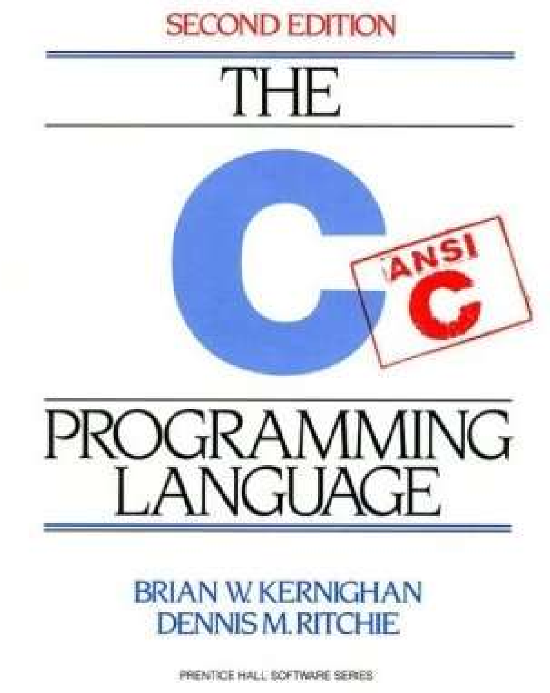
Subject 4
INTRODUCTION TO C LANGUAGE

Programming

What is C language?

- I'ts a general purpose language invented at Bell-Labs by Dennis Ritchie between 1969 and 1973
- Currently is one of the more popular languages
- The C language syntax is a base for other language
  - C++,C#, D, Go, Java, JavaScript, Limbo, LPC, Perl, PHP, Python,
- Was designed for portability and cross-platform
- There is an ISO standard ISO/IEC 9899:2018

The MUST read



http://www.amazon.com/C-Programming-Language-2nd-Edition/dp/0131103628

Advantage over assembly ?

- Allows to work in a higher level of abstraction
- Programs are more legible
- Ease of maintenance
- Programs can be ported to other microcontroller
- Programmers are more productive
- The learning curve is lower
- Today is one of the most popular for embedded systems
- Many free source code available.

Disadvantages?

- It hides complexity and in some application this could be dangerous
- Is non predictive regarding the time it takes to executes a sequence of instructions (depends on the complier, optimization, etc).
- Cannot be used in microcontroller with low ROM and RAM resources.
- In embedded applications is important to check how efficiently the code is generated.
- There will be always a tradeoff between program size and speed.
- The developers require skills very different than desktop or even mobile application programmers.

Programs in C

- Are constructed by functions and variables
- Functions
  - Contain the statements of the actions to execute
    - Declarations
    - Assignments
    - Aritmetic
    - Function calls
    - Control
    - Null
- Variables
  - Store the values that will be used in the computations

```
/*================================================
  = PROGRAM WRITTEN IN C
  = TEMA_04_1.C
  ================================================*/

#include<stdio.h>                    /*Function prototypes for stdio*/
                                     /*and other definitios */

/* Program starts here */

main(void)
{                                    /*Main is also afunction in this case */
                                     /*with a void argument*/
int a,b,c,i;                         /*Define 3 integer type vars */
    a=3;                             /*Assigns a 3 to variabie a*/
    b=5;                             /*Assigns a 5 to variable b*/
    printf( "A = %d B= %d\n",a,b);   /*Prints the value of a and b*/
    c=a+b;                           /*Add a withb and assignt to c */
    printf("a + b = %d\n",c);        /*Print variable c */
    return;                          /*Return to the operating system*/
}                                    /*Main function end bracket */
```

Header Files

- The #include<file.h> directive is used to tell the compiler the header files that will be used in the programs
- Header files like “stdio.h” contain the function prototypes and variable definitions that will be used in the program when they require to access a library another module of the same program
- In C language, functions and variables must be declared before they are called on a statement

Variables

- Variables in C language must be declared before they are used on a statement. They are formed of:
  - Name
  - Type
  - Initial value [Optional]

Variable names

- Must start with a letter
- The name can contain number but cannot use:
  - Arithmetic signs, points, apostrophes, C reserved words or special symbols (#,@;?...etc)
- Adding the "_" character between words gives the name more legibility
  - MyIntegerVariable → my_integer_variable
  - Avoid _name o __name

The underscore character is used traditionally to code libraries, so is possible to generate conflicts with the compiler or linker

Variable Types

- It defines the size and type of storage or the variable in the main types are:
  - void Specifies an empty values
  - char Character (a character or a byte)
  - int Integer (two bytes)
  - long Long (four bytes)
  - float Floating point single press (four bytes)
  - double Floating point double press (eight bytes)
  - _bool Boolean (C99) (size is variable)**

**Avoid since not all compilers implement this

Variable declaration

- Variables must be declared before used, the declaration specifies the type and a single or a list of variables of the same type.

```
int i,j,k;
char my_char1,my_char2;
```

- Variables can be initialized during the declaration:

```
int i = 0;
char mi_char = 'a';
```

Constants

- There are five basic type of constants

`strings:`	`“Hello world”`
`char:`	`'a'`
`integer:`	`1234`
`float:`	`1234.345`
`long:`	`98769876L`

Radix

- Like in assembly language in C you can specify a number written in an specific radix (numeric base) to ease the legibility

RADIX	PREFIX	EXAMPLE
Decimal	None	1234
Octal	O	O4723
Hex	0x	0x2A
Float	None	0.12 or 1.2e-1

Arithmetic Operators

- There are 7 basic arithmetic operators

`+`	`Addition`
`-`	`Subtraction`
`*`	`Multiplication`
`/`	`Division`
`%`	`Modulus (remainder)`
`++`	`Increment by one`
`--`	`Decrement by one`

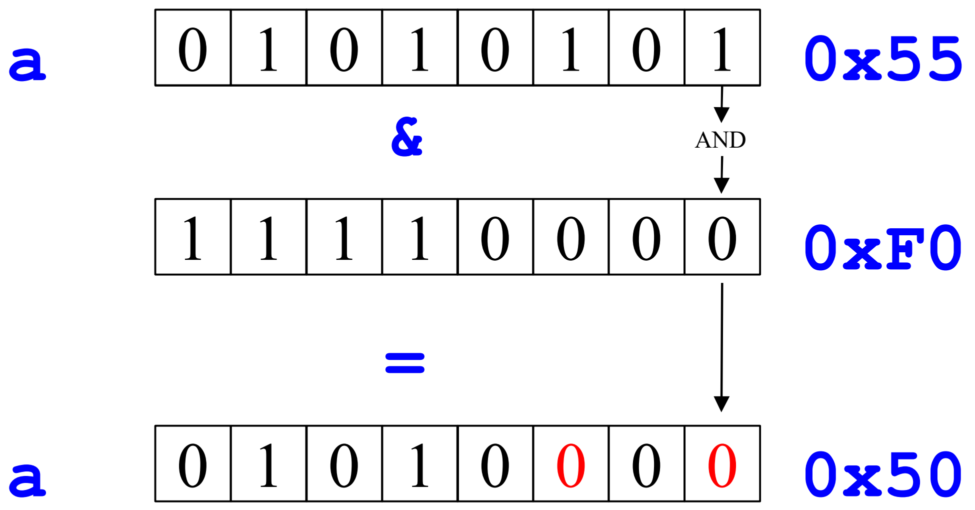
Bitwise operations

- C provides operator to manipulate variables at the bit level and can be applied to char,int,long (signed or not signed)

`&`	**AND**
`|`	**OR**
`^`	**XOR**
`~`	**NOT**
`>>`	**RIGHT SHIFT**
`<<`	**LEFT SHIFT**

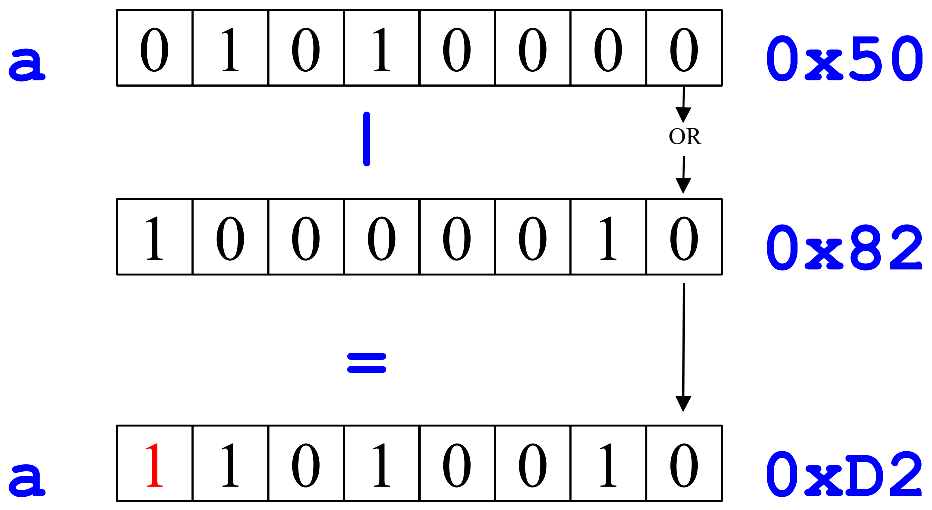
Example of bitwise operations

```
unsigned char a = 0x55;
     a = a & 0x0F0;
     a = a | 0x082;
```



Example of bitwise operations

```
unsigned char a = 0x55;
        a = a & 0x0F0;
        a = a | 0x082;
```



Example of bitwise operations

```
//Turn on bit 0 (LSB) of PORTB
PORTB = PORTB | 0x01

//Turn off bit 1 of PORTB
PORTB = PORTB & 0xFD  //0xFD = 11111101

//Shift left the variable xyz
xyz = xyz<<2
```

Alternative notation when the operator is the same after the = sign

```
//Enciende bit 0 de puerto B
PORTB = PORTB | 0x01
//Can also be written like:
PORTB |= 0x01

//Turn off bit 1 of PORTB
PORTB = PORTB & 0xFD //0xFE = 11111101
//Can also be written like:
PORTB &= 0xFD
```

Expressions in C language can be written in several ways we must favor clarity over efficiency in coding, ergo is more readable
PORTB = PORTB | 0x01 than PORTB |= 0x01
http://www.ioccc.org/

Relational and logic operators

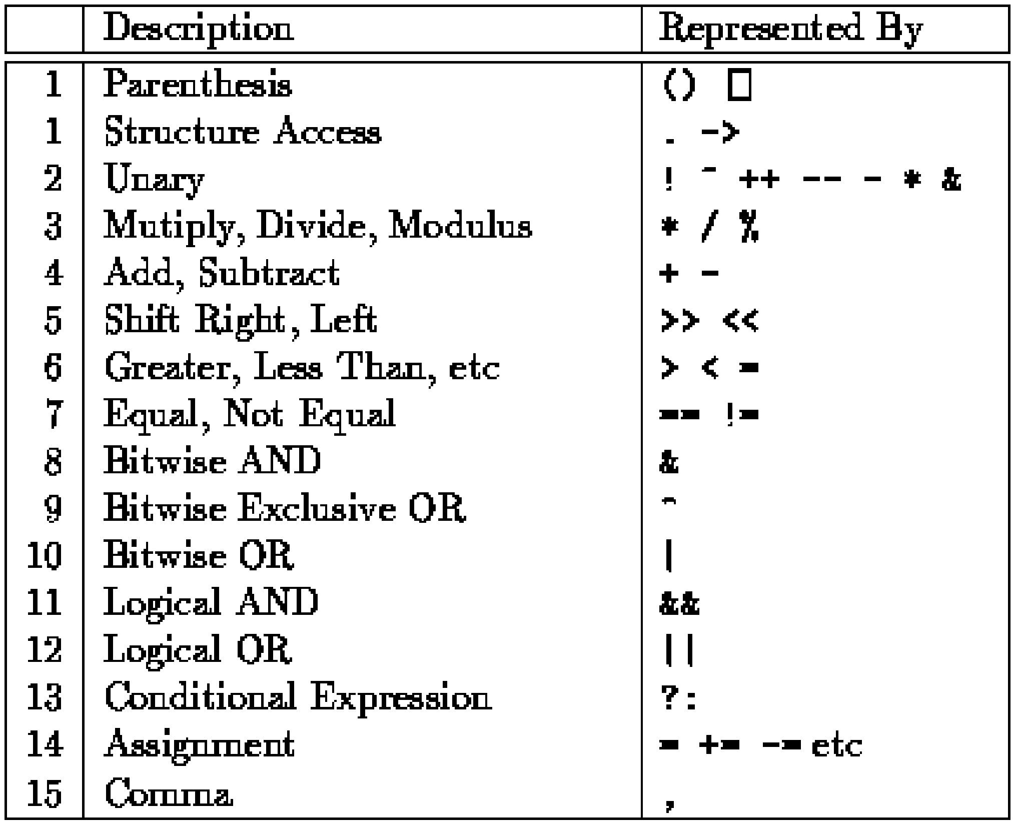
- The relational operators are used in expressions to compare between two operands, if the comparison is true the result is 0 if is false, is different than 0 if true

