT1	Output	4

A function called **getc(fd)** similar to getchar() reads one character from file fd every time it is called. This means that `getchar()` is identical to `getc(stdin)`

Comments
COM1 and LPT1 are names which are used also as file names, in which case they are not used as standard devices.

The ANSI descriptors are pointers of type FILE (a special structure predefined in C – see next section) . So while file pointers are used to access files (or devices) using ANSI file functions , we can see that using system file functions the descriptors are non-negative integers.
We call such a file pointer a **stream** since using the basic file functions which read or write a character every time other functions read/write a sequence of characters for example one line terminated with CarriageReturn and LineFeed

8.2 Filters

A filter is a program that reads from the standard input (keyboard) transforms or processes the input data and sends the results to the standard output. In other words for a program to be called "a filter" , it must contain statements that read from the keyboard and statements that write to the screen.

Such a program is **sort** which reads lines of text from the standard input , sorts them and then sends the sorted results to the standard output.
Obviously the author of this utility did not have in mind that some idiots will sit and type names just for the fun of seeing them sorted on the screen. The purpose of such a program is to use redirection.
For instance suppose that we have a text file called MYNAMES.TXT and its contents are :

```
Polyhymnia
Terpsichore
Calliope
Melpomene
Clio
Euterpe
Thalia
Urania
Erato
```

We can use the **sort** program to put the names in ascending alphabetical order. And because sort is a filter we will redirect the standard input and the standard output as follows :

```
SORT < MYNAMES.TXT > SORTED.TXT
```

This command will read the given file, will sort the lines and send the result to the standard output which due to redirection will create a file called SORTED.TXT in which the output will be stored. Of course we will not see any output on the screen.

File : SORTED.TXT

```
Calliope
Clio
Erato
Euterpe
Melpomene
Polyhymnia
Terpsichore
Thalia
Urania
```

8.3 Pipes

A pipe is a command line containing more that one commands (built-in or program names) where each command redirects its output to the next command's standard input . The format of a pipe is the following :

$$\textbf{cmd}_1 \mid \textbf{cmd}_2 \mid \ldots \mid \textbf{cmd}_n$$

Now it is clear why we defined filters : for a program to participate in a pipe it must be a filter because it must be able to read the output of the previous command in line (with the exception of the first command of the pipe) and send output to the standard input of next command.

C Language / Lesson 9

9. File Functions

There are two groups of file functions in C :

- system functions
- ANSI functions

If you want to be on the safe side, you must use the ANSI functions because they are standardised and hopefully they will exist under a new platform in case you decide to migrate you're application. And these are the ones that are presented here and used through out this document.

Regarding the system functions one two words : they "skip" the C language methods for file handling and directly talk to the interface of the operating system. Results: a. they are primitive and don't do any formatting or character recognition for instance find the end of a text line and b. they are faster. A good reason for using them is retrieve or write very big blocks of data from a disk .

You can tell that a file function is an ASCII one because they begin with an f (only confusion fflush vs. flush)

fopen Opens a file.

FILE *fopen(char *filename, char *mode);

Required Header : <stdio.h>

Return Value : If the function fails , it returns NULL.

Parameters

filename	Filename
mode	Type of access permitted

Remarks

The fopen function opens the file specified by filename.

The character string mode specifies the type of access requested for the file, as follows:

"r" Opens for reading. If the file does not exist or cannot be found, the fopen call fails.

"w" Opens an empty file for writing. If the given file exists, its contents are destroyed.

"a" Opens for writing at the end of the file (appending) without removing the EOF marker before writing new data to the file; creates the file first if it doesn't exist.

"r+" Opens for both reading and writing. (The file must exist.)

"w+" Opens an empty file for both reading and writing. If the given file exists, its contents are destroyed.

"a+" Opens for reading and appending; the appending operation includes the removal of the EOF marker before new data is written to the file and the EOF marker is restored after writing is complete; creates the file first if it doesn't exist.

fgets Reads a line of text from a file opened for input

int fgets(char line[] , int nmax, FILE *fd);

Required Header : <stdio.h>

Read a line of text from fd not more than nmax chars

Returns 0 on End of File

fclose Closes a stream.

int fclose(FILE *fd);

Required Header : <stdio.h>

Close the stream **fd**

fseek Moves the file pointer to a specified location.

int fseek(FILE *stream, long offset, int origin);

Function	Required Header
fseek	<stdio.h>

Return Value

If successful, fseek returns 0. Otherwise, it returns a nonzero value. On devices incapable of seeking, the return value is undefined.

Parameters

stream	Pointer to FILE structure (file descriptor)
offset	Number of bytes from origin
origin	Initial position

Remarks

The fseek function moves the file pointer (if any) associated with stream to a new location that is offset bytes from origin. The next operation on the stream takes place at the new location. The `origin` must be one of the following constants, defined in STDIO.H:

SEEK_SET Beginning of file (= 0)
SEEK_CUR Current position of file pointer (= 1)
SEEK_END End of file (= 2)

You can use fseek to reposition the pointer anywhere in a file. The pointer can also be positioned beyond the end of the file.

