Summary

LOGICAL/BOOLEAN OPERATORS

INTRODUCTION

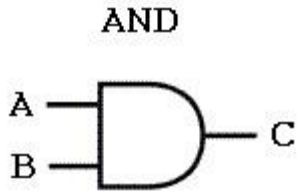
Boolean algebra is based on using only two value, 0(FALSE) and 1(TRUE).

OPERATIONS

Basically, three operations are defined for Boolean algebra:

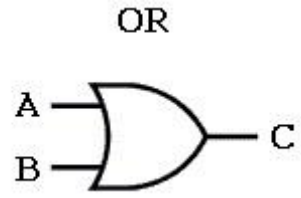
- AND: logical product. Given two Boolean values, AND operation gives as result 1 if and only if both inputs are equal to 1.

IN_1	IN_2	OUT = IN_1 AND IN_2
0	0	0
0	1	0
1	0	0
1	1	1



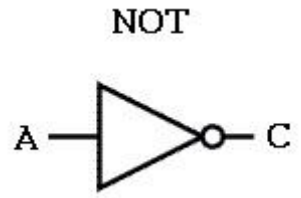
- OR: logical sum. Given two Boolean values, OR operation gives as result if at least one of the two inputs is equal to 1.

IN_1	IN_2	OUT = IN_1 OR IN_2
0	0	0
0	1	1
1	0	1
1	1	1



- NOT: logical negation. Requires only one value, so given a Boolean value, NOT operation gives as result the other Boolean one.

IN	OUT = NOT IN
1	0
0	1



BOOLEAN EXPRESSIONS

We can build a Boolean expression as for algebraic equations. Indeed:

- Letters represent a generic value;
- Usually AND is left out, as occur in product operation);
- OR is indicated with '+' sign;
- NOT is indicated with a bar above the letter (i.e.: $\overline{A}$ means NOT A)

The precedence rules for computing Boolean expressions are the same for algebraic ones, too. Firstly we compute NOT, then AND, last OR operation. To follow those rules we use the parenthesis.

PROPERTIES AND THEOREM OF BOOLEAN ALGEBRA

- Commutative: $X + Y = Y + Z$
- Associative: $XYZ = (XY)Z = X(YZ)$

- Distributive of product with respect to sum: $X(Y + Z) = XY + XZ$
- Distributive of sum with respect to product: $X + (YZ) = (X + Y)(X + Z)$
- Identity: $X + 0 = X$
  $X + 1 = 1$
- Annihilator: $X0 = 0$
  $X1 = 1$
- Idempotence: $X + X + X + \ldots + X = X$
  $XXX\ldots X = X$
- De Morgan Theorem: $\overline{X + Y} = \overline{X}\ \overline{Y}$
  $\overline{XY} = \overline{X} + \overline{Y}$
- Absorption: $X + XY = X$
  $X(X + Y) = X$

Those properties and theorem are valid for each value to X, Y, Z variables and can be demonstrated by means of truth tables.

HOW CAN SOLVE A LOGICAL OPERATION

EXAMPLE

Determine the truth table of following expression: $AB + \overline{C}$

1. Determine numbers of non-repeated variables: three in this case;
2. Build a table including all possible values combinations of variables: 3 variables means 8 possible combinations:

A	B	C
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

Now evaluate the expression following the precedence rules:

A	**B**	C	**AB**
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Then

A	B	**C**	AB	$\overline{\mathbf{C}}$
0	0	0	0	1
0	0	1	0	0
0	1	0	0	1
0	1	1	0	0
1	0	0	0	1
1	0	1	0	0
1	1	0	1	1
1	1	1	1	0

Now we are able to compute the expression

AB	$\overline{\mathbf{C}}$	**AB +** $\overline{\mathbf{C}}$
0	1	1
0	0	0
0	1	1
0	0	0
0	1	1
0	0	0
1	1	1
1	0	1

Useful links:

http://www.ee.surrey.ac.uk/Projects/Labview/boolalgebra/

NUMBERS REPRESENTATION

DECIMAL REPRESENTATION

EXAMPLE

To represent 189, which is a positive integer number, we can write:

$$189_{10} = 1*100 + 8*10 + 9*1 = 1*10^2 + 8*10^1 + 9*10^0$$

Each digit is multiplied by a 10(decimal) power; the exponent indicates position of the digit. So, the rightmost digit is multiplied by 10^0, the next one by 10^1, and so on until the last digit at the left.

To represent this number in decimal system, you need only 3 digits. Just compute the logarithm of number and add 1:

$$\log_{10}(189) + 1 = 3.27$$

Note that the integer part is 3, so it means that 3 digit are needed. In fact, we can represent 10^3 numbers from 0 to 10^3-1 (0-999).

