LE LANGAGE C POUR L'EMBARQUE

Patrice KADIONIK
http://www.enseirb.fr/~kadionik

D'après le cours originel de M. Touraïvane, Maître de Conférence à l'ESIL

© pk-mt/enseirb/2005

PLAN

1. Objectifs ... 9

2. Introduction ... 10

- 2.1. Historique ... 10
- 2.2. C K&R ... 11
- 2.3. C ANSI ... 13
- 2.4. C++ ... 14
- 2.5. Le langage C par rapport aux autres langages de programmation ... 15
- 2.6. Avantages du langage C ... 16
- 2.7. Inconvénients du langage C ... 18

3. Les principes de base de la compilation ... 21

- 3.1. Quelques généralités ... 21
- 3.2. Programme C : Hello world ... 23
- 3.3. Compilation et édition de liens ... 25

4. Remarques générales ... 29

- 4.1. Les fichiers include ... 29
- 4.2. Les commentaires ... 30

© pk-mt/enseirb/2005

5. Eléments de base 32

5.1. Les types de données élémentaires 32

5.2. Les entiers 38
- ❖ Le type char 39
- ❖ Les types short, long ou int 41
- ❖ Le type réel 41

5.3. Les constantes littérales 43
- ❖ Les constantes de type caractère 43
- ❖ Les chaînes de caractères 45
- ❖ Les booléens 46

5.4. Les variables 47

6. Opérateurs et expressions 51

6.1. Généralités sur les opérateurs 51
- ❖ Opérateurs et priorités 52
- ❖ L'opérateur d'affectation 54
- ❖ Les opérateurs arithmétiques 56
- ❖ Les opérateurs de comparaison 59
- ❖ Les opérateurs logiques 61
- ❖ Les opérateurs de manipulation de bits 62
- ❖ Les autres opérateurs binaires d'affectation 67
- ❖ Les autres opérateurs unaires d'affectation 68
- ❖ Autres opérateurs 69

6.2. Conversion de types 75
- ❖ Les conversions implicites 76
- ❖ Les conversions explicites 78

6.3. Récapitulatif 79

© pk-mt/enseirb/2005

7. Structures de contrôle de programme 83

7.1. L'instruction if 83

7.2. Les instructions while, do et for 88

7.3. L' instruction goto 90

8. Structures de programmes et fonctions 92

8.1. Introduction 93
- ❖ Déclaration de fonction 97
- ❖ Arguments formels 97
- ❖ Arguments effectifs 97
- ❖ Type de la fonction 98

8.2. Appels de fonction et gestion de la pile 98
- ❖ Passage par valeur 99
- ❖ Passage par adresse 101

8.3. Divers 105
- ❖ Fonction sans argument et/ou qui ne retourne rien 105
- ❖ Fonction récursive 106

9. Visibilité des données 107

9.1. Variables globales et locales 107

9.2. Variables globales privées et publiques 110

9.3. Fonctions privées et publiques 111

10. Pointeurs 113

10.1. Introduction 113

10.2. Pointeurs et tableaux 115

© pk-mt/enseirb/2005

10.3. Le type void 116

11. Les entrées sorties haut niveau formatées 118
11.1. Les E/S formatés. Les E/S bas niveau 118
11.2. Le tampon d'entrée sortie des appels d'E/S formatés........ 120
11.3. Fonctions générales sur les flots 123
❖ fopen........ 124
❖ fclose........ 128
❖ fflush 129
❖ feof........ 129
11.4. Les flots standards d'E/S........ 130
11.5. Lecture et écriture en mode caractère 131
❖ fputc 131
❖ fgetc........ 131
❖ Les macros putc et getc 132
❖ getchar 132
❖ putchar........ 133
11.6. Lecture et écriture en mode chaîne de caractères........ 133
❖ fgets 133
❖ fputs........ 134
❖ gets........ 135
❖ puts........ 135
11.7. Lecture et écriture formatée 136
❖ Ecriture formatée avec printf........ 137
❖ Ecriture formatée avec scanf........ 140

12. Les entrées sorties bas niveau 141
❖ open et create........ 142
❖ read et write 144

© pk-mt/enseirb/2005

❖ ioctl........ 147
❖ close........ 149

13. Comparaison entre les E/S bas niveau et haut niveau 150
13.1. Cas de Linux........ 150
13.2. Comparaisons par l'exemple 153

14. Programmation C avancée 162
14.1. Arguments passés à un programme........ 162
14.2. Options de compilation........ 164

15. Les bibliothèques standards........ 168
15.1. Entrées sorties <stdio.h>........ 169
❖ Opérations sur les fichiers........ 169
❖ Accès aux fichiers........ 170
❖ E/S formatées et bufferisées........ 171
❖ E/S mode caractère........ 172
❖ E/S mode binaire........ 173
❖ Position dans un fichier........ 173
❖ Gestion des erreurs........ 174
15.2. Mathématiques <math.h> 175
❖ Fonctions trigonométriques et hyperboliques........ 176
❖ Fonctions exponentielles et logarithmiques........ 177
❖ Fonctions diverses........ 178
15.3. Manipulation de chaînes de caractères <string.h> 179
15.4. Manipulation de caractères <ctype.h>........ 180
15.5. Utilitaires divers <stdlib.h>........ 182

© pk-mt/enseirb/2005

- Conversion de nombres ... 182
- Génération de nombres pseudo aléatoires ... 183
- gestion de la mémoire ... 183
- Communication avec l'environnement ... 184
- Arithmétique sur les entiers ... 184

16. Exemples de programmes C pour l'embarqué ... 185

16.1. Exemple 1 : programmation mémoire EPROM ... 185
16.2. Exemple 2 : pilotage d'un module par la liaison série ... 188
16.3. Exemple 3 : bibliothèque de contrôle d'une carte VME ... 192
16.4. Exemple 4 : pilotage de la liaison série d'un microcontrôleur ... 196
16.5. Exemple 5 : pilotage d'un périphérique d'E/S. Structure C ... 208

17. Quelques pièges classiques ... 211

17.1. Erreurs sur l'affectation ... 211
17.2. Erreurs avec les macros ... 211
- Un `#define` n'est pas une déclaration ... 211
- Un `#define` n'est pas une initialisation ... 212

17.3. Erreurs avec l'instruction `if` ... 213
17.4. Erreurs avec les commentaires ... 213
17.5. Erreurs avec les priorités des opérateurs ... 214
17.6. Erreurs avec l'instruction `switch` ... 215
- Oubli du `break` ... 215
- Erreurs sur le `default` ... 216

17.7. Erreurs sur les tableaux multidimensionnels ... 217

© pk-mt/enseirb/2005

18. Bibliographie ... 218

19. Webographie ... 218

© pk-mt/enseirb/2005

Ce cours a été rédigé à partir du cours originel de **M. Touraïvane**, Maître de Conférence à l'ESIL (http://www.esil.univ-mrs.fr/~tourai/), revu et complété pour introduire les points importants du langage C pour l'embarqué.

1. OBJECTIFS

1. Balayer et revoir les aspects **importants et essentiels** du langage C que doit maîtriser tout ingénieur électronicien afin de concevoir le logiciel de base d'un système numérique (système embarqué). Les bases du langage C ont été vues en première année…
2. Connaître les points forts et les points faibles du langage C. Eviter les pièges classiques.
3. Maîtriser les appels d'E/S de base et formatés en langage C. Intérêt pour les systèmes embarqués.
4. Comprendre comment on développe une application embarquée en langage C à travers des exemples.

© pk-mt/enseirb/2005

2. INTRODUCTION

2.1. Historique

Aucun langage de programmation n'a pu se vanter d'une croissance en popularité comparable à celle de C.

Le langage C n'est pas un nouveau né dans le monde informatique :
- 1972 : développement par ATT Bell Laboratories.