When a file is opened for appending data, the current file position is determined by the last I/O operation, not by where the next write would occur. If no I/O operation has yet occurred on a file opened for appending, the file position is the start of the file.

ftell Gets the current position of a file pointer.

long ftell(FILE *stream);

Function	Required Header	Optional Headers
ftell	<stdio.h>	<errno.h>

Return Value

ftell returns the current file position. On error, ftell returns –1L and errno is set to one of two constants, defined in ERRNO.H.

```
//--------------- Example -----------------
//  Display the contents of file "c:\\wrk\\sample.txt"

#include <stdio.h>

char text[120];
char file[] = "c:\\wrk\\sample.txt" ;
FILE * fdtest;
void main( )
{
        if ( (fdtest=fopen(file,"r")) == NULL)
        {       printf("OPEN ERROR\n");
                return;
        }
        while ( fgets(text,sizeof(text),fdtest))
                printf( "%s\n" , text);

        fclose (fdtest);
}
```

C Language / Lesson 10

10. Passing parameters at the Command Line

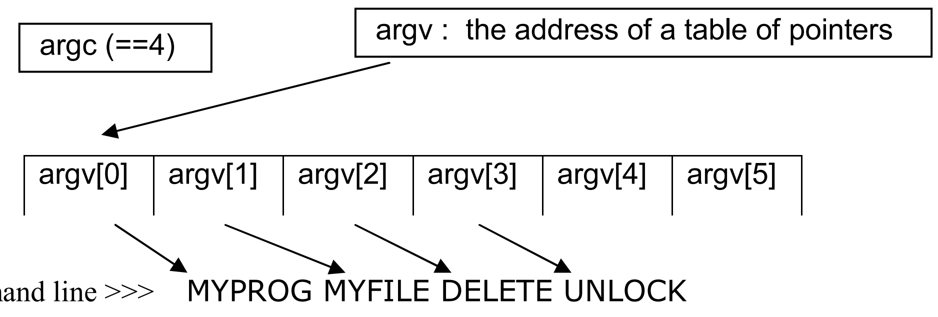
A command line is a text line submitted to the operating system (by the user or by another program) in order to start the execution of a command (usually a program).

- A command line is not a program line

The command line consists of arguments which –according to UNIX and C specifications- are separated **with spaces.** The first argument is the command name (program name) and the rest are the so called **passing parameters**. When a program is executed, the command line which launched the execution is passed by the operating system to it. The C language start up code -which is invisible to you but always executed before the main( ) function starts - sees that the information from the command line is passed to main( ). This is done with the use of two arguments passed to main ( ) as shown below.

An example with 3 passing parameters i.e. 4 arguments at the command line.

```
void
main(int argc , char **argv)
{...
```


argc (==4)
argv : the address of a table of pointers
argv[0]
argv[1]
argv[2]
argv[3]
argv[4]
argv[5]
Command line >>> MYPROG MYFILE DELETE UNLOCK

...

Example

```
///// Display all the arguments (incl. the program name) ///
void main(int argc, char **argv)
{ short i;

      for (i=0 ; i<argc ; ++i )
         printf("%s\n" , argv[i] ) ;

}
```

Exercise

If the program name is MYPROG in each of the following cases, what is the output when executing the command line

```
MYPROG MYFILE DELETE UNLOCK
```

Case I

```
void main(int argc, char **argv)
{ short i;
      for (i=0 ; i<argc ; ++i )
        printf("%s\n" , argv[i] ) ;
}
```

answer

```
MYPROG
MYFILE
DELETE
UNLOCK
```

Case I I

```
void main(int argc, char **argv)
{     while (argc-- )
        printf("%s\n" , *argv++ ) ;
}
```

Case I I I

```
void main(int argc, char **argv)
{     while ( --argc )
        printf("%s\n" , *++argv ) ;
}
```

Case I V

```
void main(int argc, char **argv)
{     while (argc-- )
        printf("%s\n" , argv[argc] ) ;
}
```

Find the remaining answers yourself !

All four cases above very clearly demonstrate the various syntactic rules that apply to pointer pointers (this is the first time they were encountered in this text). However in reality we would never write a program to display the arguments passed to it at the command line.

Here is a more realistic program :

```
#include <stdio.h>
#include <string.h>

char filename[80];

int main(int argc, char **argv)
{
       if (argc>1)
              strcpy(filename,argv[1]);
       else
       {      printf("You did not pass a file name at the command line !\n");
              printf("\nFilename ? ");
              gets(filename);
              if (filename[0]=='\0')
                     return 1;
       }
       printf("File=%s\n", filename);
       /////  the program continues here
       return 0;
}
```

The above program expects a filename as a passing parameter at the command line.
If there is no passing parameter (`argc==1`) then it displays a message and asks the user to enter the name from the keyboard.