BINARY REPRESETATION

Now we can represent the same number in binary system, adopting the same procedure as for decimal system:

$$189_{10} = 1*2^7 + 0*2^6 + 1*2^5 + 1*2^4 + 1*2^3 + 1*2^2 + 0*2^1 + 1*2^0 =$$

$$= 10111101_2$$

Each digit is multiplied by a 2(binary) power; the exponent indicates position of the digit. The result is a number composed by 8 digits. Just compute the logarithm of number and add 1:

$$\log_2(189) + 1 = 8.56$$

The integer part is 3, so it means that 3 digit are needed. In fact we can represent 2^8 numbers from 0 to 2^8-1 (0-255). In binary system, a digit is called “bit”; a number of 8 bit is called “byte”.

HEXADECIMAL REPRESENTATION

Several ways have been developed in order to simplify our handling binary data. The most common is the hexadecimal representation, in which 4 bits are used for a single digit. But there’s an issue: 4 bits means 16 possible combinations, but we can represent them by using only 10 unique decimal (0-9). This is solved by adopting the first 6 letters of the alphabet (A-F). The following table shows the relationship between decimal, binary and hexadecimal representation.

Decimal	Hexadecimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

SIGNED BINARY INTEGERS

When we deal with signed binary integers we use two's complement method.

To compute a two's complement negative number you simply take the corresponding positive number, invert all the bits, and add 1. The example below illustrates this procedure:

$35_{10} = 0010\ 0011_2$
invert -> $1101\ 1100_2$
add 1 -> $1101\ 1101_2$

Let's check it: $1101\ 1101_2 = -1*2^7 + 1*2^6 + 0*2^5 + 1*2^4 + 1*2^3 + 1*2^2 + 0*2^1 + 1*2^0 =$

$$= -128 + 64 + 0 + 16 + 8 + 4 + 0 + 1 = -35$$

Two's complement notation can be used to find easily a positive number starting from the corresponding negative one. Just take the number and extend it from 4 bits to 8 bits by simply repeating the leftmost bit 4 times. Look at the table which explain the procedure:

Decimal	4 bit	8 bit
3	0011	0000 0011
-3	1101	1111 1101
7	0111	0000 0111
-7	1001	1111 1001

5	0101	0000 0101
-5	1011	1111 1011

Useful links:

http://www.swarthmore.edu/NatSci/echeeve1/Ref/BinaryMath/NumSys.html

COMPUTER ARCHITECTURE

In computer engineering, computer architecture is a set of rules and methods that describe the functionality, organization, and implementation of computer systems.

DESIGN GOALS

Computer architecture must be:

- Functional: what functions should it support? Remember that, unlike software, it is difficult to update once deployed.
- Reliable: does it continue to perform correctly?
- High performance
- Low cost: in terms of manufacturing, design
- Low power/energy.

REMARK:
All these requirements are strongly influenced by the architecture final application: based on the architecture purpose the balance between these goals is constantly changing.

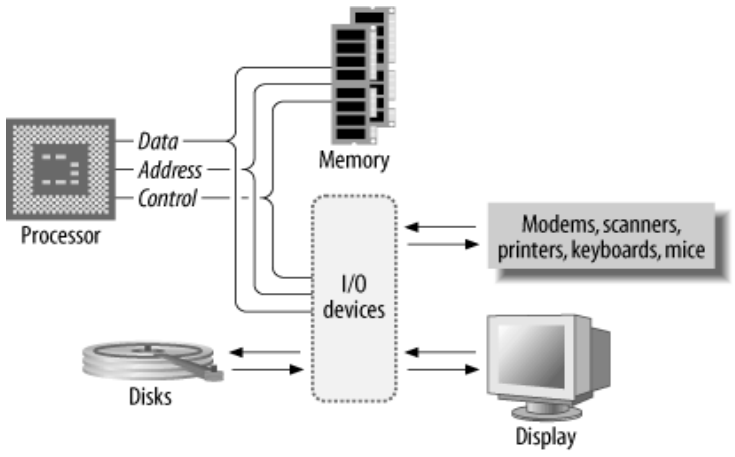
In order to understand better computer architectures, let's now recall some basic concepts: in essence, a computer is a machine designed to process, store, and retrieve data which is stored in the computer as numbers. Everything that a computer does, from web browsing to printing, involves moving and processing numbers thus the electronics of a computer is nothing more than a system designed to hold, move, and change numbers.

BASIC SYSTEM ARCHITECTURE

A computer system is composed of both hardware and software components, interconnected in some fashion to achieve the main function of the computer, which is to execute programs

There are four main structural elements:

1. Processor
2. Memory
3. I/O Modules
4. System Interconnections



This form of computer architecture is known as a ***Von Neumann machine***, named after John Von Neumann, one of the originators of the concept. With very few exceptions, nearly all modern computers follow this form.

Von Neumann computers are what can be termed *control-flow computers* since the steps taken by the computer are governed by the sequential control of a program: in other words, the computer follows a step-by-step program that governs its operation.