But :
- Développer une version portable d'un système d'exploitation ouvert qui ne plante pas (UNIX) (Kernighan et Ritchie) sur PDP7 afin de mettre en commun les imprimantes des secrétaires du laboratoire.

© pk-mt/enseirb/2005

Le langage C par rapport à d'autres langages pour comparaison :
- C++ : 1988, 1990
- Fortran : 1954, 1978 (Fortran 77), 1990, 1995 (Fortran 1995)
- Cobol : 1964, 1970
- Pascal : 1970
- Lisp : 1956, 1984 (CommonLisp)

2.2. C K&R

1978 : Kernighan et Ritchie publient la définition classique du langage C (connue sous le nom de standard K&R-C) dans un livre intitulé *The C Programming Language*.

Anecdote :
Remarquer le style d'indentation d'un source en langage C : K&R vs C-ANSI :

© pk-mt/enseirb/2005


```
for(i=0 ; i<100 ; i++) {
   …
}         K&R
```

```
for(i=0 ; i<100 ; i++)
   {
   …
   }      C-ANSI
```


© pk-mt/enseirb/2005

2.3. C ANSI

Nécessité la définition d'un standard actualisé et plus précis.

1983 : l'*American National Standards Institute* (ANSI) charge une commission de mettre au point une définition explicite et indépendante machine pour le langage C (**standardisation des appels d'E/S. Il n'y a pas de mots clés réservés read, readln write, writeln comme avec le langage Pascal…**) : naissance du standard C-ANSI (*ANSI-C*). Dans le cas d'un système embarqué programmé en C, il faut même se poser la question de l'existence du "printf()" ! Voir plus loin…

1988 : seconde édition du livre *The C Programming Language* respectant le standard - C-ANSI. C'est la bible du programmeur en C (Le langage C. K&R. Editions Masson).

© pk-mt/enseirb/2005

2.4. C++

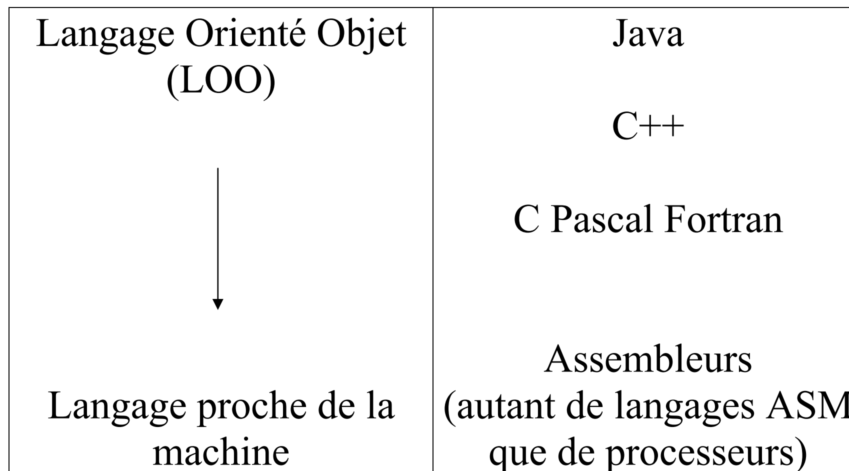
1983 : création du langage C++ par un groupe de développeurs de AT&T sous la direction de Bjarne Stroustrup.

But :
- Développer un langage qui garde les avantages de ANSI-C mais orienté objet. Depuis 1990, il existe une ébauche pour un standard ANSI-C++.

Remarque :
- C++ est un sur-ensemble du C si bien que 80 % des gens qui déclarent développer en C++ développent en fait en C !

© pk-mt/enseirb/2005

2.5. Le langage C par rapport aux autres langages de programmation



Les langages précédents sont orientés informatique scientifique et technique.
Il existe des langages dédiés : SQL, COBOL, ADA, Scheme, HTML, Perl…

© pk-mt/enseirb/2005

2.6. Avantages du langage C

C est un langage :

(1) universel :
C n'est pas orienté vers un domaine d'applications spéciales comme SQL par exemple…

(2) compact :
C est basé sur un noyau de fonctions et d'opérateurs limités qui permet la formulation d'expressions simples mais efficaces.

(3) moderne :
C est un langage structuré, déclaratif et récursif. Il offre des structures de contrôle et de déclaration comme le langage Pascal.

© pk-mt/enseirb/2005

(4) près de la machine :

C offre des opérateurs qui sont très proches de ceux du langage machine (manipulations de bits, pointeurs…). C'est un atout essentiel pour la programmation des systèmes embarqués.

(5) rapide :

C permet de développer des programmes concis et rapides.

(6) indépendant de la machine :

C est un langage près de la machine (microprocesseur) mais il peut être utilisé sur n'importe quel système ayant un compilateur C.
Au début C était surtout le langage des systèmes UNIX mais on le retrouve comme langage de développement aujourd'hui de tous les systèmes d'exploitation.

© pk-mt/enseirb/2005

(7) portable :

En respectant le standard ANSI-C, il est possible d'utiliser (théoriquement ☺) le même programme sur tout autre système (autre hardware, autre système d'exploitation), simplement en le recompilant. Il convient néanmoins de faire attention dans le cas d'un système embarqué qui n'inclut pas toujours un système d'exploitation…

(8) extensible :

C peut être étendu et enrichi par l'utilisation de bibliothèque de fonctions achetées ou récupérées (logiciels libres…).

2.7. Inconvénients du langage C

(1) efficacité et compréhensibilité :

C autorise d'utiliser des expressions compactes et efficaces. Les programmes sources doivent rester compréhensibles pour nous-mêmes et pour les autres.

© pk-mt/enseirb/2005

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

Exemple :

Les deux lignes suivantes impriment les N premiers éléments d'un tableau A[] en insérant un espace entre les éléments et en commençant une nouvelle ligne après chaque dixième chiffre :

```
for (i=0; i<n; i++)
     printf("%6d%c", a[i], (i%10==9)?'\n':' ');
```

Cette notation est très pratique mais plutôt intimidante pour un débutant. L'autre variante est plus lisible :

```
for (I=0; I<N; I=I+1){
      printf("%6d", A[I]);
      if ((I%10) == 9)
          printf("\n");
      else
          printf(" ");
}
```


© pk-mt/enseirb/2005

☹ La programmation efficace en langage C nécessite beaucoup d'expérience.

Sans commentaires ou explications, les fichiers sources C peuvent devenir incompréhensibles et donc inutilisables (maintenance ?).

(2) portabilité et bibliothèques de fonctions :

- La portabilité est l'un des avantages les plus importants de C : en écrivant des programmes qui respectent le standard ANSI-C, nous pouvons les utiliser sur n'importe quelle machine possédant un compilateur ANSI-C.
- Le nombre de fonctions standards ANSI-C (d'E/S) est limité. La portabilité est perdue si l'on utilise une fonction spécifique à la machine de développement.

(3) discipline de programmation :

- C est un langage près de la machine donc dangereux bien que C soit un langage de programmation structuré.

© pk-mt/enseirb/2005

- C est un langage qui n'est pas fortement typé comme par exemple Pascal ou Java **(ou VHDL**).
- C n'inclut pas de gestion automatique de la mémoire comme Java. **La gestion mémoire en C est la cause de beaucoup de problèmes** (certains ont peut être eu une mauvaise expérience avec les *malloc()/free()*)…
- L'instruction *goto* existe en C.

3. LES PRINCIPES DE BASE DE LA COMPILATION

3.1. Quelques généralités

Un langage de programmation a pour finalité de communiquer avec la machine. Il y a diverses manières de communiquer avec la machine.

© pk-mt/enseirb/2005

Le langage naturel de la machine n'utilise que deux symboles (0 et 1) : c'est le langage machine. Le langage machine ne comprend et n'exécute qu'une suite de 0 et 1 structurée sous forme d'octets (8 bits).