Read also : K&R Section 5.10

C Language / Lesson 11

11. Arrays of pointers vs. Arrays of Strings

11.1. Arrays of strings - Initialization

Let's first remember the syntax rules for initialising an array of strings (officially: a two dimension char aggregate)

Here is an example of an array containing the names of the weekdays :

```
        char weekday[7][10] = {
                "Sunday", "Monday", "Tuesday", "Wednesday",
                "Thursday", "Friday", "Saturday"
        } ;
```

Note: [7] can be written as [] since the array is initialised

Now let's see how memory is allocated by the above definition :

S	u	n	d	a	y	\0			
M	o	n	d	a	y	\0			
T	u	e	s	d	a	y	\0		
W	e	d	n	e	s	d	a	y	\0
T	h	u	r	s	d	a	y	\0	
F	r	i	d	a	y	\0			
S	a	t	u	r	d	a	y	\0	

This is a 7x10 frame i.e. a total of 70 bytes. We observe that some of these bytes are not used (they contain garbage) but in the above example that doesn't really matter. The table is relatively small and the total number of unused bytes is only 13. This as an absolute number is small but also the percentage of unused bytes compared to the total size of the allocated memory block (13/70) is also small . There are two factors that can worsen the memory leak of the string array implementation :

- the diverse length of the string values in the array
- the number of elements of the array

In the following example it is obvious that the waste of memory is significant

```
char message[10][47] = {
        "Bye",
        "Do you want to permanently delete these files?" ,
        "Exit",
        "OK",
        "Pause",
        "Please wait...",
        "Press a key to continue",
        "Quit ?",
        "Save file?",
        "Variable not declared"
} ;
```

Only 144 bytes out of 470 (10x47) are used in the above array definition. And 470-144= 326 wasted bytes on a total of 470 is significant. Almost double the size of the used memory is not used.

Anyway, let's use the array

Let's write a few lines of code to display one of the strings in the array

```
void display_message(int msg_no)
{    fprintf( stderr, "%s\n", message[msg_no] );
}
```

message[msg_no] is the address of the respective string because the defined variable is *a two dimension aggregate.* You must keep in your mind that given a multi-dimension aggregate, an expression with less pairs of square brackets that the number of dimensions is the address and not the value of an element in the aggregate. In our example message[msg_no] is an expression with **one** pair of brackets while the definition defines a **two-dimension** variable. It therefore represents an address and this proves the proper usage in the code because the modifier %s requires as its corresponding argument the address of a string.

We will re-visit the above example under a different approach in the next paragraph.

11.2 Using a pointer array

A pointer array contains pointers of course. In other words it contains a list of addresses and in the case of a char * array , the addresses point to strings. If the pointer array is initialised to string constants then the addresses of the pointer elements of the array are addresses of constants. In other words the strings **do not belong in the array** but instead they are stored somewhere in the constants memory segment whereas the array contains only the pointers. Assuming that a pointer is a 32-bit value then a pointer's size is 4 bytes. Therefore the size of an array of 7 pointers is 4x7=28 .

Here is the implementation of the dayweek definition , this time using pointers instead of strings.

```
char *weekday[7] = {
      "Sunday", "Monday", "Tuesday", "Wednesday",
      "Thursday", "Friday", "Saturday"
} ;
```

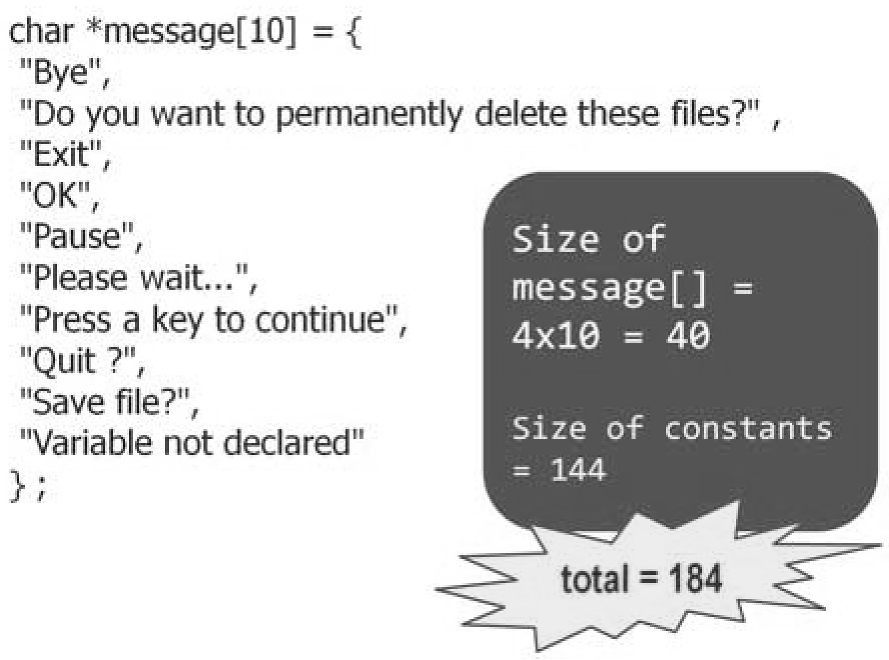
Doesn't it look like the definition of the array of string at the top of this paragraph ?

