# CHAPITRE 2:INITIATION A LA PROGRAMMATION EN C POUR PIC

## 8. Les qualités du langage C

Le langage C est :

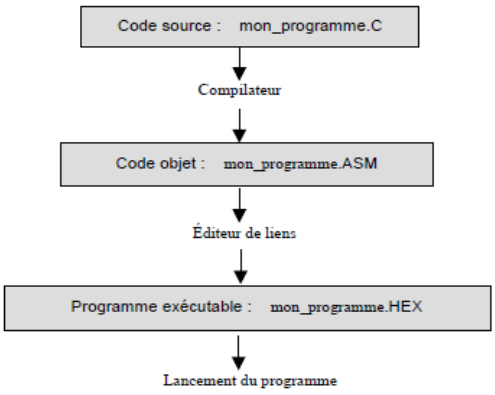
- PORTABLE : Les modifications d'un programme pour passer d'un système à un autre sont minimes.
- COMPLET : Un texte C peut contenir des séquences de bas niveau (proches du matériel) en assembleur.
- SOUPLE : Tout est possible en C mais une grande rigueur s'impose.
- EFFICACE : On réfléchit (devant une feuille de papier) et on écrit (peu).

## 9. Notion de filière de développement

On désigne ainsi l'ensemble des outils qui interviennent entre le texte source C et le code objet téléchargé dans le microcontrôleur PIC.

Les étapes de génération d'un programme écrit en langage C sont :

- **L'édition** du fichier source **mon_programme.C** avec un éditeur de texte (simple sans mise en forme du texte).
- **La compilation** du fichier source pour obtenir un fichier objet : **mon_programme.ASM**. La compilation est la transformation des instructions C en instructions assembleur pour microcontrôleur PIC.
- **L'édition de liens** permet d'intégrer des fonctions prédéfinies. Le programme auxiliaire Éditeur de liens (linker ou binder) génère à partir du fichier mon_programme.ASM un fichier exécutable **mon_programme.HEX** compatible avec le PIC



Structure d'un programme en C

**/*** Tout ce qui se trouve entre ces symboles

est du commentaire ***/**

**//** Ce qui est à droite de ces symboles est également du commentaire

**void main()** // Un programme en C comporte au moins une fonction

// principale appele *main.*

// Nous verrons plus loin le terme *void.*

**{** // Les accolades définissent l'intérieur de la

// fonction.

***fonction_1() ;*** // Les espaces doivent être remplacés par _

***fonction_2() ;*** // Les fonctions sont séparée par un ;

**…**

***fonction_n() ;*** // Attention à la casse car le C fait la différence

// entre minuscules et majuscules.

**}** // On aligne les accolades et on décale le corps de la

// fonction par souci de lisibilité.

Le programme doit également contenir la définition des différentes fonctions créées par le programmeur . Les fonctions doivent être définies (au moins leur prototype) avant d'être appelées par une autre fonction.

Le langage C comporte des bibliothèques de fonctions prédéfinies sous forme de fichiers comportant l'extension .h. Ces fonctions seront ajoutées au programme exécutable lors de l'édition de liens. Pour incorporer dans un programme un fichier .h, on utilise la commande #include <fichier.h> placée habituellement en début de fichier. Exemples :

```
#include <delay.h> // delay.h contient des fonctions réalisant des
                   // temporisations logiciels.
#include <pic16f6x.h> // Grâce à ce fichier .h, le compilateur connaît
                   // l'adresse de chaque registre et port
// Ex : PORTB correspond à l'adresse 06, etc.
```

**10. Composition d'un programme en C**

Un programme en C utilise 2 zones mémoire principales :

- La zone des VARIABLES est un bloc RAM où sont stockées des données manipulées par le programme.
- La zone des FONCTIONS et CONSTANTES est un bloc ROM qui recevra le code exécutable du programme.

Avant d'utiliser une variable, une fonction ou une constante, il faut la déclarer afin d'informer le compilateur de son existence.

Leur nom que l'on utilise est un identificateur. Leur écriture doit :

- Utiliser les lettres de l'alphabet, de a à z, et de A à Z, les chiffres de 0 à 9 (sauf pour le premier caractère), le souligné ( _ ).
- Ne contenir ni espace, ni caractère accentué.
- Être représentative de leur rôle dans le programme

Les différents types de valeur du langage C

Toutes les valeurs (constantes et variables) utilisée en C sont classées selon des types. Un type décide de l'occupation mémoire de la donnée. Pour déclarer correctement une variable ou une constante, il faut donc savoir auparavant ce qu'elle va contenir. On distingue les types suivants :

```
char a ;                 // Déclare un entier signé 8 bits [-128 à +127 ]
unsigned char b ;        // Déclare un caractère non signé 8 bits [0 à 255]
int c ;                  // Déclare un entier signé 16 bits [-32768 à +32767 ]
unsigned int d ;         // Déclare un entier non signé 16 bits [0 à 65535]
long e ;                 // Déclare un entier signé 32 bits
```