Le langage assembleur permet à l'humain de générer cette suite de 0 et 1 à partir d'un fichier source compréhensible qui utilise des instructions assembleur. Le programme assembleur ou assembleur par abus de langage traduit le source assembleur en code objet (suite de 0 et 1) exécutable par la machine.

Le langage C va opérer la même chose que précédemment mais avec plus de lisibilité encore.

Un programme C est un texte écrit avec un éditeur de texte respectant une certaine syntaxe et stocké sous forme d'un ou plusieurs fichiers (généralement avec l'extension .c).

© pk-mt/enseirb/2005

A l'opposé du langage assembleur, les instructions du langage C sont obligatoirement encapsulées dans des fonctions et il existe une fonction privilégiée appelée *main()* qui est le point d'entrée de tout programme C (sous UNIX pour l'instant). **Dans le cas d'un système embarqué sans système d'exploitation, la fonction spécifique main() perd son sens. Le point d'entrée du programme sera celui spécifié (en mémoire ROM) dans la table des vecteurs pour l'initialisation du compteur programme au RESET**…

3.2. Programme C : Hello world

Fichier source C hello.c :

```
#include <stdio.h>

int main() {
        printf("Hello world \n");
        return(0) ;
}
```


© pk-mt/enseirb/2005

Affichage de "Hello world " à l'écran.

Pour afficher cette chaîne, le programme fait appel à la fonction `printf` qui fait partie des appels d'E/S standards définis par la norme C-ANSI.
L'ensemble de ces fonctions standards (appelé bibliothèque C) est stocké dans un ou plusieurs fichiers (ici `libc.a`).

On utilisera indifféremment fonction à la place d'appel système bien que la dernière expression soit la plus correcte sous UNIX.

© pk-mt/enseirb/2005

3.3. Compilation et édition de liens

La traduction du fichier texte ci-dessus en un programme exécutable se décompose en deux phases :

1. la compilation qui est la traduction d'un programme C en une suite d'instructions machine. Le résultat produit est un fichier objet (généralement avec l'extension .o).
2. l'édition de liens produit à partir d'un ou de plusieurs fichiers objets et des bibliothèques un fichier exécutable.

Outre la compilation des divers fichiers objets, l'édition des liens inclut le code objet des fonctions standards utilisées par le programme.

Compilation de hello.c (sous UNIX qui est la plateforme traditionnelle de développement) :

```
% cc hello.c [-lc]             [-lm pour libm.a]
   création de l'exécutable a.out par défaut
```


© pk-mt/enseirb/2005

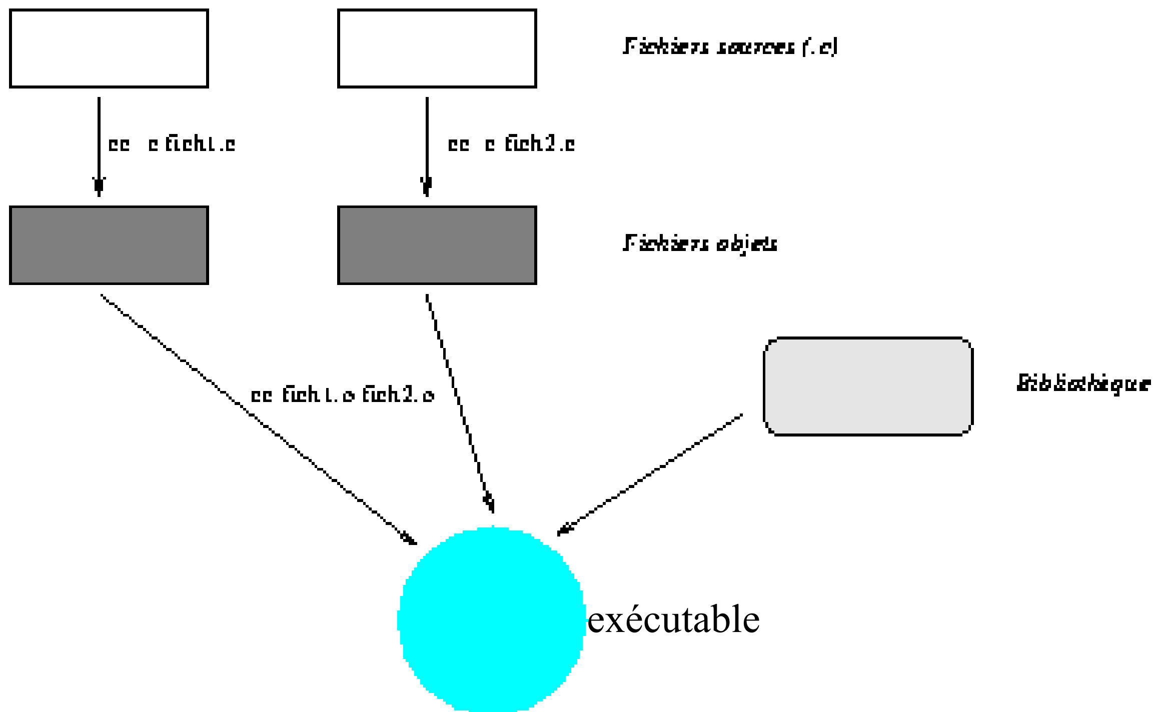

```
% cc hello.c -o hello
   création de l'exécutable hello
% hello
ou
% ./hello
   exécution du programme depuis le shell
```

Les compilateurs C font subir deux transformations aux fichiers sources :

1. un préprocesseur réalise des transformations d'ordre purement textuel pour rendre des services du type inclusion de source, compilation conditionnelle et traitement de macros.
2. le compilateur C proprement dit prend le texte généré par le préprocesseur et le traduit en code objet.

La fonction de préprocesseur est assez souvent implémentée par un programme séparé (cpp sous UNIX) qui est automatiquement appelé par le compilateur C.

© pk-mt/enseirb/2005



© pk-mt/enseirb/2005

Chaque fichier source (ou texte) est appelé *module* et est composé :

- ♦ des définitions de fonctions
- ♦ des définitions de variables
- ♦ des déclarations de variables et fonctions externes
- ♦ des directives pour le préprocesseur (lignes commençant par #) de définitions de types ou de constantes (struct, union...).

C'est la base de la programmation modulaire !

© pk-mt/enseirb/2005

4. REMARQUES GENERALES

4.1. Les fichiers include

Pour compiler correctement un fichier, le compilateur a besoin d'informations concernant les déclarations des structures de données et de variables externes ainsi que de l'aspect (on dira *prototype* ou *signature*) des fonctions prédéfinies.
Un programme de compose donc :
- De structures de données.
- D'algorithmes utilisant ces structures de données.

Ces informations sont contenues dans des fichiers *header* avec l'extension .h. Ces fichiers doivent être inclus dans le fichier source que l'on veut compiler.
Pour cela, le langage C offre la directive du préprocesseur :

```
#include nom_de_fichier
```


© pk-mt/enseirb/2005

Exemple pour utiliser l'appel printf (affichage à l'écran) :

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

`stdio.h` : contient les déclarations de variables externes et les prototypes de fonctions de la bibliothèque d'E/S formatées (*standard input output*).

4.2. Les commentaires

Dès lors que l'on écrit un programme important, il est indispensable d'y inclure des commentaires qui ont pour but d'expliquer ce qu'est sensé faire le programme rendant le programme lisible à soi même et à autrui ☺. **Il est important d'assurer la maintenance du code ; ce qui est bien programmer !**

Un commentaire commence par les caractères /* et se termine par */.

© pk-mt/enseirb/2005

L'erreur classique avec les commentaires est d'oublier la séquence fermante */.