`==`	equal than
`!=`	different than
`>`	more than
`>=`	less or equal than
`<`	less than
`<=`	less or equal than
`&&`	And (logic)
`||`	Or (logic)
`!`	Not (logic)

Example of relational and logic

```
If( !(PORTB & 0x80)){statements}
//If bit 7 is not one execute statements

If( i > 0 && j < 10)) {statements}
//If i greater than 0 and j less than 10
  execute statements

If(A == B ){statements}
//If A equal B execute statements
```

Precedence of operators in C



	Description	Represented By
1	Parenthesis	() []
1	Structure Access	. ->
2	Unary	! ~ ++ -- - * &
3	Mutiply, Divide, Modulus	* / %
4	Add, Subtract	+ -
5	Shift Right, Left	>> <<
6	Greater, Less Than, etc	> < =
7	Equal, Not Equal	== !=
8	Bitwise AND	&
9	Bitwise Exclusive OR	^
10	Bitwise OR	|
11	Logical AND	&&
12	Logical OR	||
13	Conditional Expression	?:
14	Assignment	= += -= etc
15	Comma	,

Flow control statements (if)

- The if is conditional, if the condition is evaluated as "true" (different than 0), the preceding statements are executed
  - `if(expression){statements}`

- Example

  -
    ```
    if(a > b){
            sum = sum + 3; //Add 3 to sum
    }
    ```

Flow control (if-else)

- The if-else allow to execute two sets of statements, one for true (different than 0) or false (0)
  - `if(expression){statements}else{statements}`

- Example

```
• if(a!=0){
        r = b;
  } else {
        r = c;
  }
```

It can be replaced by the ? statement as follows
r=(a!=0) ? b:c AVOID THIS OR USE AT YUR OWN RISK

Flow control (switch)

- The switch statement is a multi option based on the value of the control expresson

  -
    ```
    switch(expresion){
      case const1: {statements 1
                    }break;
      case const2: {statements 2
                    }break;
      case constn: {statements n
                    }break;
      default:   {statements if none}
    }
    ```

Flow control (switch)

- Example for switch
  - `switch(i+1){`

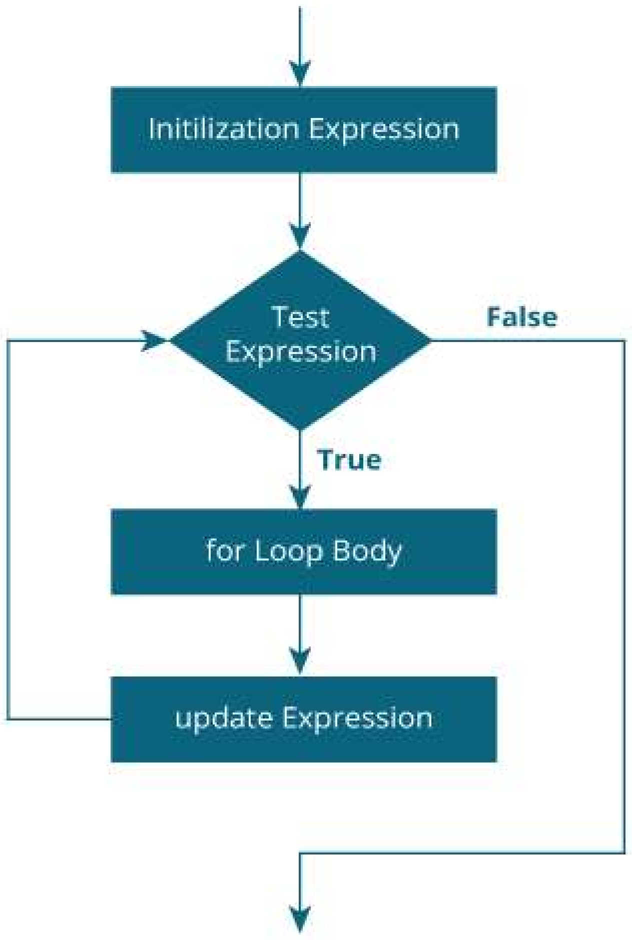
```
      case 1:    printf("Option A\n");
                  break;
      case 2:    printf("Option B\n");
                 break;
      case 3:    printf("Option C\n");
                 break;
      default:   printf("Invalid\n");
  }
```

Flow control (for-loop)

- Executes a controlled cycle with the following structure

```
for (initializationStatement; testExpression; updateStatement)
        {
            // statements inside the body of loop
        }
```

Flow control (for-loop)

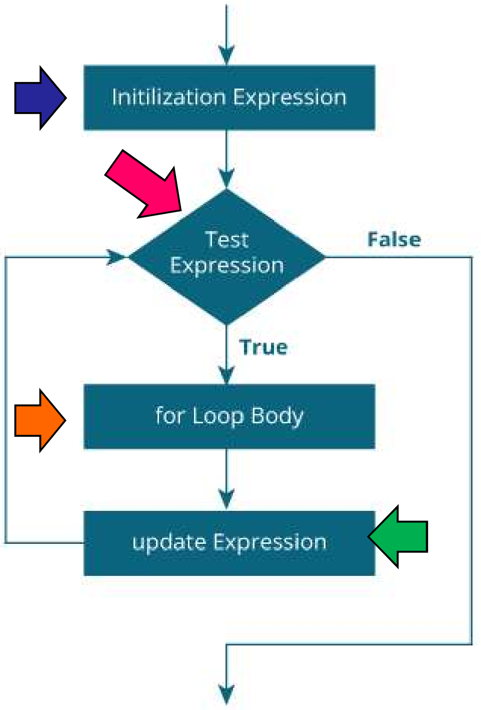


Fuente: https://www.programiz.com/c-programming/

Flow control (for-loop)

- Example, add the squared value of numbers 1 to 9

```
sum = 0;
for(i=1;i<10;i++){
   sum = sum + i*i;
   }
```

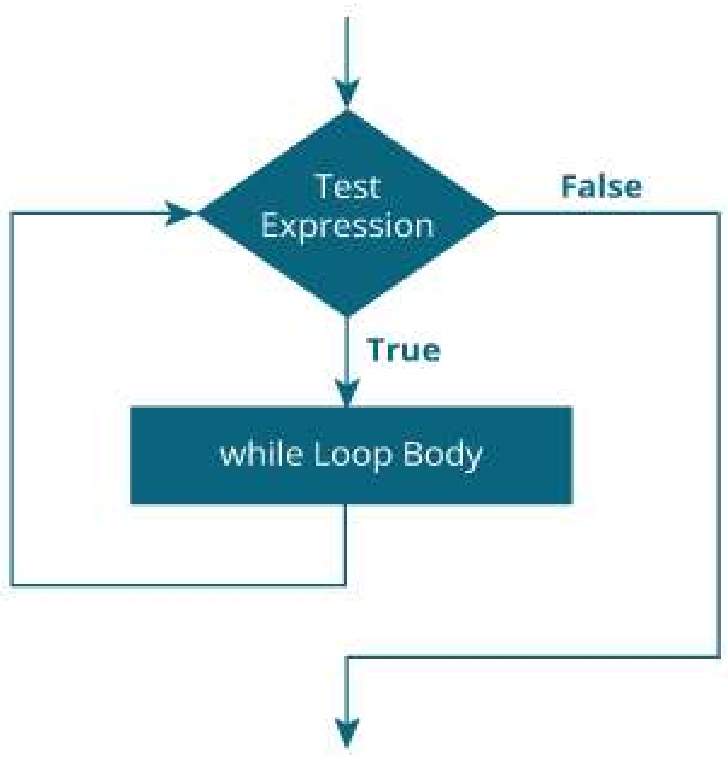


Flow control (while)

- While the evaluated expression is different than 0 the it will execute the statements. Each cycle is evaluated and it will continue until the evaluated expression is 0.

```
while(test_xpression){statements};
```

Flow control (while-loop)



Fuente: https://www.programiz.com/c-programming/

Flow control (while)

- Example: while the PORTA = 0x01, blink the LED connected at bit 0 of PORTB

```
while(PORTA == 0x01){
        PORTB = PORTB | 0x01;
        PORTB = PORTB & 0xFE;
};
```

- Blink the LED till the end of times.

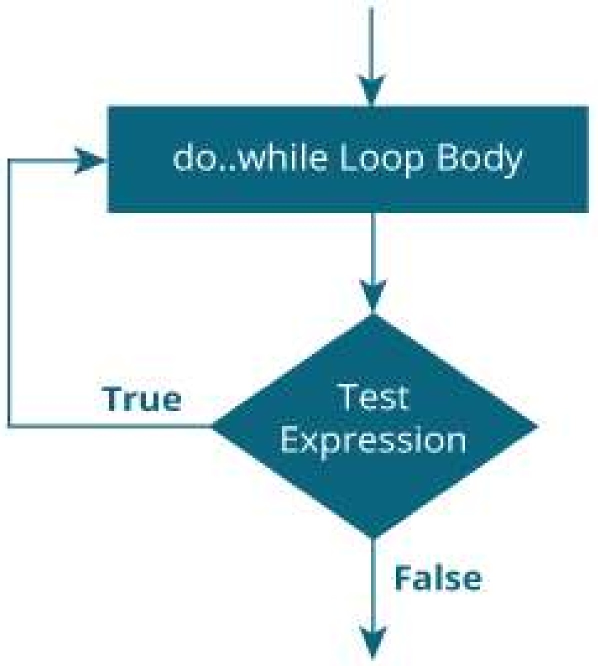
```
while(1){
        PORTB = PORTB | 0x01;
        PORTB = PORTB & 0xFE;
};
```

Flow control (do-while)

- The “while” verifies the expression before executing the statements, the “do-while” verifies the expression till the end, so the statements are performed at least one time.

```
do{
        statements
        }while(test_xpression);
```

Flow control (do-while)

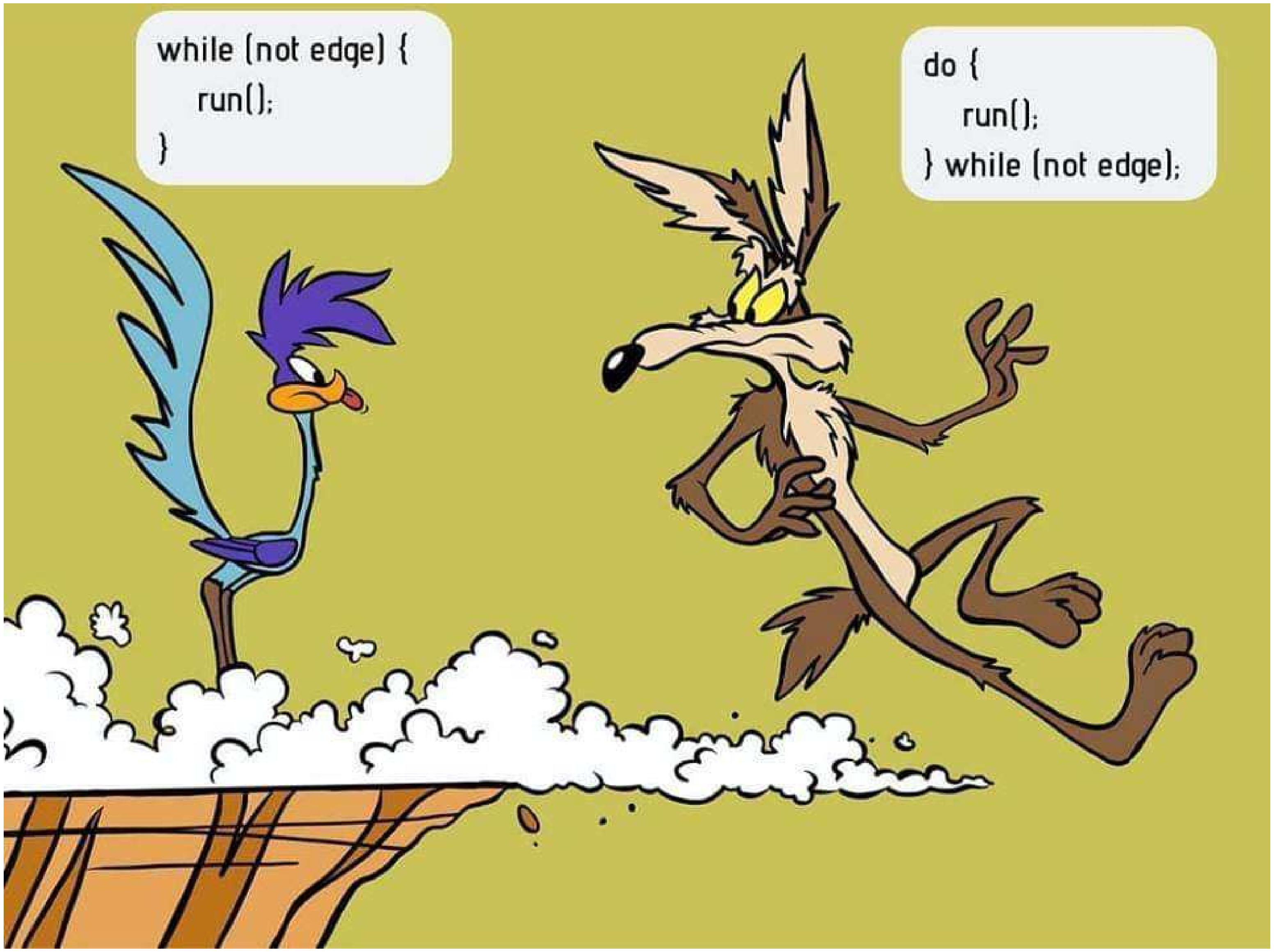


Fuente: https://www.programiz.com/c-programming/

Flow control (do-while)

- Blink the led at least once and continue to blink if PORTA = 0x01

```
• do{
       PORTB = PORTB | 0x01;
       PORTB = PORTB & 0xFE;
  }while(PORTA==0x01);
```


while (not edge) {
run();
}
do {
run();
} while (not edge);

Flow control (break)

•Is used inside a loop, when is executed, it breaks the loop to continue with the program flow.

```
while(1){
   printf("Whait\n");
   if(PORTA == 0x01)break;
}
printf("Port was = 0x01\n");
```

Flow control (cotinue)

- When in a loop continue forces the loop to go to de end with out performing the following statements but the loop continues.

```
for(i=0;i<10;i++{
    if(i == 4)continue;
    printf(“Iteración %i”,i);
 }
```

Flow control (goto)

- The goto goes directly to the statement to a label in the program, the label is placed in the goto statement and in the code where it will be directed.

```
goto my_label;
```

- It is recommened to AVOID its use because it breaks with the rules or structured programming generating “spaghetti” code.