We have not finished yet. While the table is a 28 byte memory block this is not the only memory allocated. There's a an additional allocation of memory for the string constants. This time we count the characters (and the terminators) of each dayname in the initialisation section . Total = 57 . And maybe this is less than the size of the 2 dimensional definition 7x10 but there is the additional 28 bytes of the array. Total 28+57 = 85. Conclusion: we have allocated more memory with this second approach using pointer arrays.

If we repeat this last process for the message[ ] array , the results will be quite different.

- ✔ **char *message[10]** allocates 4x10=40 bytes
- ✔ the total size of the string constants in the definition is 144

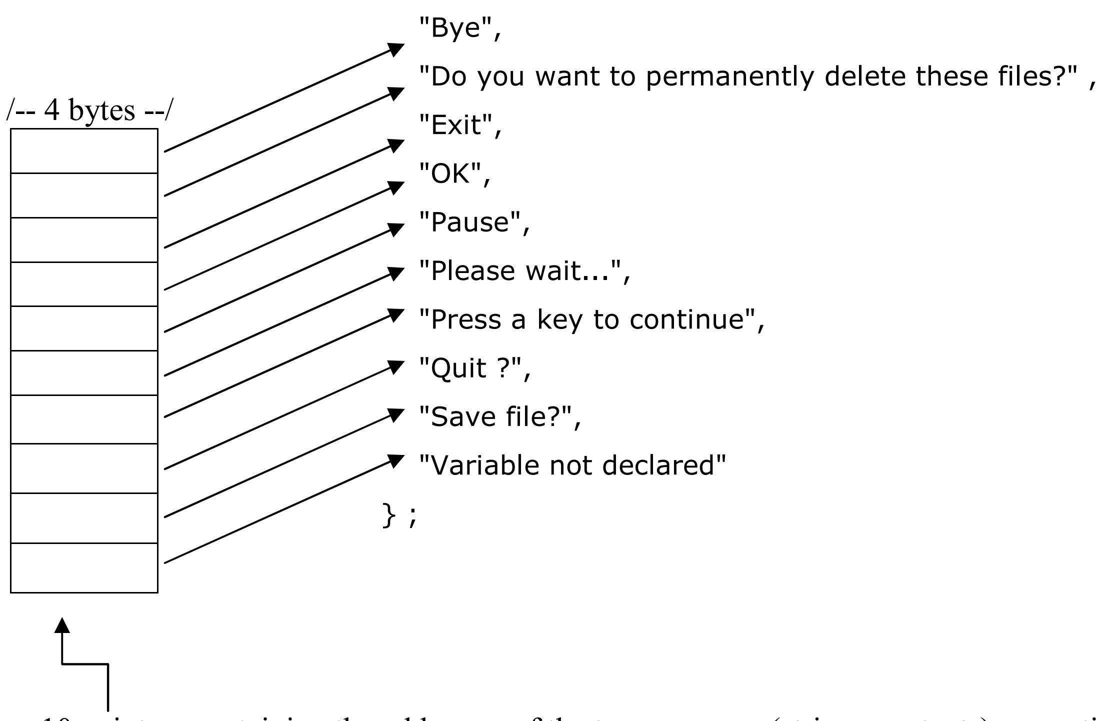
Result : Total memory used = 144+40 = 184 ***only !***



```
char *message[10] = {
 "Bye",
 "Do you want to permanently delete these files?" ,
 "Exit",
 "OK",
 "Pause",
 "Please wait...",
 "Press a key to continue",
 "Quit ?",
 "Save file?",
 "Variable not declared"
};
```

Here is a diagram of the memory allocation of the above array of pointers :

```
char *message[10] = {
```



```
"Bye",
"Do you want to permanently delete these files?" ,
"Exit",
"OK",
"Pause",
"Please wait...",
"Press a key to continue",
"Quit ?",
"Save file?",
"Variable not declared"
} ;
```

10 pointers containing the addresses of the ten messages (string constants) respectively.
(we assume here that a pointer is a 32 bit (4 byte) value.

We have come to the conclusion that the more the string values are of uniform size the stronger is the possibility that an array of strings is the best choice in terms of space optimisation and in the opposite case an array of pointers is the choice.

But do we have to modify our code because we have chosen the latter ? Thank God , **No** !