Dans ce cas, le compilateur va considérer que le commentaire se poursuit jusqu'à la fin du prochain commentaire. Ceci peut ne pas générer d'erreur syntaxique ☹.

Exemple :

```
instruction
/* premier commentaire
instruction
...
instruction
/* second commentaire */
instruction
```


© pk-mt/enseirb/2005

On voit que dans ce cas, tout un ensemble d'instructions sera ignoré par le compilateur sans générer le moindre message d'erreur.

5. ELEMENTS DE BASE

5.1. Les types de données élémentaires

Un programme C manipule deux types de données de base :
- les nombres entiers.
- les nombres flottants.

Un autre type de donnée élémentaire est le *pointeur* (adresse d'une variable).

Toute autre structure de donnée sera dérivée à partir de ces types fondamentaux.
Par exemple, les caractères, les booléens, les constantes ... ne sont rien d'autres que des nombres.

© pk-mt/enseirb/2005

Notation binaire, décimale, hexadécimale, octal

Rappelons qu'un nombre n en notation décimale est représenté en base b par le nombre $a_i\ a_{i-1} \ldots a_1\ a_0$
où :
$n = a_m * b^m + a_{m-1} * b^{m-1} + \ldots + a_1 * b^1 + a_0\ b^0$ avec $0 <= a_i < b$.
Voici une liste des bases habituellement utilisées en (micro)informatique :

Base	Notation	Symboles
2	binaire	0, 1
8	octale	0, 1, ... 7
10	décimale	0, 1, ... 9
16	hexadécimale	0, 1, ... 9, a, b, c, d, e, f

© pk-mt/enseirb/2005

Exemple :

Le nombre 70 (en notation décimale) est représenté par 1000110 en notation binaire, 0106 en notation octale, 0x46 en notation hexadécimale.

Représentation des nombres signés (positifs et négatifs)

On appelle bit de signe, le bit le plus élevé (*bit de poids fort ou MSB Most Significant Bit*) d'un nombre entier.

Si l'on considère un nombre codé sur un octet, lorsque le bit de signe vaut 0, il s'agit d'un nombre positif. Inversement lorsque ce bit vaut 1, il s'agit d'un nombre négatif.

© pk-mt/enseirb/2005

Exemple sur 8 bits :

```
01111111: +127
10000000: -128
11111111 : -1
00000000:  0 (+0, -0)
```

Ce format de représentation d'un entier est **appelé complémentation à 2** qui permet d'avoir une seul unité arithmétique et logique (ALU) pour manipuler des nombres entiers positifs ou négatifs, une soustraction se ramenant à une addition et assure l'unicité de représentation du nombre 0.

Entiers non signés :

Dans un octet, il est possible de ranger 2^8 valeurs différentes. Si l'on décide que cet octet est susceptible de contenir des entiers positifs et négatifs (le bit de signe est occupé), on codera les valeurs comprises entre -2^7. et 2^7-1. Inversement, si l'on décide

© pk-mt/enseirb/2005

que cet octet ne contient que des entiers sans signe (positifs ou nuls), on codera les valeurs comprises entre 0 et 2^8-1.

C'est à nous de savoir à priori si l'on travaille sur un nombre entier signé ou non signé (structure de données) et non le processeur.

La différence entre ces 2 types apparaît au niveau des tests d'inégalité :

```
char c;        /* nombre signé */

   c = 129;

   if(c >= 128) {
      Je fais qq chose;  /* Est-on vraiment sûr de rentrer ici ? */
   }
```


© pk-mt/enseirb/2005

```
unsigned char c;

   c = 129;

   if(c >= 128) {
      Je fais autre chose;    /* oui… */
   }
```

Le langage C considère par défaut les variables de base comme signées. Dans les structures de données utilisées par un programme, il faut se poser la question du type signé ou non. **Si l'on travaille en valeur absolue (non signé), il est obligatoire d'ajouter la qualificatif *unsigned* sous peine d'effets de bord en cours d'exécution !**

© pk-mt/enseirb/2005

5.2. Les entiers

En C, on dispose de divers types d'entiers qui se distinguent par la place qu'ils occupent en mémoire :

- sur 1 octet, les entiers *signés* et *non signés* (`char`) et (`unsigned char`).
- sur 2 octets, les entiers *signés* et *non signés* (`short`) et (`unsigned short`).
- sur 4 octets (**en général**), les entiers *signés* et *non signés* (`long`) et (`unsigned long`).
- le type `int` (`unsigned int`) est selon les machines synonyme de `short` (`unsigned short`) ou de `long` (`unsigned long`).

Il faudra à priori s'assurer de la taille de représentation du type *int* sur sa machine en utilisant par exemple l'instruction sizeof :

```
printf("int=%d octets\n", sizeof(int));
```


© pk-mt/enseirb/2005

❖ Le type char

Le type `char` désigne un nombre **entier signé codé sur 1 octet**.

Décimal	Caractère
	NULL
...	...
48	
...	...
57	9
...	...
65	A
...	...
90	Z
...	...

© pk-mt/enseirb/2005

97	a
...	...
122	z
...	...
127	

Code ASCII

Toutes les opérations autorisées sur les entiers peuvent être utilisées sur les caractères : on peut ajouter ou soustraire deux caractères, ajouter ou soustraire un entier à un caractère ☺.

Exemple :

Conversion du caractère *c* désignant un chiffre en sa valeur numérique *v* correspondante :

```
v = c - '0';
```


© pk-mt/enseirb/2005

❖ Les types short, long ou int

Le type `short` représente un entier signé codé sur 2 octets (de -32768 à 32767) et le type `unsigned short` représente un entier non signé codé sur 2 octets (de 0 à 65535).
Le type `long` (ou `int` suivant la machine) représente un entier signé codé sur 4 octets (de -2147843648 à 2147843647) et le type `unsigned long` (ou `unsigned int`) représente un entier non signé codé sur 4 octets (de 0 à 4294967295).

❖ Le type réel

Les nombres à *virgule flottante* (abusivement appelés réels) servent à coder de manière approchée les nombres réels. Un nombre à virgule flottante est composée d'un signe, d'une mantisse et d'un exposant.
On dispose de trois types de nombres à virgule flottante :

© pk-mt/enseirb/2005

- `float`
- `double`
- `long double`

Les floats :

Un `float` est codé sur 4 octets avec 1 bit de signe, 23 bits de mantisse et 8 bits d'exposant (valeurs comprises entre $-3.4 * 10^{-38}$ et $+3.4 * 10^{38}$).

Les doubles :
Un `double` est codé sur 8 octets avec 1 bit de signe, 52 bits de mantisse et 11 bits d'exposant (valeurs comprises entre $-1.7 * 10^{-308}$ et $+1.7 * 10^{308}$).

Les long doubles :
Un `long double` est codé sur 10 octets avec 1 bit de signe, 64 bits de mantisse et 15 bits d'exposant (valeurs comprises entre $-3.4 * 10^{-4932}$ et $+3.4 * 10^{4932}$).

© pk-mt/enseirb/2005

Remarques :

- La représentation d'un nombre réel est normalisé : norme IEEE754/854.
- Les nombres à virgule flottante sont des valeurs approchées. En particulier, les opérations sur ces nombres peuvent conduire à des erreurs d'arrondis.
- On ne peut pas travailler en langage C sur des nombres réels à virgule fixe.