PROCESSOR

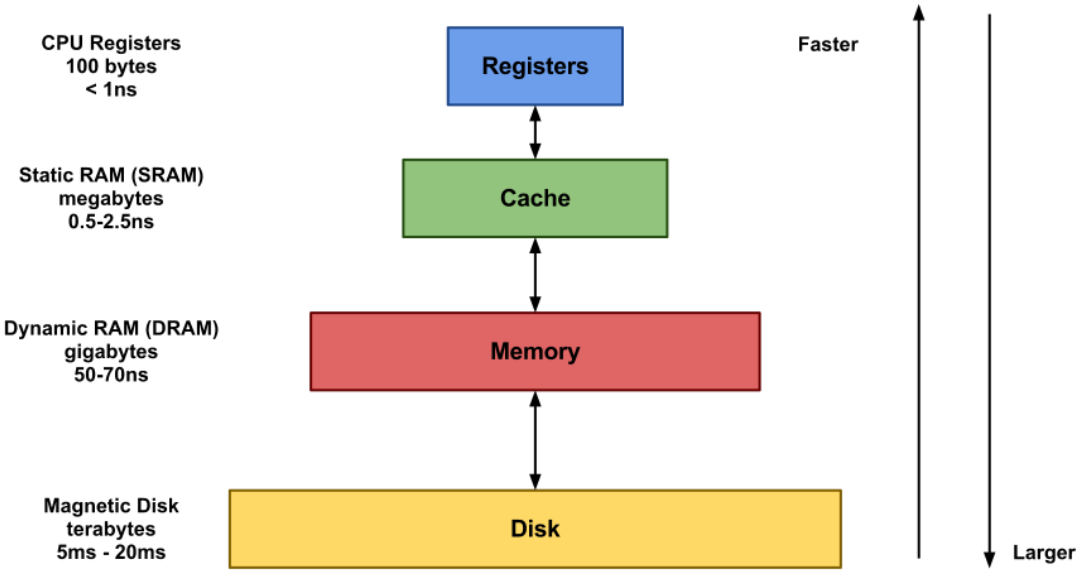
- At the heart of the computer is the processor, the hardware that carries out the instructions of a computer program, to perform the basic arithmetical, logical, and input/output operations of the system. Computer programs are made up by a sequence of instructions also known as *machine code*.
- Each type of processor has a different *instruction set*, meaning that the functionality of the instructions (and the bit patterns that activate them) varies.
- Instructions in a computer are numbers, just like data. Different numbers, when read and executed by a processor, cause different things to happen: the bit patterns of instructions feed into the execution unit of the processor. Different bit patterns activate or deactivate different parts of the processing core. Thus, the bit pattern of a given instruction may activate an addition operation, while another bit pattern may cause a byte to be stored to memory.

MICROPROCESSORS VS. MICROCONTROLLERS

- A ***microprocessor*** is a processor implemented (usually) on a single, integrated circuit. With the exception of those found in some large supercomputers, nearly all modern processors are microprocessors, and the two terms are often used interchangeably. A microprocessor is sometimes also known as a *CPU* (*Central Processing Unit*).
- A ***microcontroller*** is a processor, memory, and some I/O devices contained within a single, integrated circuit, and intended for use in embedded systems. The buses that interconnect the processor with its I/O exist within the same integrated circuit.

MEMORY

- The memory of the computer system contains both the instructions that the processor will execute and the data it will manipulate: instructions are read (fetched) from memory, while data is both read from and written to memory.
- The processor has no way of telling what is data or what is an instruction. If a number is to be executed by the processor, it is an instruction; if it is to be manipulated, it is data. Because of this lack of distinction, the processor is capable of changing its instructions (treating them as data) under program control.
- Each location in the memory space has a unique, sequential address. The address of a memory location is used to specify (and select) that location. The memory space is also known as the address space, and how that address space is partitioned between different memory and I/O devices is known as the memory map.
- Memory chips can be organized in two ways, either in word-organized or bit-organized schemes. In the word-organized scheme, complete nybbles, bytes, or words are stored within a single component, whereas with bit-organized memory, each bit of a byte or word is allocated to a separate component.
- It is worth to recall that we do not rely on a single memory component or technology but there are trade-offs among the three key characteristics of memory: cost, capacity and access time. The different technologies are represented in the following memory hierarchy:



CPU Registers: fast, but limited. It is usually an array of D flip-flops.

SRAM: commonly used on-chip as CPU cache. Data is stored in a pair of inverting gates.

DRAM: usually used as main memory (generically called RAM). Data is stored as a charge in a capacitor, thus requires being periodically refreshed (charge can last several milliseconds). Since it requires fewer transistors than SRAM, it can be packed denser.

DISK STORAGE: Disks are workhorse storage devices that hold enormous amounts of data, on the order of hundreds to thousands of gigabytes, as opposed to the hundreds or thousands of megabytes in a RAM-based memory. However, it takes on the order of milliseconds to read information from a disk, a hundred thousand times longer than from DRAM and a million times longer than from SRAM.

I/O MODULES

In addition to one or more processors and a set of memory modules, the third key element of a computer system is a set of I/O modules also known as *peripherals* used by the processor to communicate with the external world.

An I/O module is not simply mechanical connectors that wire a device into the system bus: it contains some "intelligence," that is, it contains logic for controlling the flow of data between the external device and the bus.