// [-2147483648 à +2147483647]

unsigned long f ; // Déclare un entier non signé 32 bits

// [0 à 4292967295]

float g; // Déclare un réel signé 32 bits dont la valeur

// absolue est comprise entre $3,4.10^{-38}$ et $3,4.10^{+38}$

double h; // Déclare un réel signé 64 bits dont la valeur

// absolue est comprise entre $1,7.10^{-308}$ et $1,7.10^{+308}$

long double i; // Déclare un réel signé 80 bits dont la valeur

// absolue est comprise entre $3,4.10^{-4932}$ et $3,4.10^{+4932}$

## 11. Représentation des différentes bases et du code ASCII

int a = 4 ; // Un nombre seul représente un nombre décimal.

int b = **0b**1010 ; // Un nombre précédé de **0b** est un nombre binaire.

int p = **0x**00FF ; // Un nombre précédé de **0x** est un nombre hexadécimal.

char c = **'**x**'** ; // Un caractère entre '' représente son code ASCII.

## 12. Les opérateurs

### 12.1. L'opérateur d'affectation

Cet opérateur a déjà été utilisé dans les exemples précédents. Il permet, entre autres, d'initialiser une variable.

**=** Exemple : **a = 5 ;** // **Range** 5 dans a.

**PORTB = 0** ; // le PORTB est mis à 0

Attention : Le transfert de la valeur va toujours de la droite vers la gauche du signe égal

### 12.2. Les opérateurs arithmétiques

**+** Exemple : a = 5 ; b = 4 ; x = **a+b** ; // rend la **somme** de a et b.

// x vaut **9**

**-** Exemple : a = 5 ; b = 4 ; x = **a-b** ; // rend la **soustraction** de a et b.

// x vaut **1**

***** Exemple : a = 5 ; b = 4 ; x = **a*b** ; // rend la **multiplication** de a et b.

// x vaut **20**

**/** Exemple : a = 10 ; b = 3 ; x = **a/b** ; // rend le **quotient** de la division

// entière de a et b.

// x vaut **3**

**%** Exemple : a = 10 ; b = 3 ; x = **a%b** ; // rend le **reste** de la division

// entière de a et b.

// x vaut **1**

// % se prononce « modulo »

### 12.3. Les opérateurs de manipulation de bits

**~** Exemple : a = 0b0110 ; x = **~a** ; // rend le **complément** de a.

// x vaut **0b1001**

**&** Exemple : a = 2 ; b = 3 ; x = **a&b** ; // rend le **ET** bit à bit de a et b.

// x vaut **0b10**

**|** Exemple : a = 2 ; b = 5 ; x = **a|b** ; // rend le **OU** bit à bit de a et b.

// x vaut **0b111**

**^** Exemple : a = 2 ; b = 7 ; x = **a^b** ; // rend le **OU EXCLUSIF** bit à

// bit de a et b.

// x vaut **0b101**

**>>** Exemple : a = 2 ; b = 1 ; x = **a>>b** ; // rend la valeur de a **décalée à**

// **droite** de b bits.

// x vaut **0b01**

<< Exemple : a = 2 ; b = 3 ; x = **a<<b** ; // rend la valeur de a **décalée à**

// **gauche** de b bits.

// x vaut **0b10000**

### 12.4. Les opérateurs de test

Remarque : En C, **FAUX** est la valeur **0**, **VRAI** est tout ce qui est ≠ **0**.

> Exemple : a = 6 ; x = **a>4** ; // rend VRAI si a est **supérieur** à 4. FAUX sinon.

// x vaut **VRAI**

>= Exemple : a = 2 ; x = **a>= 2** ; // rend VRAI si a est **supérieur ou égal** à 2.

//FAUX sinon.

// x vaut **VRAI**

< Exemple : a = 6 ; x = **a<3** ; // rend VRAI si a est **inférieur** à 3. FAUX sinon.
// x vaut **FAUX**

<= Exemple : a = 3 ; x = **a<= 6** ; // rend VRAI si a est **inférieur ou égal** à 6.
//FAUX sinon.
// x vaut **VRAI**

== Exemple : a = 6 ; x = **a == 5** ; // rend VRAI si a est **égal** à 5.
// FAUX sinon.
// x vaut **FAUX**

**!=** Exemple : a = 4 ; x = **a != 2** ; // rend VRAI si a est **différent** de 2.
//FAUX sinon.
// x vaut **VRAI**

**&&** Exemple : a = 9 ; b = 1 ; // **ET LOGIQUE :** rend VRAI si les
x = **(a == 9) && (b != 8)**; // deux tests sont VRAI.
// FAUX sinon.
// x vaut **VRAI**

|| Exemple : a = 6 ; b = 3 ; // **OU LOGIQUE :** rend VRAI si au
x = **(a == 5) || (b != 3)**; // moins un des deux tests
// est VRAI. FAUX sinon.
// x vaut **FAUX**

**!** Exemple : a = 1 ; x = **!a** ; // **NEGATION LOGIQUE :** rend VRAI si
// a est FAUX. FAUX sinon.
y = **!(a == 5)** ; // x vaut **FAUX** et y vaut **VRAI**

Remarques :

Les opérateurs ont une priorité. Cette priorité n'est pas forcément celle des mathématiques et varie d'un langage informatique à un autre. Il vaut donc mieux utiliser les parenthèses pour éviter tous problèmes.