5.3. Les constantes littérales

❖ Les constantes de type caractère

Les constantes de type caractère se note entre apostrophes ' :
`'a'  '2'  '"'`

© pk-mt/enseirb/2005

Le caractère `'` se note `'\''` et le caractère `\` se note `'\\'`. On peut également représenter des caractères non imprimables à l'aide de *séquences d'échappement :*

Séquence	
`\n`	nouvelle ligne
`\t`	tabulation horizontale
`\v`	tabulation verticale
`\b`	retour d'un caractère en arrière
`\r`	retour chariot
`\f`	saut de page
`\a`	beep
`\'`	apostrophe
`\"`	guillemet
`\\`	anti-slash

Séquences d'échappement

© pk-mt/enseirb/2005

❖ Les chaînes de caractères

Les chaînes de caractères se note entre guillemets :

```
"coucou"
"C'est bientot fini !!!"
```

Une chaîne de caractères est une suite de caractères (éventuellement vide) entre guillemets. Il en découle que l'on est autorisé à utiliser les séquences d'échappement dans les chaînes.

En mémoire, une chaîne de caractères est une suite de caractères consécutifs et dont le dernier élément est le caractère nul `'\0'`.

	c	o	u	c	o	u	\n	\0	

"coucou\n"

© pk-mt/enseirb/2005

Une chaîne de caractère doit être écrite sur une seule ligne. Lorsqu'il est trop long pour tenir une même ligne, on découpe celle-ci en plusieurs bouts. Chaque bout étant écrite sur une seule ligne et on masque le retour à la ligne par le caractère `\`.

❖ Les booléens

Contrairement à d'autres langages (comme Pascal), il n'y a pas de type booléen en C.

Le type booléen est représenté par un entier. Il se comporte comme la valeur booléenne *vraie* si **cette valeur entière est non nulle**.

Dans un contexte qui exige une valeur booléenne (comme les tests, par exemple), un entier **non nul équivaut à *vrai* et la valeur nulle équivaut à *faux*.**

De même, une fonction qui retourne une valeur booléenne pourra retourner une valeur non nulle comme équivalent à *vrai* et la valeur 0 comme équivalent à *faux*.

© pk-mt/enseirb/2005

5.4. Les variables

Une variable possède :

- Un nom (un *identificateur*) composé d'une suite de caractères commençant par un caractère alphabétique et suivi de caractères alphanumériques ou du caractère _.

Exemple :

```
x, x1, x_1, Var, VarLocal, var_local
```

Contrairement aux constantes, le *contenu* d'une variable peut être modifiée à volonté ; ce qui ne change pas c'est l'adresse de la variable.

© pk-mt/enseirb/2005

Définition et déclaration de variables :

En C, toute variable utilisée dans un programme doit auparavant être définie.

La *définition* d'une variable consiste à la nommer et lui donner un type et éventuellement lui donner une valeur initiale (on dira *initialiser*). C'est cette définition qui réserve (on dira *alloue*) la place mémoire nécessaire en fonction du type.

Initialiser une variable consiste à remplir, avec une constante, la zone mémoire réservée à cette variable. Cette opération s'effectue avec l'opérateur =

Il ne faut pas confondre l'initialisation et affectation. Malheureusement, ces deux opérations utilisent le même symbole =.

© pk-mt/enseirb/2005

Exemple :

```
int x = 2;
char c = 'c';
float f = 1.3;
```

Lorsque le programme que l'on réalise est décomposé en plusieurs modules, une même variable, utilisée dans plusieurs modules, doit être déclarée dans chacun de ces modules. Par contre, on ne définira cette variable que dans un seul de ces modules. C'est au moment de l'édition des liens que l'on mettra en correspondance les variables apparaissant dans plusieurs modules (variable partagée, mot clé *extern*).

© pk-mt/enseirb/2005

Exemple :

```
int x, y = 0, z;
extern float a, b;
unsigned short cpt = 1000;
```

Portée des variables :
La position de la déclaration ou de la définition d'une variable détermine sa portée i.e. sa durée de vie et sa visibilité.

- Les variables *globales* sont déclarées en dehors de toute fonction.
- Les variables *locales* sont déclarées à l'intérieur des fonctions et ne sont pas visibles à l'extérieur de la fonction dans laquelle celle-ci est définie.

© pk-mt/enseirb/2005

6. OPERATEURS ET EXPRESSIONS

6.1. Généralités sur les opérateurs

Une expression est un objet syntaxique obtenu en assemblant des constantes, des variables et des opérateurs.

Exemple :

x+3

Dans le langage C, il y a bien d'autres opérateurs que les opérateurs arithmétiques qu'on a l'habitude de manipuler. Il y en a en fait plus de 40 opérateurs.

© pk-mt/enseirb/2005

❖ Opérateurs et priorités

Nous avons l'habitude de manipuler des expressions arithmétiques :
2+3*4*5-2 2-3-4

On sait que ces expressions sont équivalentes à (2+(3*(4*5)))-2, (2-3)-4.

Introduire les parenthèses permet de définir sans ambiguïté l'expression que l'on manipule.
Pour éviter l'usage des parenthèses qui alourdissent la lecture, il existe des règles pour lever toute ambiguïté :

Exemple :
2+3*4
la sous expression 3*4 est évaluée en premier et le résultat obtenu est ajouté à la valeur 2 (forme avec parenthèse : 2 + (3 * 4)).

© pk-mt/enseirb/2005

On dit que l'opérateur * possède une *priorité* supérieure à la priorité de l'opérateur +.

2-3-4

la sous expression 2-3 est évaluée en premier et au résultat obtenu, on soustrait la valeur 4 (forme avec parenthèse : (2 - 3) - 4).

On dit que l'*ordre* (d'évaluation) de l'opérateur - est de *gauche à droite*.

La donnée d'une *priorité* et d'un *ordre d'évaluation* permet de fixer des règles communes d'évaluation des expressions.

Ces priorités et ordre d'évaluation ne permettent évidemment pas de se dispenser complètement des parenthèses :

(2+3)*4 à comparer à 2+3*4 ☺

© pk-mt/enseirb/2005

ENSEIRB Le langage C pour l'embarqué pk

Caractéristiques de l'opérateur ( ) :

Opérateur	Nom	Notation	Priorité	Ordre
( )	parenthèses	(...)	15	gauche-droite

Système de classification des opérateurs C

❖ L'opérateur d'affectation

L'opération la plus importante dans un langage de programmation est celle qui consiste à donner une valeur à une variable.

Cette opération est désignée par le symbole =.

© pk-mt/enseirb/2005

Comme l'affectation range une valeur dans une variable (une zone mémoire), il est impératif que le membre gauche d'une affectation représente une zone mémoire (*left value*).

Une constante n'est pas une *left value* car il ne désigne pas l'adresse d'une zone mémoire. Elle ne peut donc pas figurer en membre gauche d'une affectation.

Le membre droit d'une affectation peut désigner soit une constante soit une zone mémoire soit une expression.

Opérateur	Nom	Notation	Priorité	Ordre
=	Affectation	$x = y$	2	droite-gauche

© pk-mt/enseirb/2005

❖ Les opérateurs arithmétiques

Opérateur	Nom	Notation	Priorité	Ordre
+	Addition	$x + y$	12	gauche-droite
-	soustraction	$x - y$	12	gauche-droite
*	multiplication	$x * y$	13	gauche-droite
/	division	x / y	13	gauche-droite
%	modulo	$x \% y$	13	gauche-droite

Les opérateurs +, −, * fonctionnent comme en arithmétique.

Par contre, l'opérateur / (division) se comporte de manière différente selon que les opérandes sont des entiers ou des nombres flottants :

- nombres flottants : le résultat est un nombre flottant obtenu en divisant les deux nombres.

© pk-mt/enseirb/2005

- nombres entiers : **le résultat est un nombre entier obtenu en calculant la division entière ou division euclidienne.** L'opérateur `%` n'est défini que pour les entiers et le résultat est le reste de la division entière des opérandes.

Rappel sur la division entière : On appelle quotient et reste de la division entière de a et de b , les nombres entiers q et r vérifiant :

$$a = q * b + r; 0 < r < b$$

Exemple :

*Si a =20 et b =3 alors q =6 et r =2 (20 = 6 * 3 + 2).*

Il faut aussi faire attention et savoir si l'on fait une division euclidienne ou réelle.