There are three main ways in which data may be exchanged with the external world:

- *Programmed I/O:* the processor accepts or delivers data at times convenient to it.
- *Interrupt-driven I/O:* external events control the processor by requesting the current program to be suspended and the external event be serviced. An external device will interrupt the processor (assert an interrupt control line into the processor), at which time the processor will suspend the current task (program) and begin executing an interrupt service routine. The service of an interrupt may involve transferring data from input to memory or from memory to output.
- *Direct Memory Access (DMA):* DMA allows data to be transferred from I/O devices to memory directly without the continuous involvement of the processor. DMA is used in high-speed systems, where the rate of data transfer is important. Not all processors support DMA.

SYSTEM INTERCONNECTIONS (BUSES)

- A bus is a physical group of signal lines that have a related function. Buses allow for the transfer of electrical signals between different parts of the computer system and thereby transfer information from one device to another. For example, the data bus is the group of signal lines that carry data between the processor and the various subsystems that comprise the computer.
- The "width" of a bus is the number of signal lines dedicated to transferring information: e.g. an 8-bit-wide bus transfers 8 bits of data in parallel.

- The majority of microprocessors available today (with some exceptions) use the three-bus system architecture made up of *address bus*, *data bus* and *control bus*. The *data bus* is bi-directional, the direction of transfer being determined by the processor. The *address bus* carries the address, which points to the location in memory that the processor is attempting to access. It is the job of external circuitry to determine in which external device a given memory location exists and to activate that device (address decoding). The *control bus* carries information from and back to the processor regarding the current access.

Useful links: link 1, link 2

C LANGUAGE

PROGRAMME

A programme is a sequence of instruction written in machine language

THE FIRST PROGRAMME

First programme in C will be very easy. It consists in printing on screen console "Hello world". Let's see the code:

```
/* hello.c */
#include <stdio.h>
int main (void) {
printf ("Hello world! \n");
return 0;
}
```

First row is a comment, which start with "/*" and ends with "*/ (or simply using //…//), useful for writing information about the programme; first row of the code is #include<stdio.h>. This instruction says to the calculator in the programme the functions defined in file "stdio.h" (STandarD Input/Output) are used for the code.

In "int main" block the code is executed. The "int" says that main() function return an integer number.

Now let's call printf() function defined in stdio.h file, which prints our message (written between "") on screen; "\n" character is used to go to a new line.

USE OF VARIABLES

Variables can contain numbers (integer, decimal in general), text character and so on…

TYPES OF VARIABLES

A variable has to be declared, in this way:

```
type variable_name;
```

We can assign an initial value, as well:

```
type variable_name = initial_value;
```

C language admits any type of variable, which are listed in the following table:

Type	**Description**	**Dimension (in bit)**
char	Text characters ASCII	8
short int	Little integer numbers (from -32768 to 32768)	16
unsigned short int	Little positive integer numbers (from 0 to 65536)	16
int	Integer numbers (from -2147483648 to 2147483648)	32
unsigned int	Integer positive numbers (from 0 to 4294967296)	32
long int	Big integer numbers	32
float	Floating point (single precision)	32
double	Floating point (double precision)	32

EXAMPLE

```
int a;                          // Declare an integer variable without initializing
int b = 3;          //Declare an integer variable B whose initial value is 3
char c = ‘q’;                   //Declare a char variable which contains character q
float d = 3.5;                  //Declare a float variable whose initial value is 3.5
a = 2;                          // Now the value 2 is associated to variable a
int e = a+b                     // ‘e’ is sum of a and b, so equal to 5
```

BASIC OPERATIONS ON VARIABLES

Any kind of basic mathematical operation can be computed with variables. Let’s see this code:

```
int a = 2;
float b = 3.5;
int c = (int) a+b;              //Resut is converted in int. In this case c = 5;
```

Another code:

```
int a = 2;
float b = 3.5;
float c = (float) a+b;                  // Convert the result in float. In this case c = 5.5
```

To update value of a variable there’re two ways to write code:

```
int a = a+2;                            // Update the value of a
```

or:

```
int a += 2;
```

PRINTING OF VARIABLES VALUES

It's possible to use variables in function as printf().

EXAMPLE

```
int x = 3;
printf ("x is equal to %d",x);
```

Output will be: 'x is equal to 3'

The format "%d" of the string says to the compiler to print the value in integer number: Instead, if we want to print a float variable:

```
float x = 3.14;
```

we have to put "%f".

ITERATIVE CYCLES

'FOR' INSTRUCTION

In order to repeat a block of instruction, 'for' cycle is very useful and has this syntax:

```
for (variable_1=value_1, ... , variable_n=value_n; condition; step) {
        code
}
```

- variable_1=value_1, ..., variable_n=value_n is the counter variable;
- condition is a Boolean condition which establish number of cycles;
- step is the increment or decrement to the counter variable at each cycle

EXAMPLE

```
int main() {
int i;                  // Counter variable
for (i = 0;  i< 10;  i++)
        printf ("Hello world!\n");
return 0;
}
```

The code is executed until the condition 'i<10' is true; the step is 'i++'. The programme prints "Hello world" for 10 times.