Because the notation we used before in the display_message(...) example this time does the same job although it means different thinks. The message[msg_no] this time is not the address of the element but the value (according to the rules presented above regarding brackets). This time however the value of the element is a pointer : the address of the required string. So the result is the same. This means that in case we have implemented a big project using a multi-dimensional array definition with a significant number of references to it in our code and later we change our mind and wish to turn it into a pointer array, there will be no problem ; all references to the elements of the array will be correct.

11.3 Terminating Arrays

In the weekday[] example above the size of the array (regardless of the implementation method that one would choose) is fixed and known in advance, because the days of the week are (and will always be) seven. If a function is to display these daynames on the screen it would look like this :

```
void display_days()
{ int i ;
   char *weekday[7] = {
     "Sunday", "Monday", "Tuesday", "Wednesday",
     "Thursday", "Friday", "Saturday" } ;
     for (i=0 ; i<7 ; ++i )
        printf("%s\n",weekday[i] ) ;
}
```

As we can see, the size of the weekday[] is passed in the code as a constant (or it could be a symbolic constant if you prefer) and the program 'knows' where to stop.

But what would happen if an array contained a number of elements that changes through the time as the project expands and new versions are produced etc. For instance, let an array contain a list of imperative words i.e. commands which make up an instruction set of a special protocol :

```
char *cmd[] = {
     "BEGIN", "CLOSE", "ERASE", "FIND",
     "LOAD", "OPEN", "PRINT" , "RESET",
     // etc. etc.
     "WAIT", "WRITE"
} ;
```

Here the advantages of leaving the square bracket block empty is that a. we don't have to count the number of words in the array and b. whenever we add a new word there is no change to be done in the declaration of the array because the size of the dimension is omitted and automatically calculated at compilation time.
But can we display the contents of the array without counting the number of elements in it ? If we used the style of the previous for (. . .) loop, then we should mention explicitly the number of iterations that the loop must execute. Which already is the loss of the advantage of **not** counting the number of

elements and what is worse we must change the constant number whenever we add a new element in the array.
All this discussion leads to the conclusion that another way of determining the end of the array is needed. Let's add then a **terminator** to the array exactly as C does with strings (arrays of characters).
When a string is processed by a string function the characters in it are accessed one by one ***until a Null character ('\0') is found***.

So, if character arrays are terminated with a character (the null character) , then what can terminate an array of pointers is obviously a pointer : the **NULL pointer**. Remember that NULL is a pre-processed name defined usually in the stdio.h file and it represents the address 0000 which is never used to store anything in a C program and thus it can represent an unsuccessful access or a negative response. For example if the program asks a function 'where is that string stored?' and the function answers 'it is stored at NULL' then it means 'it was not found' . In a similar way including the NULL pointer in a list of string addresses it can never be confused with the address of an existing string. Let's rewrite the above definition :

```
char *cmd[] = {

        "BEGIN", "CLOSE", "ERASE", "FIND",
        "LOAD", "OPEN", "PRINT" , "RESET",
        // etc. etc.
        "WAIT", "WRITE" , NULL
} ;
```

The following function will search in the array and return the order number in the list of a word we are looking for :

```
int
cmd_index( char *word)

{ int i ;

      for (i=0 ; cmd[i] ; ++i )
        if ( strcmp(word,cmd[i])==0)
          return i ;

      return -1 ; // Not found
}
```

The `cmd[i]` condition is equivalent to `cmd[i]!=NULL` in other words the loop keeps executing while the current pointer - element in the cmd array is not NULL and when the NULL is reached the loop terminates.

How nice but what if I suddenly decide that because of the uniform size of the strings in the array and in order to reduce the size of cmd I wish to change my definition from

```
char *cmd[] = { . . . } ;
```

to

```
char cmd[][6] = { . . . } ;
```

In everyday practice some compilers would accept this without any modification. But if we want to say things with their real names, the NULL has not place in the list of initial values of the array cmd[ ] in the latter definition because it is not a string but a pointer and this array is **an array of strings** and not an array of pointers as it was in the former implementation. So, the question is : which string shall I choose to terminate a list of strings ? There could be many different answers to this but in my opinion only one answer is reasonable : the empty string "" .

Is there anything else to be changed ? Yes !

The condition cmd[i] mentioned before is not appropriate anymore because cmd[i] this time is the address of a string in the array of strings and therefore it will never be NULL . Below is the final implementation with the usage of an array of strings after all the required modifications have been done.

```
char cmd[][6] = {

      "BEGIN", "CLOSE", "ERASE", "FIND",
      "LOAD", "OPEN", "PRINT" , "RESET",
      // etc. etc.
      "WAIT", "WRITE" , ""
} ;
int
cmd_index( char *word)
{ int i ;
     for (i=0 ; cmd[i][0] ; ++i )
        if ( strcmp(word,cmd[i])==0)
           return i ;

     return -1 ; // Not found
}
```

Now the condition cmd[i][0] 'says' : ***'while the first character of the current string is not the Null character '\0' ...'*** or in other words ***'while the current string is not the empty string "" ...'***.