© pk-mt/enseirb/2005

Exemple :

```
int a, b, c ;
float x, y, z;

a = b / c;                  division entière
a = x / y;                  reste entièr
x = y / z;                  division réelle
x = 2 / 3;                  division entière
x = 2.0/3.0;                division réelle
x = (float) ((float) a)/y ; division réelle
```

L'opérateur unaire − a une priorité supérieure aux opérateurs binaires arithmétiques :

Opérateur	Nom	Notation	Priorité	Ordre
− (unaire)	négation	- *x*	14	droite-gauche

© pk-mt/enseirb/2005

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

❖ Les opérateurs de comparaison

Opérateur	Nom	Notation	Priorité	Ordre
==	test d'égalité	`x == y`	9	gauche-droite
!=	test de non égalité	`x != y`	9	gauche-droite
<=	test inférieur ou égal	`x <= y`	10	gauche-droite
>=	test supérieur ou égal	`x >= y`	10	gauche-droite
<	test inférieur strict	`x < y`	10	gauche-droite
>	test supérieur strict	`x > y`	10	gauche-droite

Théoriquement, le résultat d'une comparaison est une valeur booléenne (*vrai* ou *faux*). En C, le résultat d'une comparaison est 1 (!= 0) ou 0 selon que cette comparaison est *vraie* ou *fausse*.

© pk-mt/enseirb/2005

Il n'existe pas de type booléen en C :
La valeur entière 0 sera considérée comme équivalente à la valeur *faux* et toute valeur différente de 0 équivalente à la valeur *vrai*.

Piège :

Ne pas confondre == (test d'égalité) et = (affectation) ☺

Exemple :

```
if (x == 2) { ...     vs     if (x = 2) { ...
```

La première teste l'égalité de la valeur contenue dans la variable *x* et la valeur 2 , alors que la deuxième teste la valeur de l'affectation `x=2` qui vaut 2 quelle que soit la valeur de *x* (dans cet exemple).

© pk-mt/enseirb/2005

❖ Les opérateurs logiques

Une variable booléenne est une variable pouvant prendre la valeur *vrai* ou *faux*.

La valeur d'une expression booléenne est, comme le résultat des comparaisons, une valeur entière.

Opérateur	Nom	Notation	Priorité	Ordre
&&	ET	x && y	5	gauche-droite
||	OU	x || y	4	gauche-droite
! (unaire)	NON	! x	14	droite-gauche

© pk-mt/enseirb/2005

❖ Les opérateurs de manipulation de bits

Opérateur	Nom	Notation	Priorité	Ordre
&	ET bit à bit	x & y	8	gauche-droite
|	OU bit à bit	x | y	6	gauche-droite
^	OU exclusif bit à bit	x ^ y	7	gauche-droite
~ (unaire)	NON bit à bit	~ x	14	droite-gauche
	décalage à droite	x	11	droite-gauche
<<	décalage à gauche	<< x	11	droite-gauche

© pk-mt/enseirb/2005

Les manipulations de bits sont beaucoup utilisées dans l'embarqué. Pour contrôler un périphérique matériel, on retrouve des registres généralement de 8 bits appelés aussi **ports**. Cela permet généralement d'échanger des données avec le monde extérieur (liaison série, port parallèle…) :

- Des registres de donnée : **Data Register**. Exemple PADR (module PIT). Ce registre contient la donnée reçue ou à émettre.
- Des registres d'état : **Status Register**. Exemple PASR. Ce registre donne l'état courant du périphérique.
- Des registres de contrôle : **Control Register**. Exemple PACR. Ce registre permet de contrôler le périphérique.

Chaque bit d'un registre SR et CR correspond à une fonctionnalité ou état du périphérique.

© pk-mt/enseirb/2005

Exemple :

Validation d'une fonctionnalité. Mise à 1 du bit 7 du registre CR pour autoriser les interruptions par le périphérique :

```
unsigned char *PACR = (unsigned char *) 0xF500001 ;

*PACR  = *PACR | 0x80;    soit aussi *PACR |= 0x80;
```

Inhibition d'une fonctionnalité. Mise à 0 du bit 7 du registre CR pour interdire les interruptions par le périphérique :

```
unsigned char *PACR = (unsigned char *) 0xF500001 ;

*PACR  = *PACR & 0x7f;    soit aussi *PACR &= 0x7f;
```


© pk-mt/enseirb/2005

On notera que la somme des 2 constantes de masquage donne `0xff` !

Il faut aussi noter les différents types d'accès à un registre :
- Type 1 : registre accessible en lecture/écriture (registre R/W) et l'on peut lire exactement la dernière valeur écrite.
- Type 2 : registre accessible en lecture/écriture (registre R/W) et l'on ne relit pas la dernière valeur écrite. En cas de manipulation de bits sur ce registre, il est obligatoire de connaître la dernière valeur écrite !
- Type 3 : registre accessible en lecture seulement (registre Ronly).
- Type 4 : registre accessible en écriture seulement (registre Wonly). En cas de manipulation de bits sur ce registre, il est obligatoire de connaître la dernière valeur écrite !

<u>Exemple type 1 :</u>

Validation d'une fonctionnalité. Mise à 1 du bit 7 du registre R/W :

© pk-mt/enseirb/2005


```
unsigned char *PACR = (unsigned char *) 0xF500001 ;

*PACR |= 0x80;
```

<u>Exemple type 2 :</u>

Validation d'une fonctionnalité. Mise à 1 du bit 7 du registre R/W :

```
unsigned char *PACR = (unsigned char *) 0xF500001 ;
unsigned char tmp = 0x0c;

// Initialisation du registre
*PACR = tmp;
…
tmp |= 0x80;
*PACR = tmp;
```


© pk-mt/enseirb/2005

❖ Les autres opérateurs binaires d'affectation

Les opérateurs suivants ne sont que des raccourcis de notation (pas d'optimisation du code généré) :

<table>
<tr><th>Opérateur</th><th>équivalent</th><th>Notation</th><th>Priorité</th><th>Ordre</th></tr>
<tr><td>+=</td><td>x = x + y</td><td>x += y</td><td>2</td><td>droite-gauche</td></tr>
<tr><td>-=</td><td>x = x - y</td><td>x -= y</td><td>2</td><td>droite-gauche</td></tr>
<tr><td>*=</td><td>x = x * y</td><td>x *= y</td><td>2</td><td>droite-gauche</td></tr>
<tr><td>/=</td><td>x = x / y</td><td>x /= y</td><td>2</td><td>droite-gauche</td></tr>
<tr><td>%=</td><td>x = x % y</td><td>x %= y</td><td>2</td><td>droite-gauche</td></tr>
<tr><td>=</td><td>x = x y</td><td>x = y</td><td>2</td><td>droite-gauche</td></tr>
<tr><td><<=</td><td>x = x << y</td><td>x <<= y</td><td>2</td><td>droite-gauche</td></tr>
<tr><td>&=</td><td>x = x & y</td><td>x &= y</td><td>2</td><td>droite-gauche</td></tr>
<tr><td>|=</td><td>x = x | y</td><td>x |= y</td><td>2</td><td>droite-gauche</td></tr>
<tr><td>^=</td><td>x = x ^ y</td><td>x ^= y</td><td>2</td><td>droite-gauche</td></tr>
</table>

© pk-mt/enseirb/2005

❖ Les autres opérateurs unaires d'affectation

++	x = x + 1	x++ ou ++x	14	droite-gauche
--	x = x - 1	x-- ou --x	14	droite-gauche

`y = x++;` est le raccourci pour `y = x; x = x + 1;`
`y = ++x ;` est le raccourci pour `x = x + 1; y = x;`

© pk-mt/enseirb/2005

❖ Autres opérateurs

L'opérateur conditionnel

Opérateur ternaire (le seul dans C) :

`?:`	opérateur conditionnel	`e ? x : y`	3	droite-gauche

Cette expression est une sorte de *si alors sinon* sous forme d'expression : si la condition *e* est vraie alors cette expression vaut *x* sinon elle vaut *y* .