'For' cycle are very useful for manipulating array

'WHILE' INSTRUCTION

'While' cycles execute a block of instruction until a certain condition is true. The syntax is:

```
while (boolean_expression) {
        code
}
```

EXAMPLES

EX1

```
int i=0;
while (i<10) {
        printf ("Value of i: %d\n",i);
        i++;
}
```

EX2

```
while (n!=0) {
        printf ("Insert a number (0 to end): ");
        scanf ("%d",&n);
        printf ("Number: %d\n",n);
}
```

The programme will ask to insert an integer number and print the input number; if number is equal to 0, then cycle ends.

Firstly this kind of cycle verifies the condition and then executes the code.

'DO WHILE' INSTRUCTION

If we want to execute a code and then verify the condition, we can adopt 'do-while' cycle. The structure is the following:

```
do {
        code
        code
        ......
} while(boolean_condition);
```

EXAMPLE

```
int n = -1; // Variable int
do {
printf ("This code will be executed only once\n");
} while(n>0);
```

The program firstly executes printf() instruction, then verifies the specified condition.

ARRAY

Arrays are the most simple data structure, very similar to vectors in algebra. Position of an element in the array is identified by means an index. How to declare an array? Let's see:

```
type array_name[quantity];
```

EXAMPLE

```
int vector[10];
```

The array is named ‘vector’ and contains 10 elements; index start from 0 to 9 (not from 1 to 10!)

Now let’s give an example to highlights procedure for filling up array and for printing each element:

```
main() {
int vector[10];
int i;
for (i=0; i<10; i++) { // For i times...
printf ("Element n.%d: ",i); // Element n.i
scanf("%d",&mio_array[i]); // Read an int value from keyboard and save it in the i-th element

}
for (i=0; i<10; i++)
printf ("Element n.%d: %d\n",i,vector[i]); // Print all values contained in the array
}
```

MATRIX

In C language we can declare an array with two dimensions: matrices. So we declare a matrix as a monodimensional array, but specifying number of rows and columns:

```
int matrix[2][2]                //Declare a 2x2 matrix
```

The way for reading and writing on matrix elements is done by means of two indexes in order to manage rows and columns:

```
int matrix[2][2];
int i,j;

// Read input values of matrix
for (i=0; i<2; i++)
for (j=0; j<2; j++) {
printf ("Element [%d][%d]: ",i+1,j+1);
```

```
scanf ("%d",&matrix[i][j]);
}
// Print values of matrix
for (i=0; i<2; i++)
for (j=0; j<2; j++)
printf ("Element in position[%d][%d]: %d\n",i+1,j+1,matrix[i][j]);
```

Useful links:
https://www.tutorialspoint.com/cprogramming/c_quick_guide.htm

FUNCTIONS IN C

A common way of writing code is splitting complex algorithms into self-contained parts with conceptual relevance which are called FUNCTIONS. Functions make your program more understandable and they also mean re-usability: some pieces of code can be re-used in several places which reduces both the total quantity of software and the possibility of bugs.

A function is a *module* of code that takes information in (referring to that information with local symbolic names called parameters), does some computation, and (usually) returns a new piece of information based on the parameters. In order for one function to "see" –use– and thus call another function, the "prototype" of the function must be seen in the file before the usage.

REMARKS:

- The order of functions inside a file is arbitrary: it does not matter if you put function one at the top of the file and function two at the bottom, or vice versa but being ordered will surely be good for code understanding.
- C functions must be TYPED i.e. the return type and the type of all parameters must be specified. In C, all functions must be written to return a specific TYPE of information and to take in specific types of data (parameters). This information is communicated to the compiler via a ***function prototype***.

FUNCTION PROTOTYPE

A Prototype can occur at the top of a C source code file to describe what the function returns and what it takes (return type and parameter list).

Here's an EXAMPLE:

```
double myfunction(int n);
```

This prototype specifies that in this program, there is a function named "myfunction" which takes a single integer argument "n" and returns a double.

Thus the function prototype general form is:

```
return_type name_of_function (p_type p_name, … …, … …);
```

With:

- return_type − A function may return a value. The return_type is the data type of the value the function returns. Some functions perform the desired operations without returning a value: in this case the return_type is the keyword *void*.

- name_of_function − This is the actual name of the function.
- parameters list − A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. The parameter list refers to the type, order, and number of the parameters of a function. Keep in mind that parameters are optional i.e. a function may not require parameters.

REMARK:

- Having the prototype available before the first use of the function allows the compiler to check that the correct number and type of arguments are used in the function call and that the returned value, if any, is being used reasonably.
- Elsewhere in the program a function definition must be provided if one wishes to use this function.

FUNCTION DEFINITION

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

```
return_type name_of_function (p_type p_name, p_type p_name, …){
        body of the function
}
```

A function definition in C programming consists of a function *header* and a *function body*: the header is exactly as the function prototype while the function body contains a collection of statements that define what the function does.