Flow control (goto)

- Example:

```
if (i> 100) goto fatal_error;
****
****
fatal_error: printf("WTF??\n");
return(1);
```

Input and Output from user

- User input and output features are not part or the C language
- Since the philosophy of C is to use functions as the building blocks, the language provides a set of standard function libraries associated with IO (keyboard, screen,files) STDIO

Basic Input/Output

- int getchar()
  - Returns the character available on the standard input device (keyboard, serial port, etc)

```
char xchr;
xchr = getchar();
```

- int putchar(int)
  - Outputs a character to the standard output device (screen, serial port, etc)

```
putchar('a');
```

Basic Input/Output

- int puts(const char *s)
  - Ourputs a string pointed by s

```
puts("Curso de microcontroladores\n");
```

- int printf(format, arg1,arg2....argn)
  - Prints with a specific format.

```
printf("Valor = %d",val_temp);
```

- scanf(format, arg1,arg2....argn)
  - Captures an input with specific formats.

```
scanf("%d",&val_temp);
```

Format specifiers

- `%c` **character**
- `%d` **decimal (integer) number (base 10)**
- `%e` **exponential floating-point number**
- `%f` **floating-point number**
- `%i` **integer (base 10)**
- `%o` **octal number (base 8)**
- `%s` **a string of characters**
- `%u` **unsigned decimal (integer) number**
- `%x` **number in hexadecimal (base 16)**
- `%%` **print a percent sign**
- `\%` **print a percent sign**

Functions

- The programs in C use functions as building blocks (Functional programming)
- A function cannot be declared within a function.
- The same function can be called any time and any place on the same program.
- Usually processes the information provided by the caller and returns a value to the caller

Functions

- The information is give to the function using arguments
- The output of the function is transferred to the caller using the “return” statement.
- Functions can accept null or void parameters and also return nothing (void).
- Functions can be invoked by themselves (recursion).

Functions

- Function declaration

```
type function_name(argument1,...)
{
        statements
}
```

Funtions

- The function return type
- Can be any valid type (char,int,long,float etc..)

```
int function1(,,,)        //Returns type int
float function2(,,,)     //Returns type float
void function3(,,,)      //Returns nothing
```

- The function name
  - Any name that follow the naming rules of variables.

Funtions

- The argument list
  - Is the way to pass values to the functions, can be constants, variables or pointers.

```
long cube(long x)
void function1(int x, float y, char z)
void function2(void)
```

Functions

- Local variables
  - They are declared inside the function so they are private, only the function knows the name so the names can be repeated in other functions.

```
int function1(int y)
{
   int a, b=10;
   char my_char;
   ….
}
```

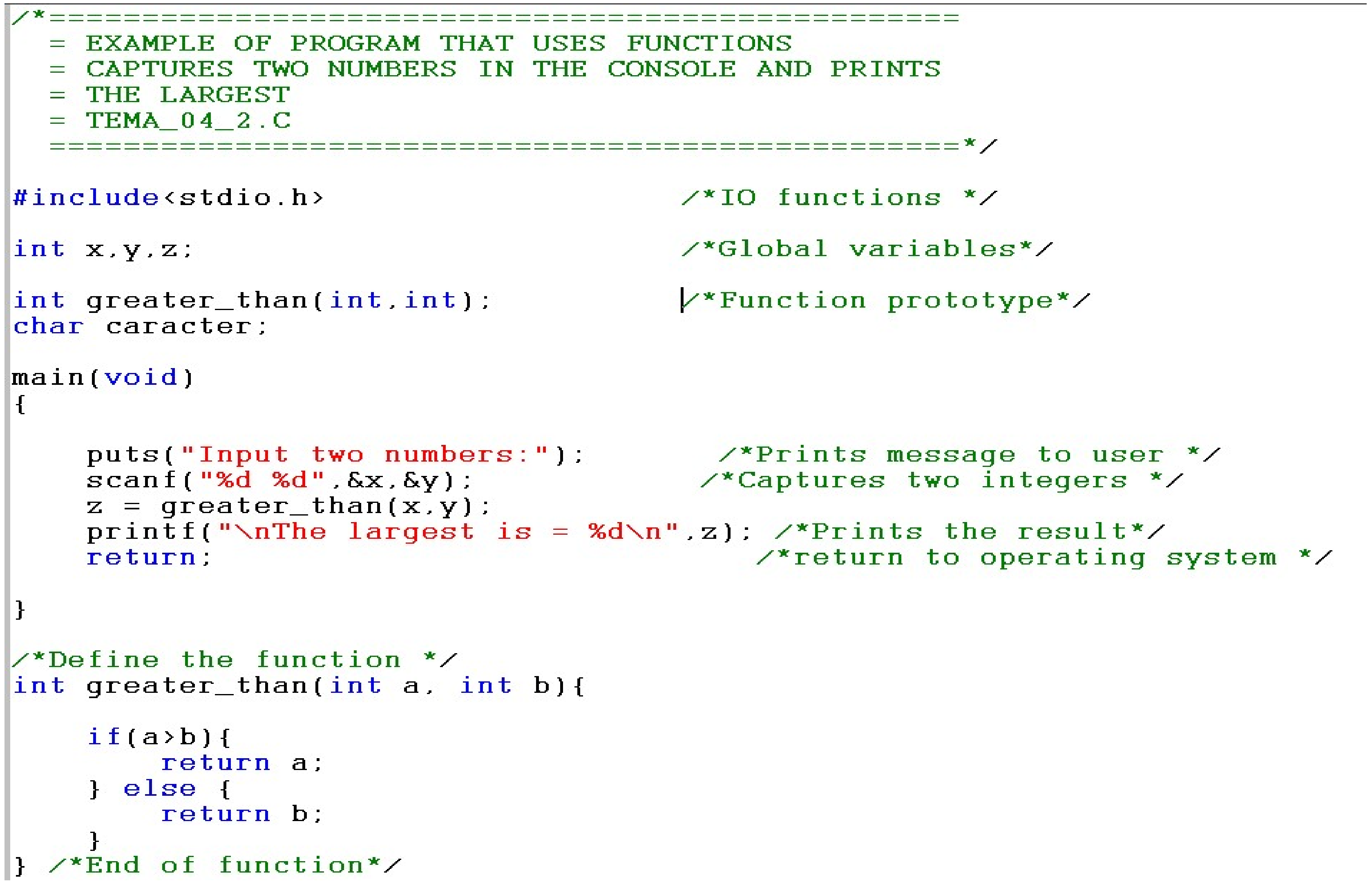
Funtions

- The return value
  - The "return" statement returns a value and the control to the caller a value (or not). Omitting the return statement also returns the control
  - Depending on the logic there can be several returns on the function but it can only be one the defined type and a single value

```
int great_than(int a, int b)
 {
    if (a > b)
       return a;
    else
       return b;
 }
```

Functions

- The function prototype
  - Is a non written rule that all programs must include a prototype of a function to ease their visibility to the rest of the program or for other functions.
  - The function prototype is a formal brief declaration that shows its type and the argument types but does not contain the body
  - They can be defined inside the program or using a header file called in the program (*.h)



```c
/*==================================================
  = EXAMPLE OF PROGRAM THAT USES FUNCTIONS
  = CAPTURES TWO NUMBERS IN THE CONSOLE AND PRINTS
  = THE LARGEST
  = TEMA_04_2.C
  ==================================================*/

#include<stdio.h>                    /*IO functions */

int x,y,z;                           /*Global variables*/

int greater_than(int,int);           /*Function prototype*/
char caracter;

main(void)
{

    puts("Input two numbers:");        /*Prints message to user */
    scanf("%d %d",&x,&y);             /*Captures two integers */
    z = greater_than(x,y);
    printf("\nThe largest is = %d\n",z); /*Prints the result*/
    return;                                /*return to operating system */

}

/*Define the function */
int greater_than(int a, int b){

    if(a>b){
        return a;
    } else {
        return b;
    }
} /*End of function*/
```

Functions

- Invocation (calling)
  - Functions are called when the name and its arguments are placed like a statement on the code, if the functions returns a value and is not assigned to a variable the value is lost.

```
greather_than(100,200);
```

  - Functions that return a value can be called by experesions and also other functions, is like a variable:

```
printf("\nThe biggest was: %d",greather_than(x,y));

if( x > greather_than(x,y)) printf("\n x was greater");
```

Pointers

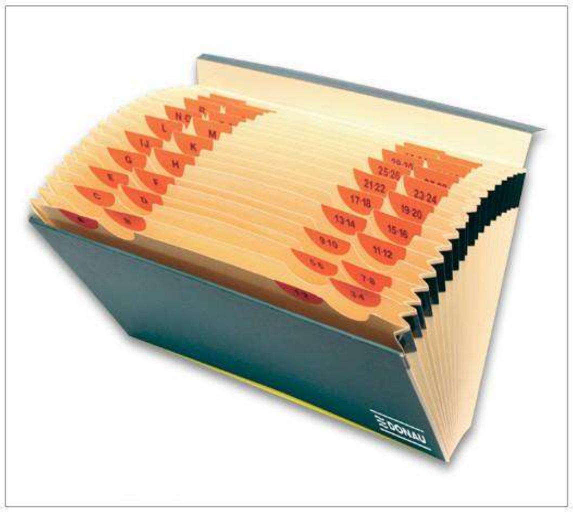
- Variables store data, pointers store the address in memory where the data is contained.
- `type_name *pointer_name`

```
int *aptr;   //Declares a pointer called aptr
int A,B;     //Declares two int variables
aptr = &A    //Assign the address a A to the aptr pointer
B = *aptr    //Copies the content of A to variable B
```

Arrays

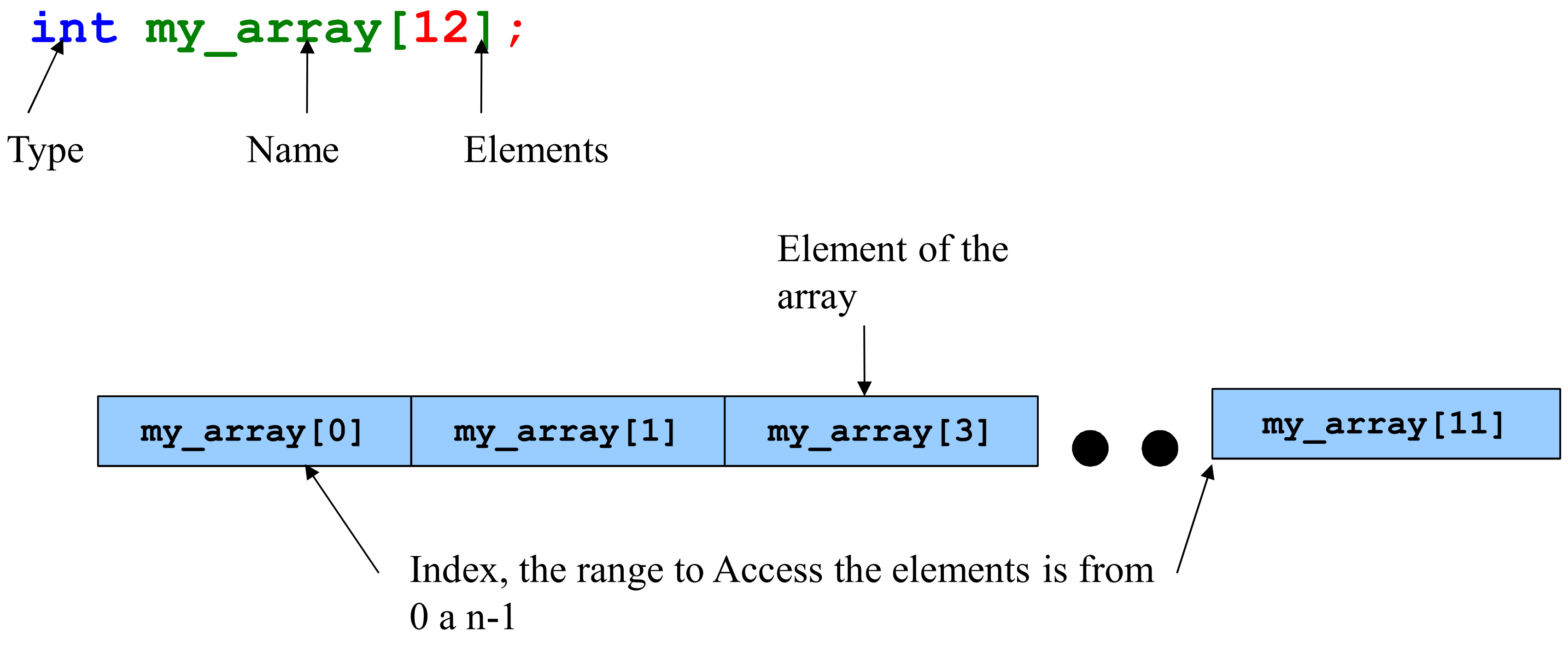
- An **array** is defined as the collection of similar type of data items stored at contiguous memory locations..