Exemple :

```
a = (v == 2) ? 1 : 2;
```

affecte la variable *a* à la valeur 1 si *v* vaut 2, sinon affecte la variable *a* à la valeur 2.

© pk-mt/enseirb/2005

L'opérateur séquentiel

`,`	opérateur séquentiel	`e1, e2`	1	droite-gauche

Cet opérateur permet de regrouper plusieurs expressions en une seule : l'évaluation de l'expression `e1, e2` consiste en l'évaluation successives (dans l'ordre) des expressions `e1` puis de `e2`.

L'opérateur de dimension

Cet opérateur donne l'occupation mémoire (en octets) d'une variable ou d'un type de donné.

`sizeof`	opérateur de dimension	`sizeof(e)`	14	droite-gauche

© pk-mt/enseirb/2005

Exemple :

La valeur de l'expression `sizeof(c)` est 1 si *c* est une variable de type `char`.
L'expression `sizeof(char)` donne également la valeur 1.

L'opérateur d'adressage

L'opérateur d'adressage `&` donne l'adresse d'une variable ou d'une expression.

&	opérateur d'adressage	&x	14	droite-gauche

© pk-mt/enseirb/2005

L'opérateur de conversion de type (cast)

Cet opérateur permet de convertir explicitement le type d'une donnée en un autre type.
Il est à noter que le compilateur C réalise des conversions implicites.

(type)	opérateur de conversion	(long int)c	14	droite-gauche

Exemple :
```
char i ;
int j ;
j = (int) i ;
```


© pk-mt/enseirb/2005

L'opérateur de parenthèse

L'opérateur de parenthèse `()` permet de définir l'ordre d'évaluation d'une expression. C'est également ce même opérateur qui est utilisé pour encapsuler les paramètres des fonctions.

Même lorsqu'une fonction n'a pas d'arguments, ces parenthèses sont obligatoires :

`()`	parenthèse	`()`	15	gauche-droite

© pk-mt/enseirb/2005

L'opérateur de sélection

Ces opérateurs `.` et `-` servent à sélectionner des champs de données structurées (cf. après).

`.`	opérateur de sélection	`x.info`	15	gauche-droite
`->`	opérateur de sélection	`x->info`	15	gauche-droite
`[]`	opérateur de sélection	`x[3]`	15	gauche-droite

© pk-mt/enseirb/2005

6.2. Conversion de types

Le problème de la conversion des types se pose lorsqu'une expression est composée de données de nature différentes. Par exemple, quel sens donner à une addition d'un entier et d'un nombre flottant ?
Le langage de C définit précisément quels sont les types de données compatibles et quel type de conversion est effectuée.

© pk-mt/enseirb/2005

❖ Les conversions implicites

Les conversion implicites sont celles faites automatiquement par le compilateur C lors de l'évaluation d'une expression (et donc également d'une affectation).

Cas de l'évaluation d'une expression :
Lorsqu'il est nécessaire de procéder à des conversions, le compilateur C effectue tout d'abord les conversions puis évalue l'expression. Les règles de conversion suivantes s'appliquent un ordre de conversion :

- Si l'un des opérandes est de type `long double`, conversion de l'autre expression en `long double`; le résultat est un `long double`.
- Si l'un des opérandes est de type `double`, conversion de l'autre expression en `double`; le résultat est un `double`.
- …

© pk-mt/enseirb/2005

Cas de l'affectation :

Lors d'une affectation, le type de l'expression membre droit est converti (si nécessaire) en le type de l'expression en membre gauche. Par exemple, les `char` sont convertis en `int`.

Cas où l'on affecte une expression plus ``longue'' à une expression plus ``courte'' ?

```
int i;
char j;

j = i;
```

La valeur affectée à la variable *j* sera tronquée des bits de poids faible.
La conversion d'un `float` en un `int` tronque la partie fractionnaire.

Dans tous les cas, il vaut mieux avoir une conversion explicite sinon se référer à la documentation du compilateur C !

© pk-mt/enseirb/2005

❖ Les conversions explicites

Le programmeur peut effectuer des conversions grâce à l'opérateur de conversion de type (*cast*).

Exemple :

L'expression `(int) 1.225` convertit le réel de type double en un entier. il en découle évidemment une perte de précision.

La conversion de `(int) 1.225` donne comme résultat la valeur entière `1`.

© pk-mt/enseirb/2005

6.3. Récapitulatif

Opérateur	Nom	Priorité	Ordre
[]	Elément de tableau	15	gauche-droite
()	Parenthèse	15	gauche-droite
.	Sélection	15	gauche-droite
-	Sélection	15	gauche-droite
&	Adressage	15	gauche-droite
sizeof	Dimension	14	droite-gauche
(type)	Conversion	14	droite-gauche
! (unaire)	NON	14	droite-gauche
~ (unaire)	NON bit à bit	14	droite-gauche
++	Incrément	14	droite-gauche
--	Décrément	14	droite-gauche
*	Multiplication	13	gauche-droite

© pk-mt/enseirb/2005

Opérateur	Nom	Priorité	Ordre
/	Division	13	gauche-droite
%	Modulo	13	gauche-droite
+	Addition	12	gauche-droite
-	Soustraction	12	gauche-droite
	Décalage à droite	11	droite-gauche
<<	Décalage à gauche	11	droite-gauche
<=	Test d'inférieur ou égal	10	gauche-droite
=	Test supérieur ou égal	10	gauche-droite
<	Test inférieur strict	10	gauche-droite
>	Test supérieur strict	10	gauche-droite
==	Test d'égalité	9	gauche-droite
!=	Test de non-égalité	9	gauche-droite
&	ET bit à bit	8	gauche-droite

© pk-mt/enseirb/2005

<table>
<tr><td>^</td><td>OU exclusif bit à bit</td><td>7</td><td>gauche-droite</td></tr>
<tr><td>|</td><td>OU bit à bit</td><td>6</td><td>gauche-droite</td></tr>
<tr><td>&&</td><td>ET</td><td>5</td><td>gauche-droite</td></tr>
<tr><td>||</td><td>OU</td><td>4</td><td>gauche-droite</td></tr>
<tr><td>?:</td><td>Opérateur conditionnel</td><td>3</td><td>droite-gauche</td></tr>
<tr><td>=</td><td>Affectation</td><td>2</td><td>droite-gauche</td></tr>
<tr><td>+=</td><td>Affectation</td><td>2</td><td>droite-gauche</td></tr>
<tr><td>-=</td><td>Affectation</td><td>2</td><td>droite-gauche</td></tr>
<tr><td>. *=</td><td>Affectation</td><td>2</td><td>droite-gauche</td></tr>
<tr><td>/=</td><td>Affectation</td><td>2</td><td>droite-gauche</td></tr>
<tr><td>%=</td><td>Affectation</td><td>2</td><td>droite-gauche</td></tr>
<tr><td>=</td><td>Affectation</td><td>2</td><td>droite-gauche</td></tr>
<tr><td><<=</td><td>Affectation</td><td>2</td><td>droite-gauche</td></tr>
<tr><td>&=</td><td>Affectation</td><td>2</td><td>droite-gauche</td></tr>
</table>

© pk-mt/enseirb/2005

<table>
<tr><td>|=</td><td>Affectation</td><td>2</td><td>droite-gauche</td></tr>
<tr><td>^=</td><td>Affectation</td><td>2</td><td>droite-gauche</td></tr>
<tr><td>,</td><td>Séquentiel</td><td>1</td><td>droite-gauche</td></tr>
</table>

© pk-mt/enseirb/2005

7. STRUCTURES DE CONTROLE DE PROGRAMME

Un programme C est constitué de :

- définitions et déclarations de variables
- de fonctions. Les fonctions sont construites à l'aide d'*instructions* combinées entre elles avec des *structures de contrôle*.