La construction Si-Alors-Sinon est l'instruction conditionnelle la plus élémentaire en C et dans tous les langages en général. La syntaxe est :

```
if (<expression conditionnelle>) { <instructions si condition vraie> }
else { <instructions si condition fausse> }
```

Si la condition est vraie alors c'est le premier bloc d'instruction qui est exécuté, sinon (**else**) c'est le second bloc. Cependant, le second bloc est optionnel et la syntaxe devient :

**if (<expression conditionnelle>) { <instructions si condition vraie> }**

La valeur d'une expression conditionnelle est de type entier qui vaut soit 1 si la condition est vraie et qui vaut 0 dans le cas contraire. Cette expression repose sur l'utilisation d'opérateurs de comparaison et logiques.

## 12.5. Opérateurs de comparaison et logiques

Il existe deux catégories d'opérateurs pour les expressions conditionnelles : les opérateurs de comparaison qui permettent la comparaison de valeurs ; les opérateurs logiques qui permettent la combinaison de plusieurs expressions logiques.

L'instruction switch est une variante des enchaînements de if-then-else mais à la différence que les expressions conditionnelles se réduisent à une comparaison d'entiers. La syntaxe est la suivante :

```
switch(<expression entière>)
{
case <valeur entière>:
<bloc d'instructions>
break;
case <valeur entière>:
<bloc d'instructions>
break;
[…]
case <valeur entière>:
<bloc d'instructions>
break;
default:
<bloc d'instructions>
break;
}
```

## 12.6. Les boucles et les itérations

Les boucles et itérations offrent la possibilité de répéter et d'exécuter en série un bloc d'instructions sans avoir à le réécrire. Ces boucles et itérations sont

exécutées autant fois que la condition d'arrêt n'a pas été vérifiée. Et parfois, les boucles tournent … en rond !!!

### 12.6.1 La boucle While

La première instruction de boucle est le **while** qui signifie **tant que**. La syntaxe est :

```
while (<expression conditionnelle>)
{
<instructions>
}
```

Le principe est le suivant : tant que la condition est vraie le bloc d'instruction est exécuté.

### 12.6.2. La boucle Do-While

Il existe une autre forme de la boucle qui est de la forme **faire-tant-que**. Dans cette forme ,l'expression conditionnelle est évaluée après l'exécution du bloc d'instruction. La syntaxe devient alors :

```
do
{
<instructions>
} while(<expression conditionnelle>);
```

Dans certains cas, il peut être utile d'exécuter une fois le bloc d'instruction avant de réaliser l'évaluation de l'expression conditionnelle.

### 12.6.3. La boucle for

La boucle **for** est une variante de la boucle **while** intégrant un compteur de boucle, fort utile lorsque l'on connaît par avance le nombre d'itération nécessaire. La syntaxe est la suivante :

```
for(<instructions>; <expression conditionnelle>; <instructions>)
{
<instructions>
}
```

Le premier membre du for offre un espace pour l'initialisation de variables (en général le compteur de boucle). Le second membre est réservé à l'expression conditionnelle qui permet d'arrêter la boucle. Le dernier membre peut contenir n'importe quelle instruction. Il est en général réservé pour l'incrémentation de la boucle for. Les différents membres peuvent être vides.

## 13. Les Fonctions

### 13.1. Présentation

Un programme en C est un ensemble de fonctions :

- La fonction principale main qui est la première fonction appelée lors de l'exécution du programme.
- Les fonctions écrites par le programmeur qui doivent être déclarées avant leur appel.

- Fonctions prédéfinies issues des bibliothèques standards du compilateur (dont le code n'est pas écrit par le programmeur mais inséré dans le programme par l'éditeur de liens grâce au fichier.h).

Pour qu'un programme soit structuré, chaque fonction doit effectuer une tâche bien spécifique.

### 13.2. Syntaxe d'écriture d'une fonction

<type de la valeur de retour> **nom_fonction(**<liste des paramètres reçus par la fonction>**)**

**{**
**définition des variables locales ;**

**instructions ;**

**}**

### 13.3. Appel de fonction avec passage de paramètres par valeur

La valeur du paramètre passé est recopiée dans une variable locale à la fonction appelée. Une modification de cette variable n'a aucun effet sur la variable la fonction appelante.

Exemple de fonction main appelant Ma_fonction_somme :

```
void main()

{

int a,x,y,z ; // déclaration de quatre entiers a, x, y et z;

x = 1 ; // initialisation des variables locales

        y = 2 ;

        a = 5 ;

z = Ma_fonction_somme(x,y) ; // z vaut la somme de x et de y.

}
```

Dans la fonction main, la variable locale **a** vaut **5**.

Dans la fonction *Ma_fonction_somme*, la variable locale **a** vaut **1** et la variable locale **b** vaut **2**.

Après appel de la fonction *Ma_fonction_somme*, la variable **a** de la fonction main vaut **toujours 5**.

Bien sûr, z vaut **3**.