Variables


Array

Unidimensional arrays

- They have only one index



Array

- The element of the array can be used as any other variable.

```
my_array[4] = 13;
my_array[3] = my_array[0];
my_array[5] = other_array[122];
```

- You can use expression as the index (as long they are not floating type).

```
my_array[i+1] = 13;
my_array[i*2] = 2;
my_array[my_index[2]]= x;
```

Multidimensional arrays

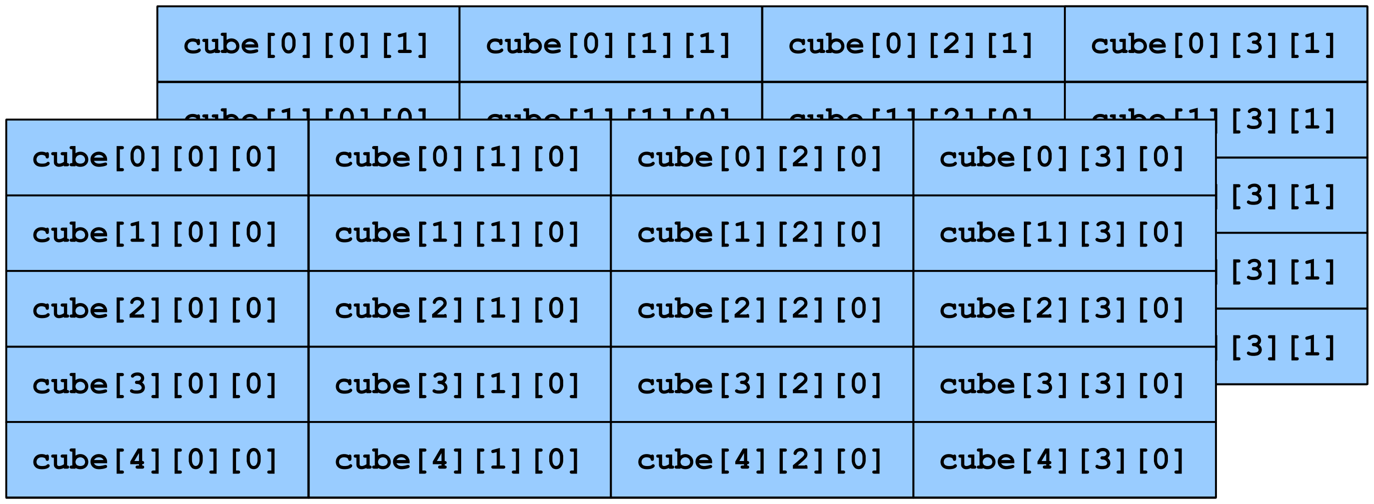
- They can have two or more indexes (In theory there is no limit)

```
float table[6][8];
```

`table[0][0]`	`table[0][1]`	`table[0][2]`	…	`table[0][7]`
`table[1][0]`	`table[1][1]`	`table[1][2]`	…	`table[1][7]`
`table[2][0]`	`table[2][1]`	`table[2][2]`	…	`table[2][7]`
……………………	……………………	……………………	……	……………………
……………………	……………………	……………………	……	……………………
`table[5][0]`	`table[5][1]`	`table[5][2]`	…	`table[5][7]`

Multidimensional arrays

```
char cube[5][4][2];
```



Array initialization

```
int my_array[4] = {1,23,44,1000};
int my_array[]  = {1,23,44,1000};
int my_table[2][3] = {1,2,3,4,5,6};
                              //Row 1 {1,2,3}
                              //Row 2 {4,5,6}
int my_table[2][3] = {{1,2,3},{4,5,6}};
```

```
/*==================================================
  = EXAMPLE OF PROGRAM THAT USES ARRAYS
  = OBTAINS THE LARGEST NUMBER STORED IN
  = AN ARRAY OF 5 ELEMENTS
  = TEMA_04_3.C
  ==================================================*/
#include<stdio.h>                    /*IO functions */

int greater_than(int,int);           /*function prototypsn */

#define ELEM 5                       /*Define a label that has a value of 5*/

main(void)
{
int arr[ELEM]={8,22,10,4,33};        /*Define and init the array*/
int z = 0;                           /*Working variable*/
int i;
    /*Loop 5 times*/
    for(i=0;i<5;i++){
        printf("\nIteration i= %d Compare %d against %d ",i,z,arr[i]);
        z = greater_than(z,arr[i]);
        printf("the largest is %d\n ",z);
    };
    printf("\nThe largest of all te array is= %d\n",z); /*Prints the result*/
    return;                                     /*Return to the caller */
}

/*Function greater_than */
int greater_than(int a, int b){

    if(a>b)
        return a;
    else
        return b;
```

Pointers and arrays

- Any operation that can be done using indexes can be done with poitners.

```
int ax[20];`   //Declares an array of 20 elements
int *ip;       //Declares a pointer to an integer
ip = &ax[0]    //Sets to the pointer the address of element 0
                //Examples of use of arrays:
ax[0] = 0x01; //Stores number 0x01 to element 0 of the array
ax[1] = 0x02; //Stores number 0x02 to element 1 of the array
                //Using with pointers can be done:
 *ip = 0x01;    //Stores 0x01 to the variable pointed by ip
 ip = ip +1;    //Increments the pointer (points to the next)
 *ip = 0x02;    //Stores 0x02 to the variable pointed by ip + 1
```

```c
/*=================================================
  = EXAMPLE USING POINTERS
  = PRINTS THE NUMBERS STORED IN AN ARRAY
  = TEMA_04_4.C
  =================================================*/

#include<stdio.h>                   /*IO cuntions */

main(void)
{
int arr[5]={8,20,33,12,6};           /*Define the arraya*/
int i=0;                             /*Work variablel*/
int *pointer;                     /*Define a pointer*/

    /*Prints the array using the array handing methohd */
    printf("\n Using arrays arr[i]");
    for(i=0;i<5;i++) printf("\n arr[%d] = %d",i,arr[i]);

    /*Prints the array using the pointer conceptr*/
    pointer = &arr[0];  /*Assing the address of the first element*/
    printf("\n Using pointers");

    /*Printing loop*/
    for(i=0;i<5;i++) {
        printf("\n [pointer + %d] = %d",i,*pointer);
        pointer++;
    }
}
```

Using arrays as function arguments

- You can use an array as a function argument

```
int average(int,int arr[]);`  //Function prorotype
   void main(){
       int my_array[25];  //Stores 25 elements
       //…Here goes code to introduce values to the array
        printf("My averay= %d\n",average(25,my_array));
   }
//**********************************************************
    int average(int n,int arr[]){  //Function that uses the array
    int i;
    total = 0;
    for(i=0;i<n;i++) {
         total = total + arr[i]; //Adds all the elements
    }
    return(total/n);
    }
```

Using pointer as function arguments

- You can use pointers to pass arrays

```
int average(int,int *);  //Function prototype
   void main(){
        int my_array[25];  //Stores 25 elements
        //…Here goes code to introduce values to the array
        printf("My averay= %d\n",average(25,&my_array[0]));
   }
//**********************************************************
    int average(int n,int *ptr_array){  //Function that uses pointer
    int i;
    total = 0;
    for(i=0;i<n;i++) {
         total = total + *ptr_array; //Adds all the elements
         ptr_array++; //Point to the next address of the array
    }
    return(total/n);
    }
```

Structures

- A structure is a group of variables under the same name to ease the manipulation of data.

```
struct coord {   //Declares a structure
     int x;
     int y;
    }

void main(){
struct coord pos1,pos2; //We declare to variables having the
structure

pos1.x = 1;
pos1.y = 2;
;
//We can transfer the data from one structure value to the other
pos2 = pos1;
```

Structures

- The structures can be contain arrays and pointer in their definition
- You can also make an array of the structure

Unions

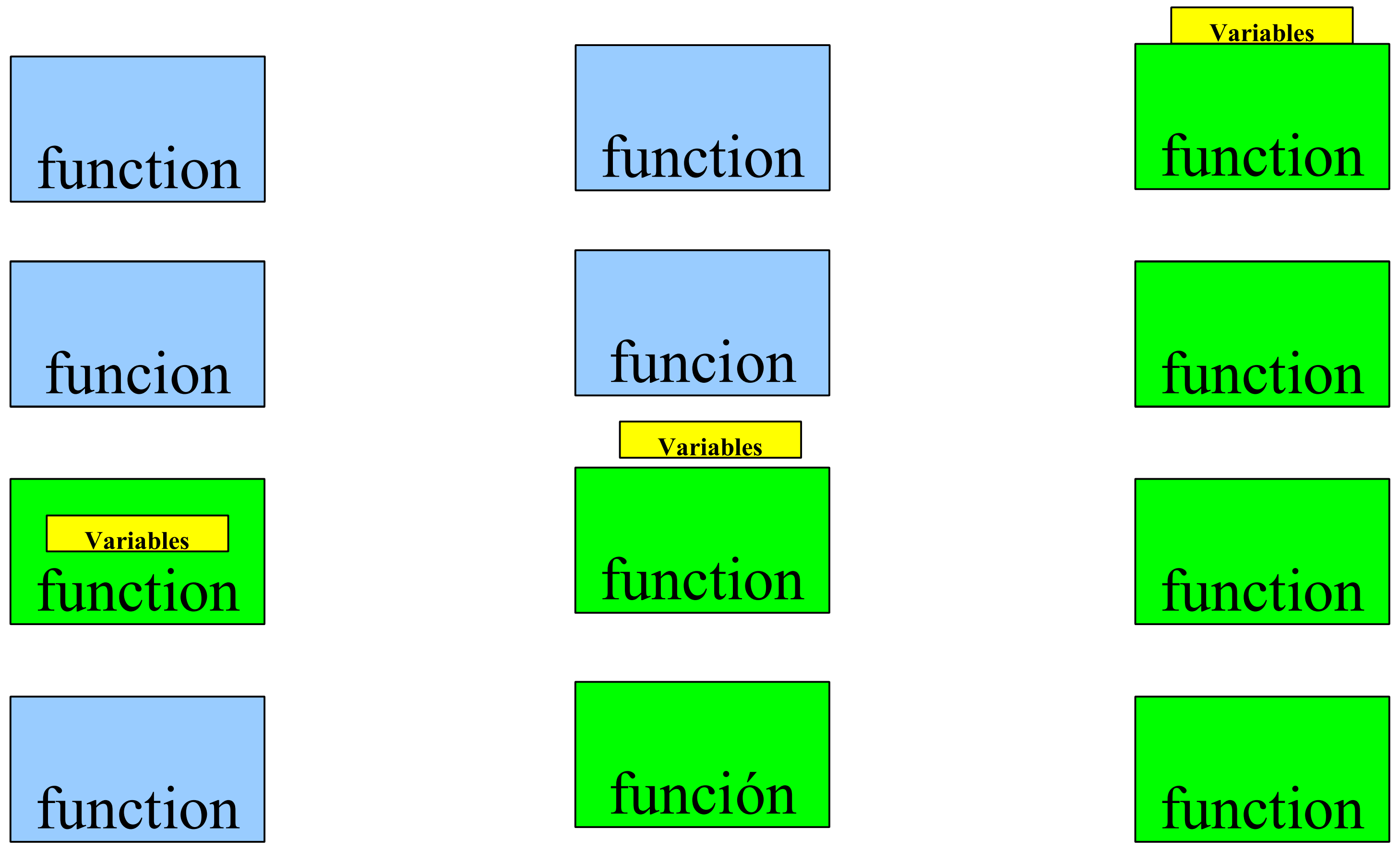
- Out of the scope of the course, but is a special data type available in C that allows to store different data types in the same memory location

```
union my_data {   //Declares a structure
     char c;
     float f;
    }

void main(){
union my_data var_1; //We declare to variables using the union

var1_1.c = ‘k’;      //var_1 stores a char type value = ‘k’
var1_1.f = 123.45;   //var_1 is now a float but ‘k’ value is lost
```