EXAMPLE

Here's a function example:

```
/* function returning the max between two numbers */
int max(int num1, int num2) {
/* local variable declaration */
int result;
        if (num1 > num2)
                result = num1;
        else
                result = num2;
        return result;
}
```

THE RETURN STATEMENT

When a line of code in a function that says: "return X;" is executed, the function "ends" and no more code in the function is executed. The value of X (or the value in the variable represented by X) becomes the result of the function.

REMARK: If the return_type is “void”, it is sufficient to write “ return; “

THE MAIN FUNCTION

In C, the "main" function is treated the same as every function: it has a return type and in some cases accepts inputs via parameters. The only difference is that the main function is "called" by the operating system when the user runs the program. Thus the main function is always the first code executed when a program starts. To use another function within the “main”, you will have to *call* that function to perform the defined task.

FUNCTION CALL

When a program calls a function, the program control is transferred to the called function. A called function performs a defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns the program control back to the main program. To call a function, you simply need to pass the required parameters along with the function name, and if the function returns a value, then you can store the returned value.

When one piece of code invokes or calls a function, it is done by the following syntax:

```
variable = function_name ( parameters list);
```

EXAMPLE

```
#include <stdio.h>
 /* function declaration */
int max(int num1, int num2);
 int main () {
   /* local variable definition */
   int a = 100;
   int b = 200;
   int ret;
    /* calling a function to get max value */
   ret = max(a, b);
    printf("Max value is : %d\n", ret);
    return 0;
}
```

Where max is the function we defined in the previous example.

PARAMETERS IN C FUNCTIONS

A Parameter is the symbolic name for "data" that goes into a function; a parameter (*or formal parameter*) is a characteristic of a function while an argument (or *actual parameter*) is a characteristic of a function call. A parameter exists for a function, even without enclosing source code while the argument exists only in a running program when a call to the function is made.

There are two ways to pass parameters in C:

1. Pass by Value
2. Pass by Reference.

PASS BY VALUE

Passing a parameter by value means that, when a parameter is passed in the function call, a copy of the data is made and stored in the parameter name in the function definition: any changes to the parameter in the function definition have **NO** effect on the data in the calling function.

EXAMPLE

In C, the default is to pass by value. For example:

```
int main()      // Test function for passing by value (i.e., making a copy)
{
  int z = 27;
  pass_by_v( z );          // z is the function argument
  printf("z is now %d\n", z);

  return 0;
}

void pass_by_v( int x )    // C function using pass by value. (Notice no &)
{
    x = 5;
}
```

The output is:

```
z is now 27
```

REMARK:

The call-by-value argument ceases to exist with the end of the function call, unless assigned to a parameter that survives the end of the function call or returned as the return value of the function call.

PASS BY REFERENCE

A reference parameter "refers" to the original data in the calling function: **any changes made to the parameter are also made to the original variable**.

There are two ways to make a pass by reference parameter:

1. The ampersand (**&**) used in the function prototype:

```
function (& p_name)
```

To make a normal parameter into a passed by reference parameter, we use the "& p_name" notation, which specifies the address of the variable i.e. the location in memory where a variable stores its data or value.

REMARK: the ampersand operator is only used with variables, not with constants.

2. Arrays: **arrays are always passed by reference in C**, they do not use the '&' notation. Any change made to the parameter containing the array will change the value of the original array.

REMARK:

- To protect from accidentally changing a reference parameter, when we really want it not to be changed (we just want to save time/memory) we can use the C keyword const**.**
- Reference parameters are used to make programs more "efficient". Consider passing in a structure as a parameter: if the structure is very big, and we copy all of it, then we are using a lot of unnecessary memory.

EXAMPLE

```
int main()        // Test function for passing by reference (i.e. passing variable address)
    {
      int z = 27;
      pass_by_r(& z );
      printf("z is now %d\n", z);
 return 0;
    }
void pass_by_r( int* x )
    {
        *x = 5;
    }
```

The output is now:

```
z is now 5
```

Note that, in general, “type *” represent a pointer to that type: the address of a variable is a pointer to the memory where the variable is stored. To better understand the meaning of “*” (deference operator), consider the following EXAMPLE:

```
int x;
int *p;  // “*”  is used in the declaration:  p is a pointer to an integer, since (after dereferencing),
         // *p is an integer
x = 0;
// now x == 0
p = &x;  // &x is the address of x
```

```
// now p == &x, so *p == x
*p = 1;  // equivalent to x = 1, since *p == x
// now *p == 1 and *p == x, so x == 1
```

Finally here's another EXAMPLE:

```
#include <stdio.h>
void swap(int *x, int *y);  /* function declaration */
 int main () {
   /* local variable definition */
   int a = 100;
   int b = 200;
   printf("Before swap, value of a : %d\n", a );
   printf("Before swap, value of b : %d\n", b );
   swap(&a, &b);    // calling a function to swap the values.
   /* &a indicates pointer to a i.e. address of variable a and &b indicates pointer to b i.e. address of variable
b. */
   printf("After swap, value of a : %d\n", a );
   printf("After swap, value of b : %d\n", b );
   return 0;
}
/* function definition to swap the values */
void swap(int *x, int *y) {
   int temp;
   temp = *x;    /* save the value at address x */
   *x = *y;      /* put y into x */
   *y = temp;    /* put temp into y */
  return;
}
```