Read also : K&R Section 5.9

11.4 Dynamic memory allocation

11.4.1 Uninitialised arrays

In the previous cases of this lesson the criterion for selecting an array of strings vs. an array of pointers was the initial data of the array. In other words we were only talking about initialised arrays.

In real coding the occurrence of uninitialised arrays is more frequent because very often the initial values are stored in a file or database so that they can be changed at run-time without having to modify the program code.

What do we do then?

One option would be to pre-allocate strings of big size so that no matter how big the input string is, it can fit in the string element of the array. Example :

```
char  title[100][180] ;  // an array of 100 strings  180 bytes each
```

And of course if we are talking about a few kilobytes then this is not a bad idea.
It is the same as a classroom with 40 seats : we do not know how many students there will be in the class but there are always 40 seats even if there are only 12 students.

But what if for the sake of one very long string we have to pre-allocate a table with very long strings (like in the example of 11.2)

11.4.2 Allocating memory from the heap

C language has a mechanism for requesting at run-time for available memory from the heap (a dedicated to the program memory area for this purpose). And in this case we are only talking "pointers".

A function called malloc is used to request for a block of a specific size. The function returns the address of the reserved memory block. We must therefore store it in a pointer variable.

```
        void * malloc(int n);
```

Example :

```
  char *text ;  // an uninitialised pointer – points to nowhere
  text=(char *) malloc(30000) ;  //  request for a block of 30000 bytes
 //  now text points to the allocated block
```

Note : a void pointer is a pointer that we do not know in advance to what type it points.
That's why every time we call the function we must declare the data type of where the return value of malloc will be assigned. This is called casting (a long story not covered in this document) and in the above example it is done by preceding the word "malloc" with (char *) .

11.4.3 Releasing the allocated memory

In many case we "borrow" very big blocks of memory from the heap using the malloc function. But if we keep borrowing without returning the memory when we don't need it then we fall into what we call "memory leakage" and we will run out of memory.

11.4.3 An example of dynamic memory allocation with an array of pointers

```
#include <stdio.h>
#include <string.h>
#include <conio.h>
#include <malloc.h>
/////////// PROTOTYPES ///////////
void display_contents_of_tbl();
void release_memory();
/////////// VARIABLES ////////////
char *tbl[50];//  An array of Pointers
char infile[60];
char phrase[60];
FILE *fdin;
/////////// THE CODE /////////////
void main(int argc, char** argv)
{ int n;
   if (argc!=2)    // If there is not exactly one pass.parameter
      return;      //  then  Bye-bye
   strcpy(infile,argv[1]);    // 1st Pass.Parameter must be a filename
   printf("Dynamic Memory Allocation\n");
   printf("Input File = %s \n",infile);
   if ((fdin=fopen(infile,"r"))==NULL)         // Open the file
   {  printf ("Open Error");
      return;
   }
   for (n=0 ; fgets(phrase,sizeof(phrase),fdin); ++n)
   // Read all the lines from the file
   {  tbl[n]=(char *) malloc(strlen(phrase)+1);
      // allocate memory for this line
      strcpy(tbl[n],phrase); // and copy the line into the allocated space
   }
   tbl[n]=NULL;         // put a NULL at the end of the table of pointers

//////
   display_contents_of_tbl();

   printf("\n== END OF FILE ==\nPress a key ");
   getch();                    // wait for a key to be pressed
   release_memory();
}

void display_contents_of_tbl()
{ char **ptbl;
   for (ptbl=tbl ; *ptbl; ptbl++)
      printf("%s\n",*ptbl);
}

void release_memory()
{ int i;
   for (i=0 ; tbl[i]    ; i++)
      free(tbl[i]);
}
```

C Language / Lesson 12

12. Structures

12.1 Definitions

A structure is a program defined composite data type containing values of different types.
We can also define the structure as "a set of values referenced collectively".
This last definition is not absolutely clear because one could say "an array is also a set of values referenced collectively". That's why we will give emphasis to the terms

program defined data type :	a structure is treated as a new data type where variables can be defined to have this data type – following are examples demonstrating this
composite :	while an array contains elements of the same type , a structure can contain members (we don't call them elements) of different type it is therefore composite.

- An array consists of **elements** whereas a structure consists of **members**

Every new structure definition is done using the word `struct` followed by a chosen identifier called ***the Tag***. Then, having defined the new structure type we can go on defining variables that are of that particular structure type
The following examples demonstrate structure definitions and usages of the tag :

```
///////////// Structures //////////
/* Following is a structure description */
struct Student
{    char code[6];
     char name[30];
     int age;
     long fees;
     char tel_no[12];
};
/* The above definition does not define any variables at all this means that there will
follow variable definitions which have the type of the above structure,
for this a tag is required that's why we included the tag  Student  in the structure.
Now we can declare variables of the type struct Student as follows
*/
struct Student st[50] , *stp , wstud ;

/* ====  A structure description with tag and variable declaration ==== */
struct Student
{    char code[6];
     char name[30];
     int age;
     long fees;
     char tel_no[12];
} w , a[14];
/* ==== a case (with variable definition) where Tag is not needed ==== */
struct {
     char code[6];
     char name[30];
     int age;
     long fees;
     char tel_no[12];
} w , a[14];
```