Scope of variables

- The scope is the control of the sections of code that can “see” an specific
- The scope is also the time that the variables lasts in memory

Variable scope



External variables

- Used when programs are formed of modules (several files), an external variable indicates the compiler that the variable is declared in another file and is visible to the modules where is declared.

```
extern int x;   //This variable is visible in the entire code
void main{void){

   }
```

Automatic and static variables

- Variables are by default automatic(the value is lost out if the funcion) except if you declare the them as static

```
int function1 {void){
     int x=0;      //The X variable is AUTOMATIC
     x = x +1 ;    //Modification of the value of x
     return(x);    //The value of x is lost after the
                   //function returns
    }

int function2 {void){
    static int x=0;   //The X variable is STATIC
     x = x +1 ;       //Modification of the value of X
     return(x);       //The value of X is kept in memory
    }
```

The PIC18 compiler

- Data types

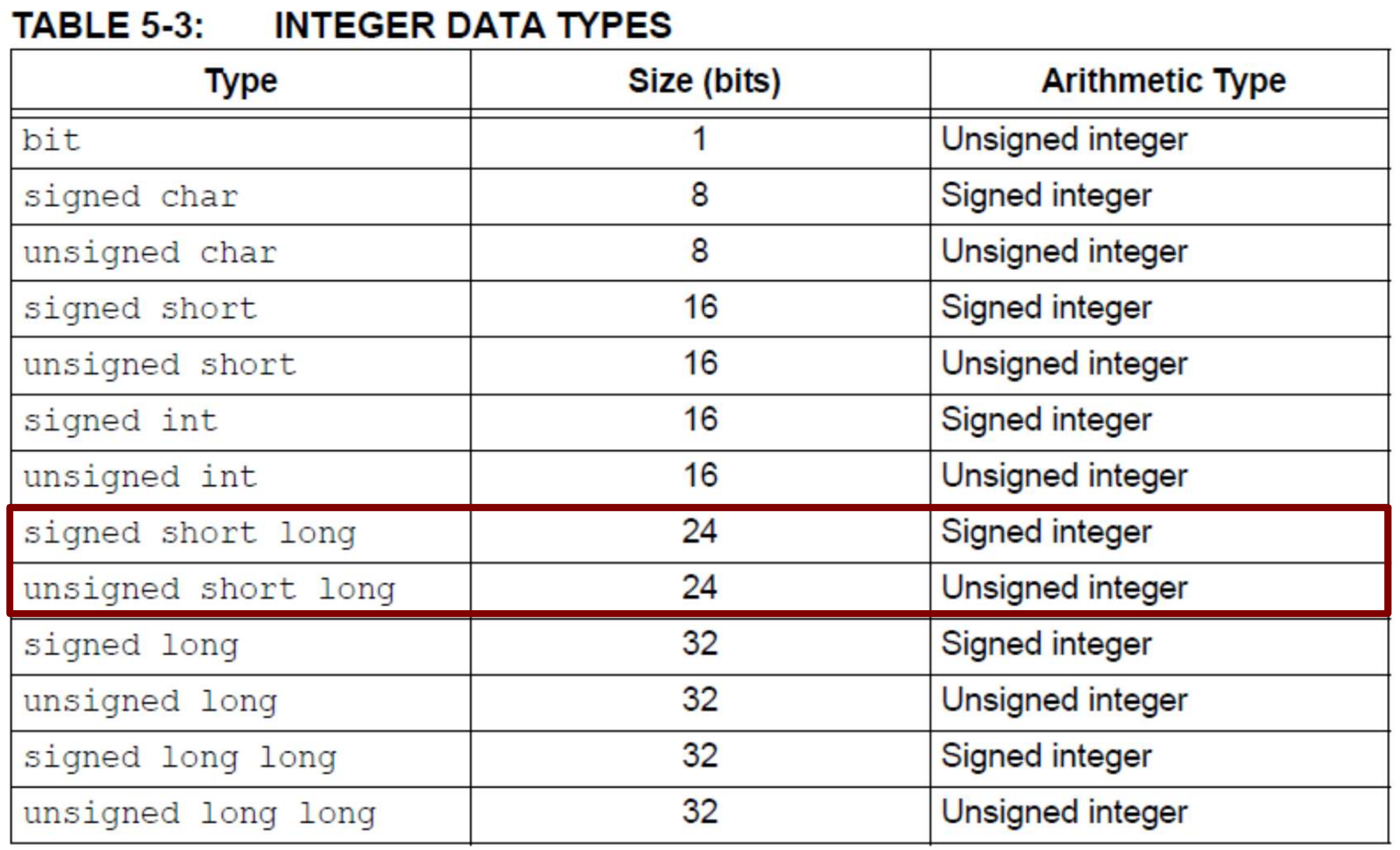


TABLE 5-3: INTEGER DATA TYPES

Type	Size (bits)	Arithmetic Type
`bit`	1	Unsigned integer
`signed char`	8	Signed integer
`unsigned char`	8	Unsigned integer
`signed short`	16	Signed integer
`unsigned short`	16	Unsigned integer
`signed int`	16	Signed integer
`unsigned int`	16	Unsigned integer
`signed short long`	24	Signed integer
`unsigned short long`	24	Unsigned integer
`signed long`	32	Signed integer
`unsigned long`	32	Unsigned integer
`signed long long`	32	Signed integer
`unsigned long long`	32	Unsigned integer

The PIC18 compiler

- Data types

TABLE 2-2: FLOATING-POINT DATA TYPE SIZES AND LIMITS

Type	Size	Minimum Exponent	Maximum Exponent	Minimum Normalized	Maximum Normalized
`float`	32 bits	-126	128	$2^{-126} \approx 1.17549435e-38$	$2^{128} * (2-2^{-15}) \approx 6.80564693e+38$
`double`	32 bits	-126	128	$2^{-126} \approx 1.17549435e-38$	$2^{128} * (2-2^{-15}) \approx 6.80564693e+38$

The PIC18 compiler

- Storage types
  - Auto,Static, Extern, Register, Persistent
- Storage types
  - Persistent, retains variable value after reset

The PIC18 compiler

- Numerical constants

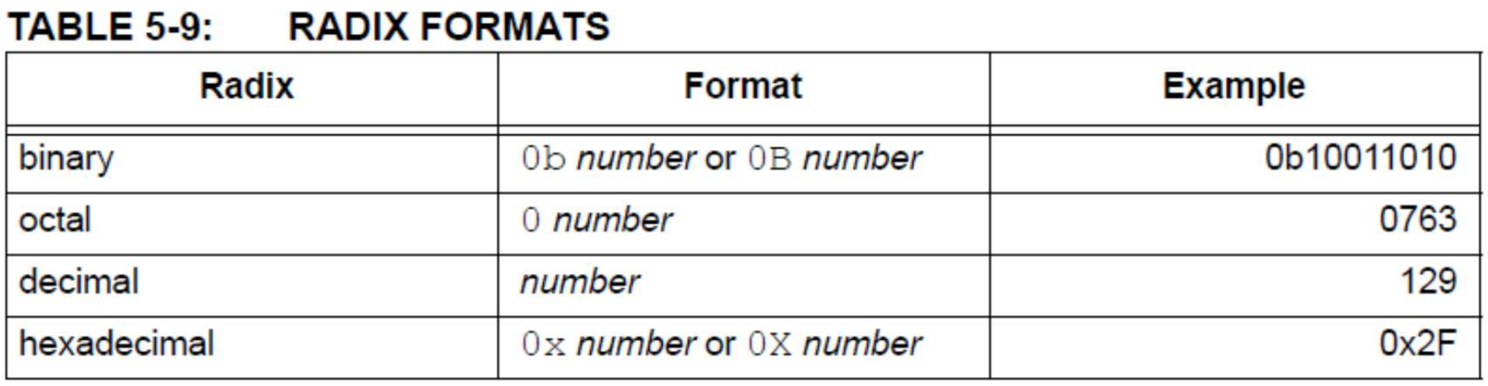


TABLE 5-9: RADIX FORMATS

Radix	Format	Example
binary	`0b` *number* or `0B` *number*	0b10011010
octal	`0` *number*	0763
decimal	*number*	129
hexadecimal	`0x` *number* or `0X` *number*	0x2F

The PIC18 compiler

- “String” type constants
  - The compiler stores string or constant arrays in the memory program ROM in a special section called “stringtable”
  - There are 4 types of methods to access the strings (this is usually transparent to the programmer)

The PIC18 compiler

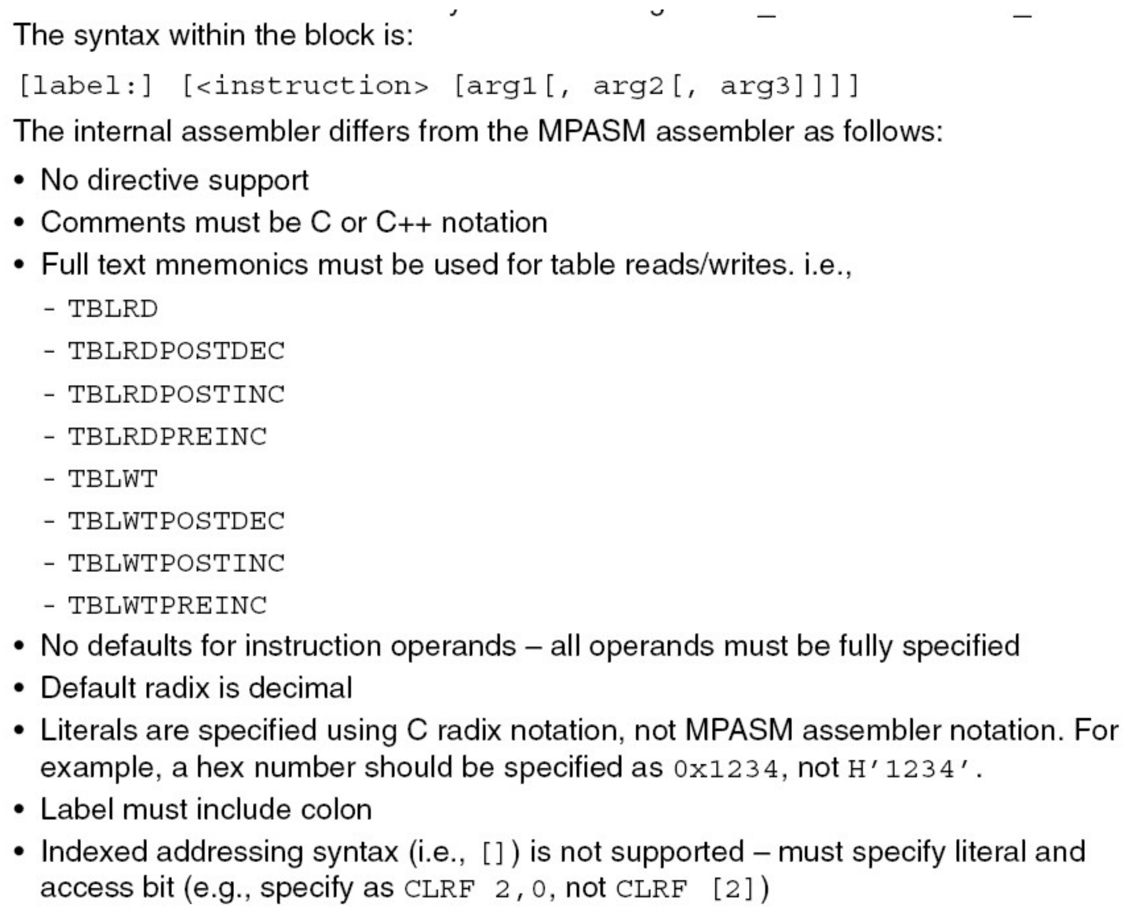
- The compiler supports bit variable types
  - Is like a unsigned char that can have only 0 or 1 value, can be used as variables, function types or arguments in functions.
  - They are efficient since up to 8 bits can be stored in a single byte in RAM
  - The bit variable definition is not part of the standard so avoid its use is your code is to be portable or compatible with other brand of compiler
  - To declares use “static __bit” to be compatible with older versions of XC18