Output is:

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :200
After swap, value of b :100
```

Useful links: : link 1, link 2

THE C PROGRAM BUILDING PROCESS

Normally the C program building process involves **four stages** and utilizes different 'tools' such as a pre-processor, compiler, assembler, and linker.

PREPROCESSING

The C **pre-processor** (CPP) takes lines beginning with '#' as directives i.e. language constructs that specifies how a compiler (or other translator) should process its input (is just a text substitution tool and it instructs the compiler to do required pre-processing before the actual compilation). Before interpreting commands, the pre-processor does some initial processing which consist of joining continued lines (lines ending with a \ used to continue a macro that is too long for a single line) and stripping comments. In general, it processes include-files, conditional compilation instructions and macros.

REMARK:

All pre-processor commands begin with a hash symbol (#).

Here are some EXAMPLE directives:

#define	Substitutes a pre-processor macro.
#include	Inserts a particular header from another file.
#ifdef	Returns true if this macro is defined.
#ifndef	Returns true if this macro is not defined.
#endif	Ends pre-processor conditional.

The CPP will produce the contents of the header file (.h) joined with the contents of the source code file (.c).

COMPILING

In this second stage, the pre-processed code is translated by a **compiler** into assembly instructions specific to the target processor architecture. These form an intermediate human readable language.

The result of the compilation stage is a file (.s) containing the generated assembly instructions the compiler merely produces the machine language instructions that correspond to the source code file that was compiled.

REMARK:
The compiler first parses (or analyses) all of the language statements syntactically and then, in one or more successive stages, builds the output code. If there are syntactical errors in the source code, we call them *compilation errors*.

ASSEMBLY

During the assembly stage, an **assembler** is used to translate the assembly instructions to *machine code*. The output consists of a file (.o) containing the actual instructions to be run by the target processor (*object code*).

REMARK:
the assembler does not assign absolute memory locations to all the instructions and data in a file. Rather, it writes some notes in the object file about how it assumed things were laid out. It is the job of the linker to use these notes to assign absolute memory locations to everything and resolve any unresolved references.

LINKING

Very often, a real program that does anything useful will need to reference other files. In C a simple program to print your name to the screen would consist of:

```
printf ("Hello!\n");
```

In the object file there is a simple reference to the printf function which will be resolved by the linker. Most programming languages have a standard library of routines to cover the basic stuff expected from that language: the linker links your object file with this standard library. Obviously you can create other .o files that have functions that can be called by another .o file.

Linking is the final stage of C code building: it takes one or more object files or libraries as input and combines them to produce a single file.

The result of this stage is the final executable program (.exe).

REMARK:
if the linker does not find a function called as the reference in the object file, the linking process fails with an error.

SUMMARY

	Input	Output

Editor	Program typed from keyboard	C source code containing program and pre-processor commands
1. Pre-processor	C source code file	Source code file with the pre-processing commands properly sorted out.
2. Compiler	Source code file with pre-processing commands sorted out	Assembly language code
3. Assembler	Assembly language code	Object code in machine language
4. Linker	Object code of our program and object code of library functions	Executable code in machine language
Loader	Executable file	

Useful links: link 1, link 2

ADDITIONAL MATERIAL

STRUCTURES IN C

The structure in C is a user defined data type available in C that allows to combine data items of different kinds: it defines a physically grouped list of variables to be placed under one name in a block of memory.

The general syntax for a struct declaration in C is, for EXAMPLE:

```
struct structure_tag{
        type member1;
        type member2;
        ... /* other members*/
  }; /* note semi-colon here */
```

To define variables of structure type, you must use the **struct** statement, e.g. having the following structure,

```
struct point{
        float x;
        float y;
        float z;
}
```

a new structure variable can be defined as:

```
struct point p1;
```

Remarks:

- *typedef* allows you to declare instances of a struct without using the keyword "struct":

```
typedef struct{
        float x;
  float y;
  float z;
} point;
```

point is now a type that can be declared without "struct" in front of it:

```
point p1;
```

- Structures are generally defined along with function prototypes thus at the top of .c files (in header files in more complex C programs).

ACCESSING STRUCTURE MEMBERS

To access any member of a structure, we use the *member access operator* "**.**": it is coded as a period between the structure variable name and the structure member that we wish to access.

With reference to the previous EXAMPLE, to access and assign a value to the "point" structure members we simply write the following:

```
p1.x = 5;
p1.y = 0.5;
p1.z = 1.4;
```

Remark: suppose we define a pointer to the "p1" structure i.e.