The line

```
struct Student st[50] , *stp , wstud ;
```

in the above piece of code contains variable definitions given the new type `struct Student.`

Here are some very important comments on this line and the given definition examples :

- In C language the word struct is mandatory before a structure tag. In C++ this is not the case, given the above you can have a definition like : `Student wstud ;` In the following we will see how such a definition is possible is C also by means of the keyword typedef .
- Given a struct data type , we can define scalars, aggregates, pointers etc. exactly as we can do with any built-in type.
- The case where the tag was omitted is because the variables were defined in the very same statement and because we don't plan to define other variables of this type later and therefore we won't need to use the tag again.

12.2. Notation

12.2.1 Expressions with members

To assign or retrieve the value of a member of a structure variable, the following format is used :

structurevariable.membername

Examples:

```
wstud.fees = 540 ;
strcpy(wstud.name,"Anderson Ian");
st[3].age = 21 ;
```

12.2.2 The special case of structure pointers

Pointers to structures are like all other pointers. They simply point to a structure. For example the definition :

```
struct Student *sp ;
```

defines a pointer sp which points to a structure of type struct Student.

A member of a structure is denoted as **s.member** where s is the structure variable. The compiler translates this as "***the offset of the member from the base address of the variable***"

However, the notation **sp.member** where sp is a pointer is incorrect because it involves the address of the pointer instead of the address of the structure.

Let's make this clear with an example :

Given

```
struct Item
{    char code[8];
     char description[40];
     double qty;
};
```

the definition `struct Item d ;` allocates memory as shown here

code	description	qty

d ------ 8 ------|------------------------- 40 ------------------------------------| ----- 8 -------- |

Where is qty ? It is at the address of d plus the offset of qty that is : &d + 8 + 40

But if instead of d we had a pointer definition like
`struct Item *z ;`
then the statement

The qty in z is at the address of z plus the offset of qty that is : &z + 8 + 40

is <u>incorrect</u> . The correct statement is

The qty in the structure <u>pointed by</u> z is at the address <u>stored in z</u> plus the offset of qty that is : z + 8 + 40

History: Originally to correct this, the following notation was used

```
(*sp).member
```

because the indirection operator * (see 2.2 Pointers) "tells" the program that we don't want to use as base address the address of the pointer but the value of the pointer.

For our previous example this would be : `(*z).qty`
An alternative notation was finally agreed for this expression because (they said that) the above notation is not very convenient

```
sp->member
```

The following example clarifies the usage of `->` but also is a good opportunity to see how structure variables can be initialized

Example :

```
#include <stdio.h>
struct Item  d= { "123123", "GLYCERINE", 45.62 };
struct Item *z;
void foo()
{
   z=&d;   // make z point to d
   printf("VERSION I : Quantity is %f\n" , d.qty);
   printf("VERSION II: Quantity is %f\n" , z->qty);
}
```

The output of both printf statements is exactly the same

12.3 typedef

The typedef directive allows us to introduce a new data type (program defined). In the case of structures the usage of typedef allows us to do something that is standard in C++. The variable definition

```
struct Student *sp ;
```

which we presented above would be written in C++ without the word struct . This in C is not legal and here is where typedef comes in.
If we define a **new data type** called Student instead of a structure with the tag Student, then we can use the C++ syntax to define a structure variable.

```
typedef struct
{    char code[6];
     char name[30];
     int age;
     long fees;
     char tel_no[12];
} Student;

Student *sp ;  // this time Student is a defined type(not a tag)
```

Additionally we could given a tag name to the structure and use that also.
Or even more we could define the structure type alone and then define the Student type using the typedef :

```
struct  _student
{    char code[6];
     char name[30];
     int age;
     long fees;
     char tel_no[12];
};

typedef struct _student Student ;
```

Now we can use both ways to define variables of type Student :