The PIC18 compiler

- We can write assembly language inside C
  - El PIC18 compiler allows to write blocks of assembly code using C language
  - The block must start with #asm and end #endasm or use the asm("assembly goes here");
  - For compatibilty with C18 the use of _asm and _endasm still supported

The PIC18 compiler

- Rules to written assembly inside C



The syntax within the block is:

```
[label:] [<instruction> [arg1[, arg2[, arg3]]]]
```

The internal assembler differs from the MPASM assembler as follows:

- No directive support
- Comments must be C or C++ notation
- Full text mnemonics must be used for table reads/writes. i.e.,
  - `TBLRD`
  - `TBLRDPOSTDEC`
  - `TBLRDPOSTINC`
  - `TBLRDPREINC`
  - `TBLWT`
  - `TBLWTPOSTDEC`
  - `TBLWTPOSTINC`
  - `TBLWTPREINC`
- No defaults for instruction operands – all operands must be fully specified
- Default radix is decimal
- Literals are specified using C radix notation, not MPASM assembler notation. For example, a hex number should be specified as `0x1234`, not `H'1234'`.
- Label must include colon
- Indexed addressing syntax (i.e., `[]`) is not supported – must specify literal and access bit (e.g., specify as `CLRF 2,0`, not `CLRF [2]`)

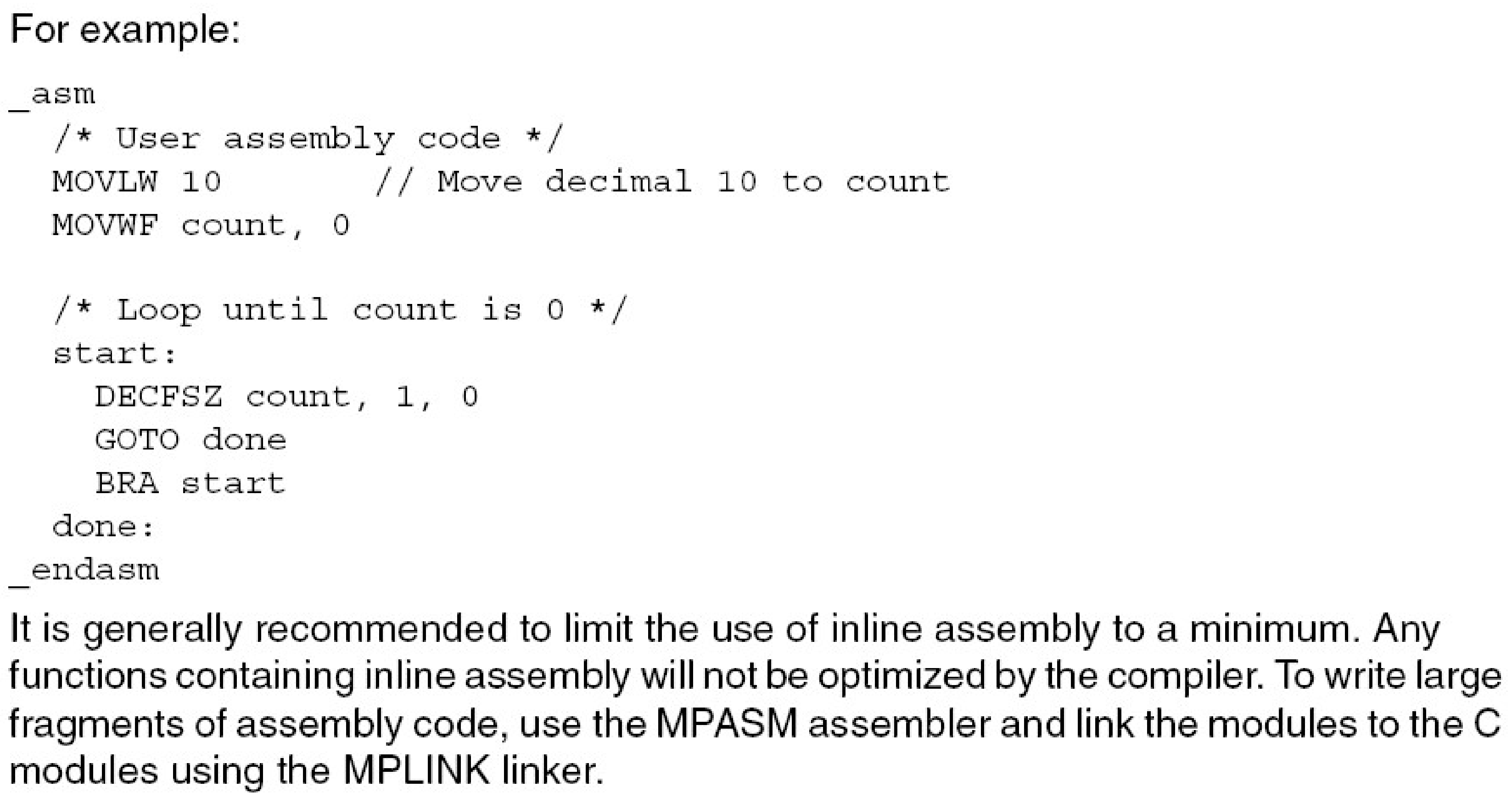
The PIC18 compiler

- Example of assembly inside C

```
unsigned int var;

void main(void)
{
    var = 1;
#asm              // like this...
    BCF 0,3
    BANKSEL(_var)
    RLF (_var)&07fh
    RLF (_var+1)&07fh
#endasm
                 // do it again the other way...
    asm("BCF 0,3");
    asm("BANKSEL _var");
    asm("RLF (_var)&07fh");
    asm("RLF (_var+1)&07fh");
}
```

The PIC18 compiler

- C18 compiler method of inline assembly (still supported)



For example:

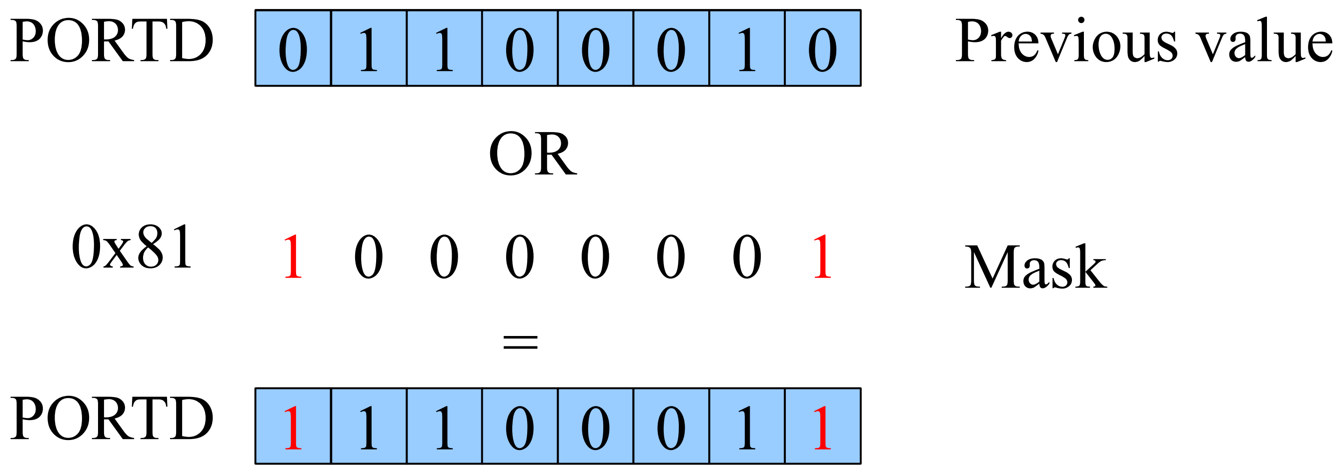
```
_asm
  /* User assembly code */
  MOVLW 10        // Move decimal 10 to count
  MOVWF count, 0

  /* Loop until count is 0 */
  start:
    DECFSZ count, 1, 0
    GOTO done
    BRA start
  done:
_endasm
```

It is generally recommended to limit the use of inline assembly to a minimum. Any functions containing inline assembly will not be optimized by the compiler. To write large fragments of assembly code, use the MPASM assembler and link the modules to the C modules using the MPLINK linker.

The PIC18 compiler

- Bit manipulation
  - The bitwise operations is the more standard way for bit level manipulation:

```
PORTD = PORTD | 0x81;  //PORTD = PORTD OR 0x81
                       //PORTD |= 0X81
```



The PIC18 compiler

- Bit manipulation

```
PORTD = PORTD & 0xC3;   //PORTD = PORTD AND 0xC3
                         //PORTD &= 0XC3
```

PORTD	1	1	0	1	0	1	1	1	Previous value
				AND					
0xC3	1	1	0	0	0	0	1	1	Mask
				=					
PORTD	1	1	0	0	0	0	1	1	

The PIC18 compiler

- Bit manipulation

```
PORTD = PORTD ^ 0x86;   //PORTD = PORRD XOR 0x86
                         //PORTD ^= 0X86
```

PORTD	1	1	0	1	0	1	1	1	Previous value
				XOR					
0x86	1	0	0	0	0	1	0	1	Mask
				=					
PORTD	0	1	0	1	0	0	1	0	

The PIC18 compiler

- Bit manipulation summary
  - To make 0 one or of a group of bits on a variable we use the AND operator with a mask that contains 0 in the bit we want to clear and 1 where we don't want any change.
  - To make 1 one or of a group of bits on a variable we use the OR operator with a mask that contains1 in the bit we want to set and 0 where we don't want any change.
  - To change toggle the state of bits on a variable we use the XOR operator with a mask that contains1 in the bit we want to toggle (turn 0 to 1 and 1 to 0)

The PIC18 compiler

- Bit manipulation using special structures defined in header files.
  - Microchip provides header file that allows direct bit manipulation to all SRF registers in a more direct way
  - This a non standard method that does not comply the the ANSI C standard.

The PIC18 compiler

- Bit manipulation using header file definition using th

```
extern volatile near unsigned char PORTA;
```

and as:

```
extern volatile near union  {
  struct {
    unsigned RA0:1;
    unsigned RA1:1;
    unsigned RA2:1;
    unsigned RA3:1;
    unsigned RA4:1;
    unsigned RA5:1;
    unsigned RA6:1;
  } ;
  struct {
    unsigned AN0:1;
    unsigned AN1:1;
    unsigned AN2:1;
    unsigned AN3:1;
    unsigned T0CKI:1;
    unsigned SS:1;
    unsigned OSC2:1;
  } ;
  struct {
    unsigned :2;
    unsigned VREFM:1;
    unsigned VREFP:1;
    unsigned :1;
    unsigned AN4:1;
    unsigned CLKOUT:1;
  } ;
  struct {
    unsigned :5;
    unsigned LVDIN:1;
  } ;
} PORTAbits ;
```

The PIC18 compiler

- Bit manipulation using header file definition as .bits bitfield, or bit variables

```
#include <xc.h>          //Contains the #includ(pxxx.h)
void main (void){
    PORTA = 0x00 ;      //Uses a #define
    RA0 = 1 ;            //Uses bit variable
    PORTAbits.RA0 = 1; //bitfield structure
}
```

The PIC18 compiler

- The header file also provides Macros of code written in assembly to make certain operations that cannot be made using direct C statements.

TABLE 2-8: C MACROS PROVIDED FOR PICmicro® MCU INSTRUCTIONS

Instruction Macro[1]	Action
`Nop()`	Executes a no operation (`NOP`)
`ClrWdt()`	Clears the Watchdog Timer (`CLRWDT`)
`Sleep()`	Executes a `SLEEP` instruction
`Reset()`	Executes a device reset (`RESET`)
`Rlcf(var, dest, access)` [2,3]	Rotates *var* to the left through the carry bit
`Rlncf(var, dest, access)` [2,3]	Rotates *var* to the left without going through the carry bit
`Rrcf(var, dest, access)` [2,3]	Rotates *var* to the right through the carry bit
`Rrncf(var, dest, access)` [2,3]	Rotates *var* to the right without going through the carry bit
`Swapf(var, dest, access)` [2,3]	Swaps the upper and lower nibble of *var*

The PIC18 compiler

- The #pragma directive
  - Is used to control the behavior or the compiler. It can be used to define memory spaces, interruption code, etc. Is a way to define that are not part of the C language standard and this is particular to the compiler or the target processor.