```
struct point* ptr; // or point* ptr with the use of typedef
ptr = &p1;
```

In this case you can access and assign values to the members of the structure through the following syntax:

```
ptr-> x = 5;
/* Note is the same as (*ptr).x = 5 */
```

Useful links: link 1, link 2, link 3

DYNAMIC MEMORY ALLOCATION: ARRAYS

We have seen that arrays and matrices can be declared as such:

```
type array_name[d];
type matrix[r][c];
```

where the d,r,c are integer values which define the array/matrix dimensions: in this way the size of the array (or matrix) is fixed at compile time. This is useful in case the length of an array is known a priori but if the required size is not known until run-time (for example, if data of arbitrary size is being read from the user or from a disk file), then using fixed-size data objects is inadequate and here's where dynamic allocation enters the picture.

THE *malloc* FUNCTION

The idea is to define a pointer to a variable of the desired type e.g.:

```
int* myArray;
```

and then, for example, the user is prompted to input a size (for the dynamically allocated array) which is stored as such:

```
int size;
scanf("%d", &size);
```

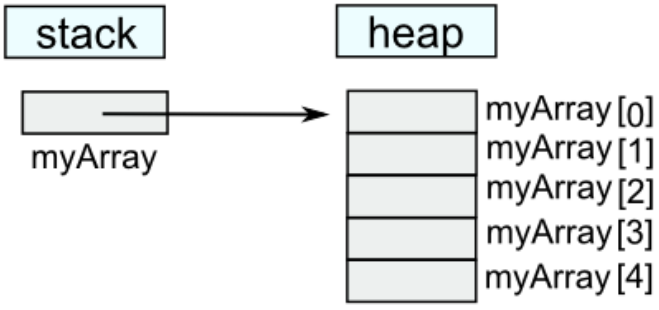
at this point an integer array is created using the defined pointer, the size and the **malloc** function:

```
myArray = (int*)malloc(sizeof(int)*size);
```

- The library function *malloc* is used to allocate a block of memory on the heap, an area of memory structured for this purpose.
- The *sizeof()* operator generates the size of the datatype.
- *malloc* returns a void pointer (void *), which indicates that it is a pointer to a region of unknown data type since it allocates based on byte count but not on type: a "cast" to int is performed on the returned pointer ([…] (int*)malloc[…]).

Remark: once the array has been allocated it can be treated exactly as a statically allocated array.

The operations that we are performing (choosing size = 5) can be easily visualized through the following graphical representation:



THE *free* FUNCTION

When the memory is no longer needed, the pointer is passed to **free** which deallocates the memory so that it can be used for other purposes:

```
free(myArray);
```

Useful links: link 1, link 2, link 3

FILES

C environment gives the chance to create, open, close text or binary files for their data storage. A set of most important calls for file management are listed here.

OPENING FILES

In order to create or to open an existing file, the **fopen()** function is used. The prototype of this function is:

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

Where:

- **filename** is the name of your file;
- **mode** concern the opening process and can have one of the following values:
  - **r**: reading purpose;
  - **w**: writing purpose. If the file doesn't exist, then a new file is created;
  - **a**: writing in appending mode. If the file doesn't exist, then a new file is created;

Other modes are possible, but here only the most important ones are listed.

CLOSING FILES

In order to close a file, the **fclose()** function is used. The prototype of this function is:

```
int fclose( FILE *fp );
```

fclose() function returns two values:

- **0** in success case;
- **EOF** if there's an error in closing file. This constant is defined in the header file *stdio.h*

WRITING FILES

There are many functions in order to write in a file. Anyway, the prototype of this function is:

```
int fputc( int c, FILE *fp );
```

and

```
int fputs( const char *s, FILE *fp );
```

- **fputc()** is used for writing a character value of the argument c to the output stream referenced by fp.
- **fputs()** is used for writing strings s to the output. You can also use **int fprintf(FILE *fp,const char *format, ...)** in order to write a string into a file.

READING FILES

There are many functions in order to read from a file. Anyway, the prototype of this function is:

```
int fgetc( FILE * fp );
```

and

```
char *fgets( char *buf, int n, FILE *fp );
```

- **fgetc()** is used for reading a character from the input file referenced by fp.

- **fgets()** is used for reading a string with n-1 characters (last cell of string is reserved for NULL character to terminate the string. You can also use **int fscanf(FILE *fp, const char *format, ...)** function to read strings from a file.

EXAMPLE:

```
#include <stdio.h>

main() {

   FILE *fp;
   char var[255];

   fp = fopen("/tmp/test.txt", "r");        %In 'filename' the path of directory in which  there's your file is specified. In this case, you want to open the file named 'test.txt'
   fscanf(fp, "%s", var);            %Reads the string from file
   printf("1 : %s\n", var );         %Prints the string

   fgets(var, 255, (FILE*)fp);
   printf("2: %s\n", var );

   fgets(buff, 255, (FILE*)fp);
   printf("3: %s\n", var );
   fclose(fp);

}
```

Once the code is executed the following output is produced:

```
1 : This
2: is a test for fprintf...

3: This is a test for fputs...
```

The **fscanf()** reads until encounter a space, so it reads only 'This'; **fgets()** reads line completely until encounter end of line.

Useful links: http://www.w3resource.com/c-programming-exercises/file-handling/

From: http://www.tutorialspoint.com/cprogramming/c_file_io.htm