a. Student t;
b. struct _student u;

```
// Start a new console project and paste this code in the CPP file
// STRUC1.cpp

#include "stdafx.h"
#include <conio.h>

#define MAX_STUDENTS  200
#define MAX_COURSES  50

struct Students {
      long ID;
      char name[30];
      struct Grades {
            char course[10];
            char grade[4];
      } g[MAX_COURSES];
} ;


struct Students st[MAX_STUDENTS] , *sp;


void
main(int argc, char* argv[])
{
 int i;
 int j;
 char w[12];
    printf("Please Enter data as required below!\n");
    printf("(To terminate , leave the name empty)\n\n");


///////// Loop to enter data into structure st[]   ////////////

    printf("\n\n");
    for (i=0 ;  i<MAX_STUDENTS ; i++)
    {
        printf("Student #%d :\nName : ",i+1);
        gets(st[i].name);
        if (st[i].name[0]=='\0')
           break;
        printf("ID : ");
        scanf("%ld",&st[i].ID);
        gets(w);  // For unknown reasons scanf doesnt read CR and LF
                 // use this line as a correction if needed
        printf("  Courses taken\n");
        for (j=0; j<MAX_COURSES ; j++)
        {
           printf("\tCourse Code : "); gets(st[i].g[j].course);
           if (st[i].g[j].course[0]=='\0')
                 break;
           printf("\tGrade : "); gets(st[i].g[j].grade);
        }
        printf("----------------------\n");
    }
///////// Loop to display data of structure st[]   ////////////

    for (sp=st ; sp->name[0] ; ++sp )
    {
        printf("%s\t%ld\n", sp->name , sp->ID);
        for (j=0 ; sp->g[j].course[0] ; ++j)
           printf("\t%s\t%s\n", sp->g[j].course,sp->g[j].grade);
        printf("\n");
    }

    printf("Press a key to exit ...");
    getch();
}
```

```
#include <stdio.h>
/*****************************
* C language sample program  *
*  Structure pointers        *
*****************************/

struct Seasons {
      char name[7];
      struct Seasons *next ;
} ss[] = {
      "Fall"  , &ss[3], // points to Winter
      "Spring", &ss[2], // points to Summer
      "Summer", NULL,         // end of struct array
      "Winter", &ss[1]  // points to Spring
};

void
main()
{
  int i;
  struct Seasons *sp;
      printf("\nSeasons in alphabetical order \n");
      for (i=0 ; i<4 ; i++)
            printf("%s\n",ss[i].name);

      printf("\nSeasons in chronological order \n");
      for (sp=ss ; sp ; sp=sp->next)
            printf("%s\n",sp->name);
}

/**** OUTPUT **************
Seasons in alphabetical order
Fall
Spring
Summer
Winter

Seasons in chronological order
Fall
Winter
Spring
Summer
*************************/
```

```
// Structure pointers
// a simple program
//

#include <stdio.h>
#include <string.h>

typedef struct {
     char word[10];
     char *meaning;
} Dictionary;
/* Now we have described a new data type called Dictionary
 BUT we have NOT yet defined any variables at all !
 Let's do it now
*/


Dictionary *dptr;
Dictionary d[]= { // since I will initialize it (in this block)
                  // the size may be omitted in the brackets []
     "RECURSION",  "A backward movement",
     "REDUNDANT",  "Excessive",
     "REGISTER" ,  "A book with regular entries of details exactly recorded",
     "REGULAR",    "Having a form which follows some rules",
     "REMOTE" ,    "Distant",
     "ROTATION",   "The action of moving around a center",
     "",   NULL    // These are terminators indicating the end of the list
} ;

char word[20];

void
main()
{
     printf("Structures\nThe dictionary structure\n\n");
// PART I : Display the words in the dictionary /////////////////
     for (dptr=d ; dptr->word[0] ; ++dptr)
        printf("%s\n", dptr->word) ;

// PART II: Find the meaning of a word   ////////////////////////
     while (1)
     {  printf("Word (press Enter to exit) : "); gets (word);
        if (word[0]=='\0')
           break;
        strupr(word);    // Convert the word to Upper case first
        for (dptr=d ; dptr->word[0] ; ++dptr)
           if (strcmp(word,dptr->word)==0)
            break;
        if (dptr->meaning)
           printf("Explanation= %s\n\n",dptr->meaning);
        else
           printf("Sorry! Unknown word\n\n");
     }
}
```

The ASCII code

	0	1	2	3	4	5	6	7
0				0	@	P	`	p
1			!	1	A	Q	a	q
2			"	2	B	R	b	r
3			#	3	C	S	c	s
4			$	4	D	T	d	t
5			%	5	E	U	e	u
6			&	6	F	V	f	v
7			'	7	G	W	g	w
8	BS		(	8	H	X	h	x
9	TB		)	9	I	Y	i	y
A	LF		*	:	J	Z	j	z
B			+	;	K	[	k	{
C	FF		,	<	L	\	l	|
D	CR		-	=	M	]	m	}
E			.	>	N	^	n	~
F			/	?	O	_	o	

Comments

To find the position of a character in the table in HEX , combine the column digit and the row digit
For example letter Q is in position hex 51 because the column number is 5 and the row number is 1.

Each upper case character is 32 positions before the corresponding lower case character

Columns 0 and 1 contain the control codes which are unprintable.

BS=BackSpace , TB=Tab , LF=LineFeed , FF=FormFeed , CR=CariageReturn

References :

1. B.Kernighan – D.Ritchie : The C programming language 2nd Edition

2. Bruce Hunter - Understanding C (SYBEX)

3. *C Language Reference* https://msdn.microsoft.com/en-us/library/fw5abdx6.aspx