Mixing assembly and C

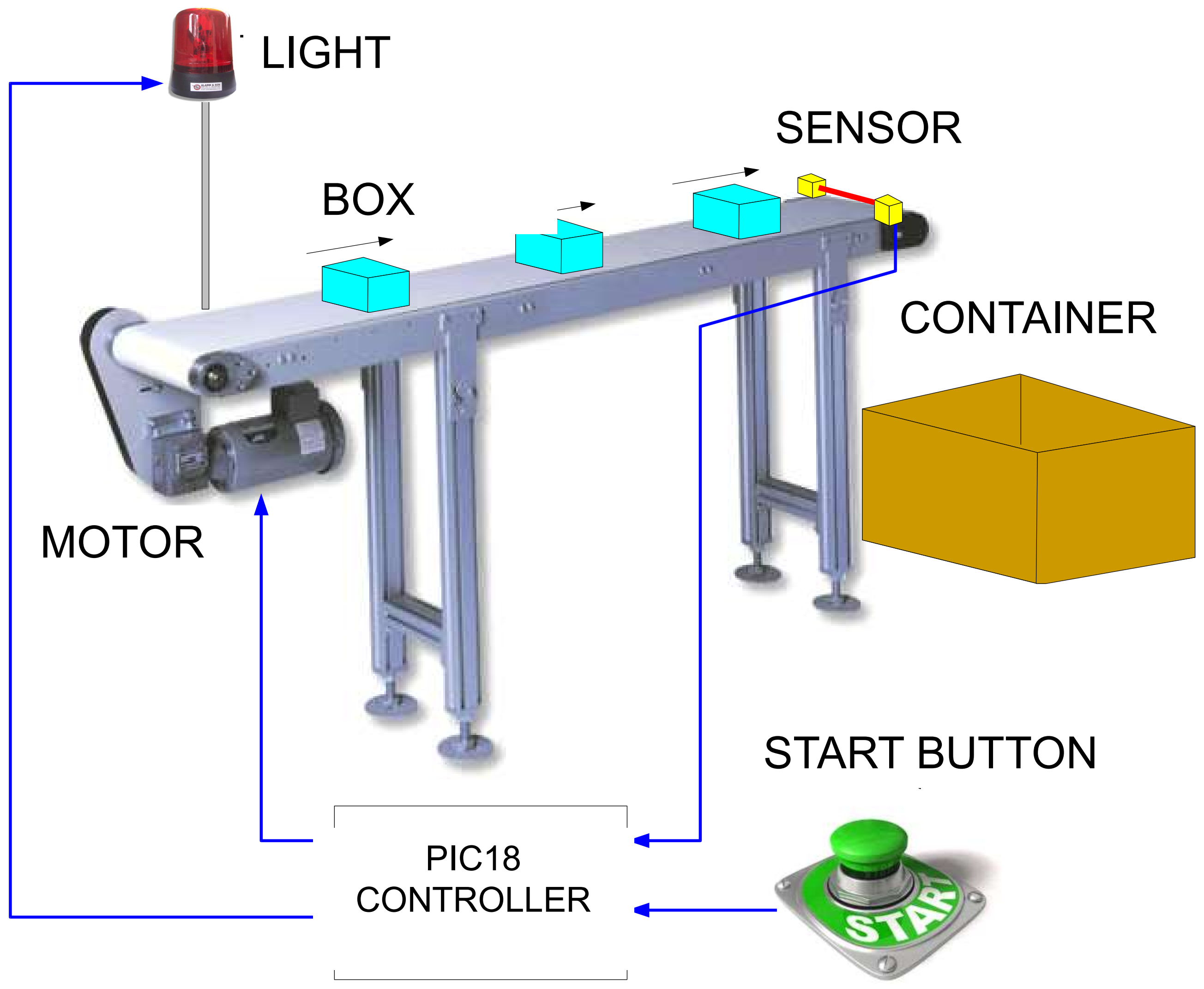
- Is possible that functions written in C to contain assembly and vice versa
- These type of coding is out of the scope of the course but it is defined in the assembly manual

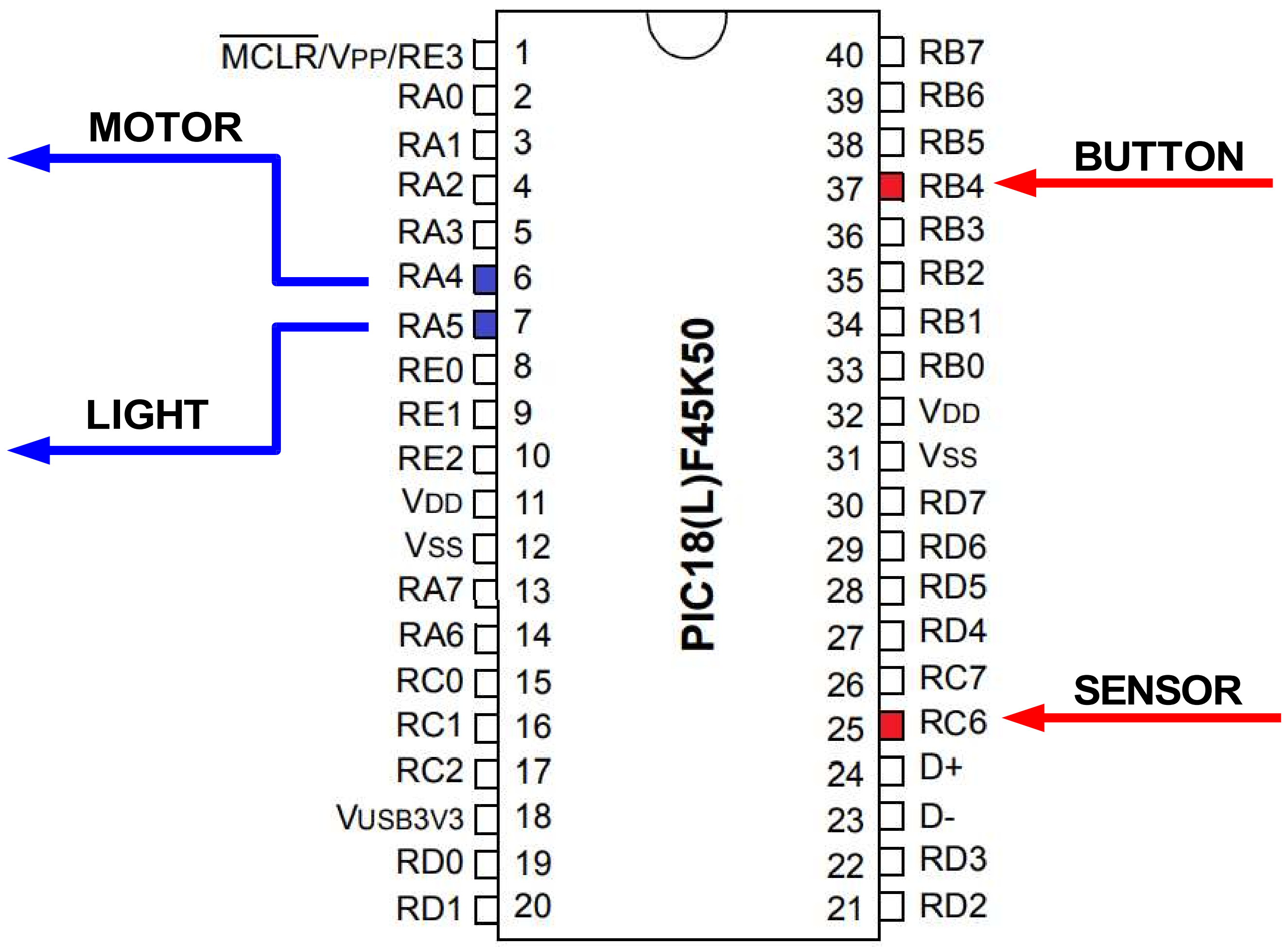
Startup Code

- When the microcontroller starts execution the first instruction at address is 0x00000
- The compiler adds a section of code that performs all the initializations before the code in main() is execute.
- This code can be modified by the user although it is not.

Example Using MPLAB

Conveyor belt controller



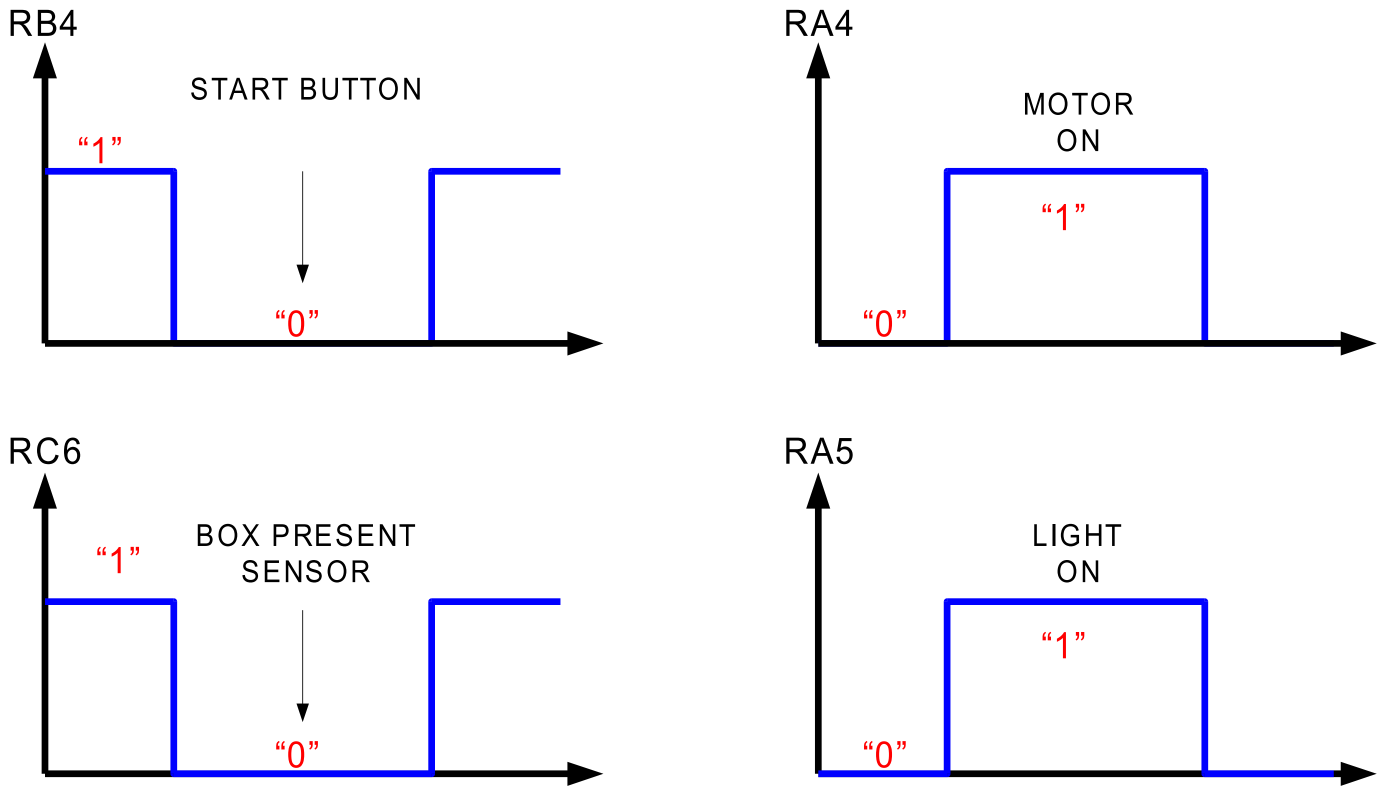

MOTOR
LIGHT
BUTTON
SENSOR
PIC18(L)F45K50
MCLR/VPP/RE3 1
RA0 2
RA1 3
RA2 4
RA3 5
RA4 6
RA5 7
RE0 8
RE1 9
RE2 10
VDD 11
VSS 12
RA7 13
RA6 14
RC0 15
RC1 16
RC2 17
VUSB3V3 18
RD0 19
RD1 20
40 RB7
39 RB6
38 RB5
37 RB4
36 RB3
35 RB2
34 RB1
33 RB0
32 VDD
31 VSS
30 RD7
29 RD6
28 RD5
27 RD4
26 RC7
25 RC6
24 D+
23 D-
22 RD3
21 RD2

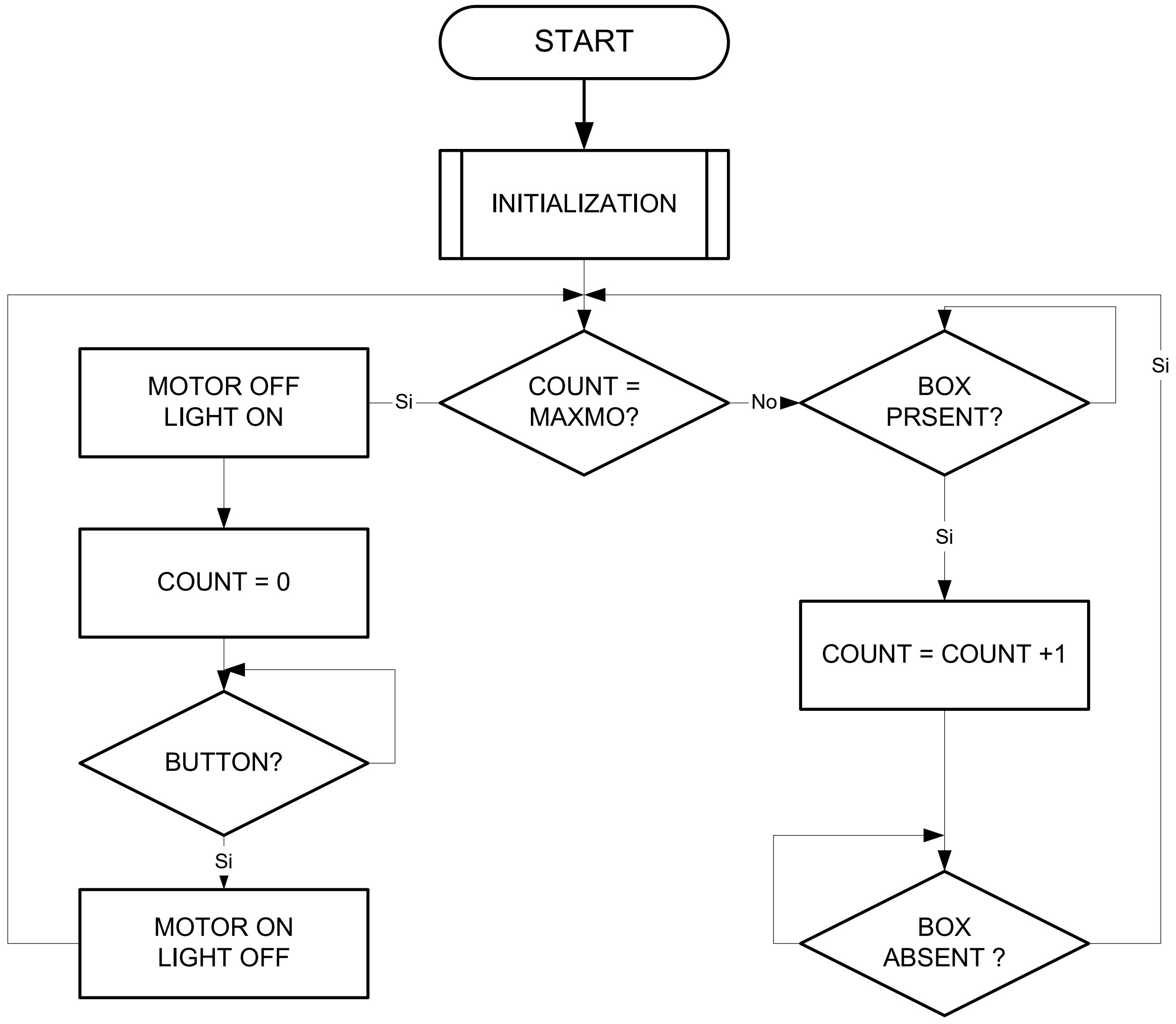
Conveyor belt controller

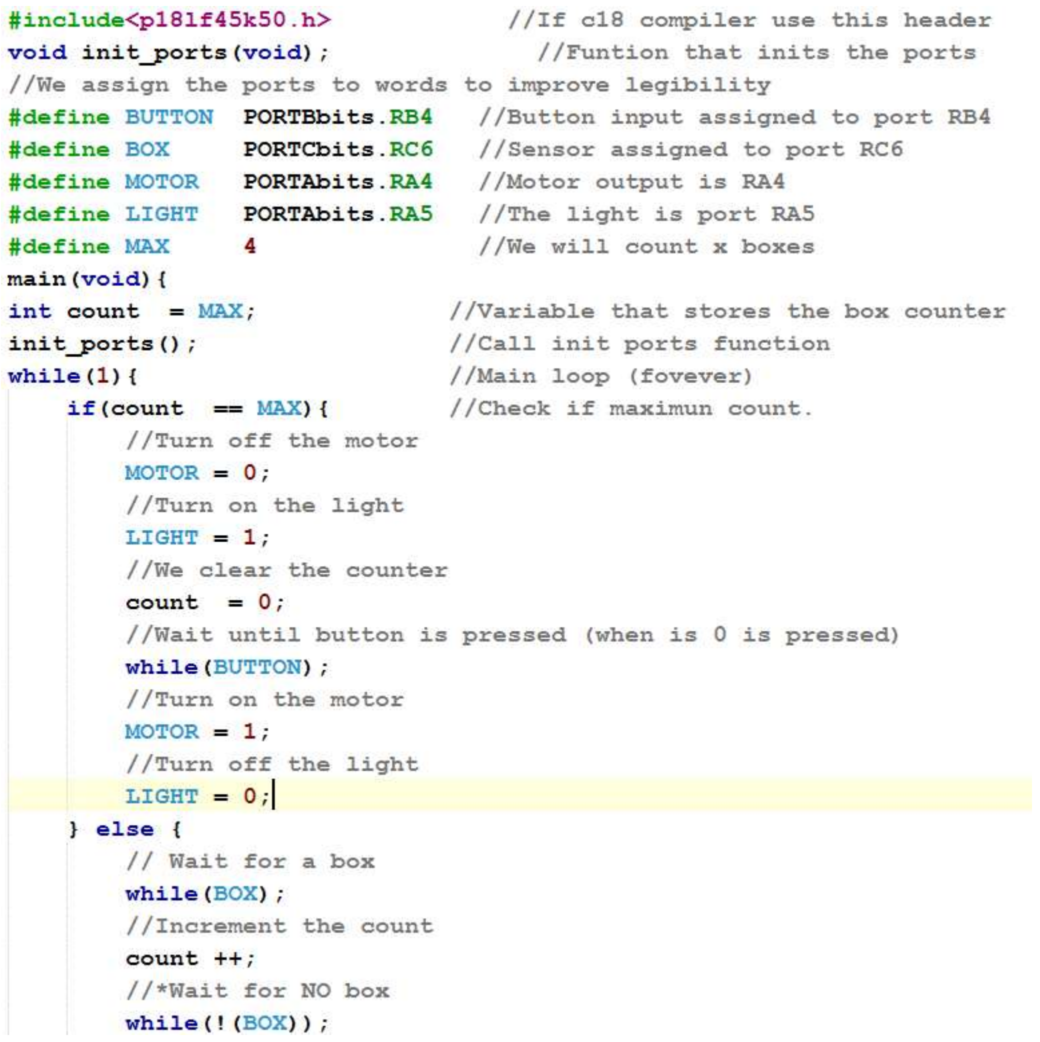
- The control program must do the following:
  - Wait for the start button to be pressed
  - Count the number of boxes
  - If the number is equal to MAXIMO
    - Stop the conveyor belt
    - Turn the light
    - Wait again for the start buton

Conveyor belt controller

- Signals for actuators and from sensors:

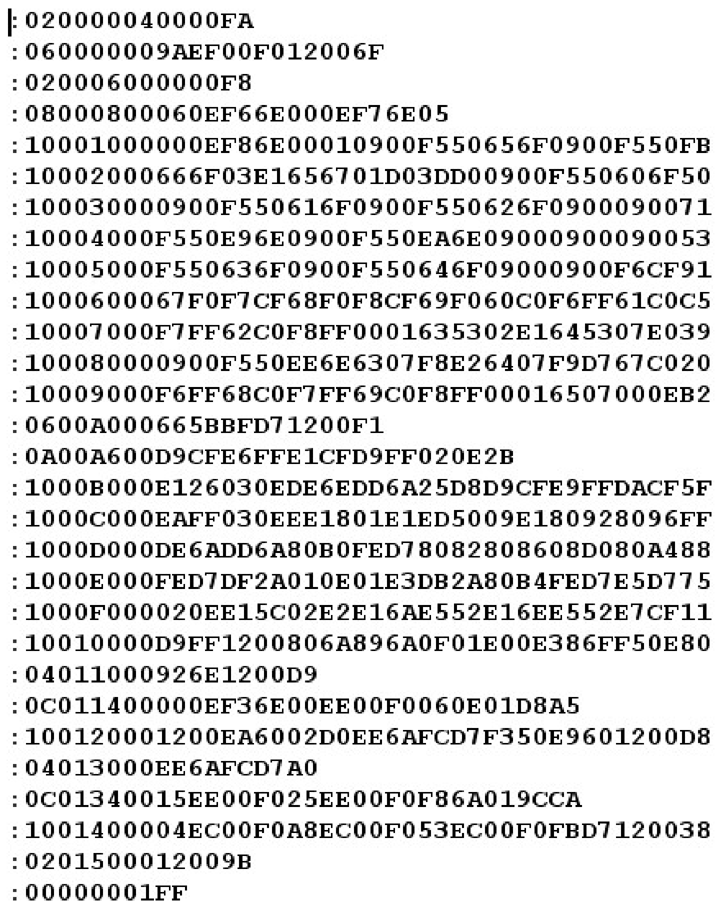



START
INITIALIZATION
COUNT = MAXMO?
Si
No
MOTOR OFF LIGHT ON
COUNT = 0
BUTTON?
Si
MOTOR ON LIGHT OFF
BOX PRSENT?
Si
COUNT = COUNT +1
BOX ABSENT ?
Si



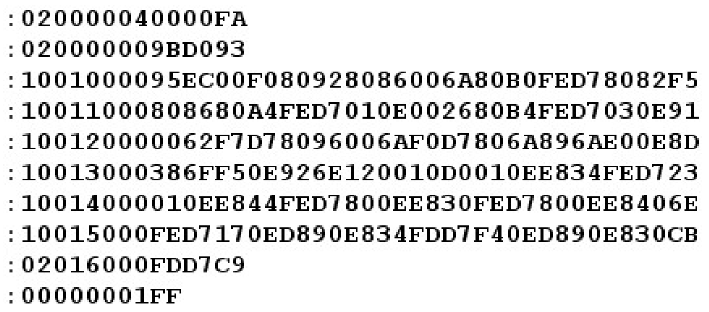
```
#include<p18lf45k50.h>                //If c18 compiler use this header
void init_ports(void);                  //Funtion that inits the ports
//We assign the ports to words to improve legibility
#define BUTTON  PORTBbits.RB4   //Button input assigned to port RB4
#define BOX     PORTCbits.RC6   //Sensor assigned to port RC6
#define MOTOR   PORTAbits.RA4   //Motor output is RA4
#define LIGHT   PORTAbits.RA5   //The light is port RA5
#define MAX     4               //We will count x boxes
main(void){
int count  = MAX;              //Variable that stores the box counter
init_ports();                  //Call init ports function
while(1){                      //Main loop (fovever)
    if(count  == MAX){         //Check if maximun count.
        //Turn off the motor
        MOTOR = 0;
        //Turn on the light
        LIGHT = 1;
        //We clear the counter
        count  = 0;
        //Wait until button is pressed (when is 0 is pressed)
        while(BUTTON);
        //Turn on the motor
        MOTOR = 1;
        //Turn off the light
        LIGHT = 0;
    } else {
        // Wait for a box
        while(BOX);
        //Increment the count
        count ++;
        //*Wait for NO box
        while(!(BOX));
        }//from if
  } //from while(1)
  } //from main()
```

Executable comparison



```
:020000040000FA
:060000009AEF00F012006F
:020006000000F8
:08000800060EF66E000EF76E05
:10001000000EF86E00010900F550656F0900F550FB
:10002000666F03E1656701D03DD00900F550606F50
:100030000900F550616F0900F550626F0900090071
:10004000F550E96E0900F550EA6E09000900090053
:10005000F550636F0900F550646F09000900F6CF91
:1000600067F0F7CF68F0F8CF69F060C0F6FF61C0C5
:10007000F7FF62C0F8FF0001635302E1645307E039
:100080000900F550EE6E6307F8E26407F9D767C020
:10009000F6FF68C0F7FF69C0F8FF00016507000EB2
:0600A000665BBFD71200F1
:0A00A600D9CFE6FFE1CFD9FF020E2B
:1000B000E126030EDE6EDD6A25D8D9CFE9FFDACF5F
:1000C000EAFF030EEE1801E1ED5009E180928096FF
:1000D000DE6ADD6A80B0FED78082808608D080A488
:1000E000FED7DF2A010E01E3DB2A80B4FED7E5D775
:1000F000020EE15C02E2E16AE552E16EE552E7CF11
:10010000D9FF1200806A896A0F01E00E386FF50E80
:04011000926E1200D9
:0C011400000EF36E00EE00F0060E01D8A5
:100120001200EA6002D0EE6AFCD7F350E9601200D8
:04013000EE6AFCD7A0
:0C01340015EE00F025EE00F0F86A019CCA
:1001400004EC00F0A8EC00F053EC00F0FBD7120038
:020150001200 9B
:00000001FF
```

C Language
SUBJECT_04_PIC_3.HEX



```
:020000040000FA
:020000009BD093
:1001000095EC00F080928086006A80B0FED78082F5
:10011000808680A4FED7010E002680B4FED7030E91
:100120000062F7D78096006AF0D7806A896AE00E8D
:10013000386FF50E926E120010D0010EE834FED723
:1001400010EE844FED7800EE830FED7800EE8406E
:10015000FED7170ED890E834FDD7F40ED890E830CB
:02016000FDD7C9
:00000001FF
```

Assembly
SUBJECT_04_1.HEX