7.1. L'instruction if

Cette instruction conditionnelle permet d'exécuter des instructions de manière sélective en fonction du résultat d'un test. La syntaxe de l'instruction est :

```
if (expression)
      instruction1
else
      instruction2
```


© pk-mt/enseirb/2005

Si l'*expression* est vraie (valeur non nulle), l'*instruction1* s'exécute sinon, dans le deuxième cas, c'est l'*instruction2* qui s'exécute.

Rappelons que si la valeur d'une expression est différente de 0, la valeur booléenne de l'expression est vraie (**C n'est pas un langage fortement typé**). Il en découle que les deux écritures suivantes sont équivalentes :

```
if (expression)
   ...
```

et

```
if (expression != 0)
   ...
```

D'après la syntaxe de l'instruction `if`, la partie `else` est facultative.

© pk-mt/enseirb/2005

Il en découle une ambiguïté lorsqu'il y a des instructions `if` imbriquées.

Dans l'exemple suivant, comme le suggère l'indentation du programme :

```
if (expression1)
    if (expression2)
        instruction1
    else
        instruction2
```

l'*instruction2* correspond au `else` de l'*expression2*.

L'écriture suivante est strictement identique à la première.

© pk-mt/enseirb/2005


```
if (expression1)
    if (expression2)
        instruction1
else
    instruction2
```

Visiblement, le programmeur veut faire correspondre l'*instruction2* au `else` de l'*expression1*. Il suffit de transformer l'instruction imbriquée en un bloc d' instruction(s) de la manière suivante :

```
if (expression1) {
    if (expression2)
        instruction1
}
else
    instruction2
```


© pk-mt/enseirb/2005

Voici une construction, que l'on rencontre fréquemment :

```
if (expression1)
    instruction1
else if (expression2)
    instruction2
else if (expression3)
    instruction3
else
    instruction4
```

Cette instruction effectue successivement chacun des tests et aucun des tests n'est vraie, l'*instruction4* sera exécutée (instruction par défaut).

© pk-mt/enseirb/2005

7.2. Les instructions while, do et for

Les instructions itératives ou boucles sont réalisées à l'aide d'une des trois structures de contrôle suivantes :

```
while (expression)
    instruction

do
 instruction
while (expression);

for (expression1; expression2; expression3)
 instruction
```


© pk-mt/enseirb/2005

Pour l'embarqué, on n'a généralement pas de système d'exploitation mis en œuvre. On a donc un programme C qui s'exécute indéfiniment. Cela correspond à une boucle infinie en concurrence éventuellement avec les interruptions du système (on parle de superboucle). Il faut donc savoir écrire une boucle infinie en langage C :

```
for (; ;) {
}
```

ou

```
while (1) {
{
```


© pk-mt/enseirb/2005

7.3. L' instruction goto

L'instruction `goto` est généralement bannie quand on programme en langage C standard. **Mais elle a son intérêt dans le cas de l'embarqué pour accélérer un traitement…**
Une *étiquette* est un identificateur; elle sera suivie par un deux points (`:`) et sera placée devant une instruction.

L'instruction

```
goto étiquette;
```

transfère le contrôle à l'instruction préfixée par l'étiquette mentionnée.

© pk-mt/enseirb/2005

Une étiquette n'a pas besoin d'être déclarée (contrairement aux variables). De plus, elle est locale à une fonction *i.e.* qu'il ne peut y avoir un `goto` vers une étiquette figurant dans une autre fonction que celle dans laquelle se trouve l'instruction `goto`.

Dans la quasi totalité des programmes, l'instruction `goto` n'est pas utile. On la retrouve néanmoins dans l'embarqué dans les routines d'interruption pour optimiser les temps d'exécution…

© pk-mt/enseirb/2005

8. STRUCTURES DE PROGRAMMES ET FONCTIONS

Dans ce qui suit, on ne s'intéresse qu'au C ANSI.

Un programme C est constitué d'un ensemble de fonctions toutes accessibles à partir du programme principal `main()`.

`main` est un mot clé réservé du langage C. C'est le point d'entrée du programme sous UNIX (et Windows).
Dans le cas d'un système embarqué, il n'y a pas de système d'exploitation sous-jacent. `main` redevient banalisé et se comporte comme une simple fonction C. Le point d'entrée du programme sera précisé alors dans la phase d'édition liens par initialisation de vecteur d'interruption RESET : adresse du point d'entrée de la « fonction » main ou le point d'entrée d'une autre fonction…

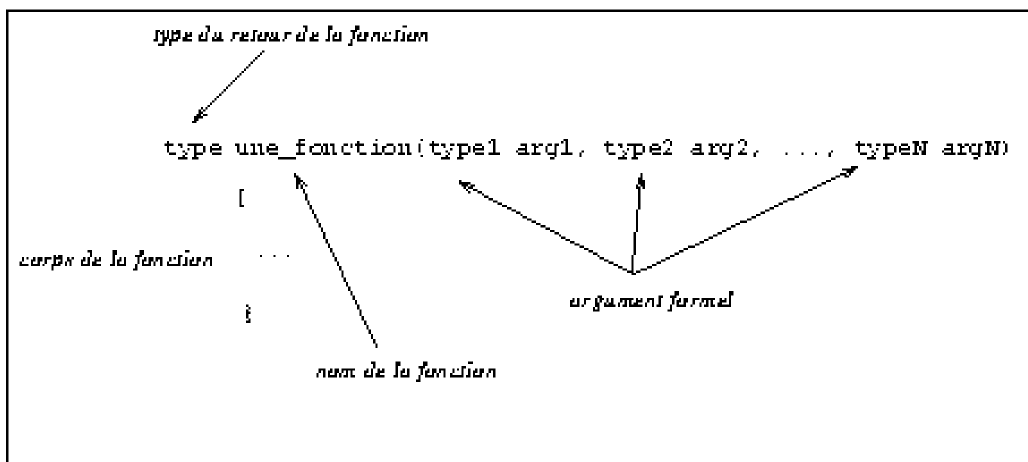
© pk-mt/enseirb/2005

8.1. Introduction

Une fonction est définie par :

- son nom
- ses arguments formels
- le type de la fonction (type de l'argument retourné par la fonction)
- son corps

© pk-mt/enseirb/2005



Exemple :

```
unsigned long pgcd(unsigned long x, unsigned long y) {
    unsigned long r;
    if (x < y) { r = x; x = y; y = x; }
    do {
        r = x % y;
        x = y;
```


© pk-mt/enseirb/2005

```
        y = r;
      }
      while (r != 0)
          ;
  }
      return x;                    /* retourne le pgcd calculé */
  }
```

Voici une utilisation de la fonction `pgcd` définie plus haut :

```
main() {
     unsigned long a, b;
     printf("Donnez deux entiers\n");
     scanf("%u %u", &a, &b);
     printf("Le pgcd de %u et de %u est %u\n", a, b, pgcd(a, b));
     printf("Le pgcd de 5 et de %u est %u\n", b, pgcd(5, b));
}
```


© pk-mt/enseirb/2005

Remarque :

La déclaration d'une fonction écrite en C-ANSI est :

```
unsigned long pgcd(unsigned long x, unsigned long y) {
    …
}
```

alors qu'en C K&R, elle est :

```
unsigned long pgcd(x, y) {
    unsigned long x ;
    unsigned long y ;
    …
}
```

Par défaut, une fonction retourne un entier `int`.

© pk-mt/enseirb/2005

❖ Déclaration de fonction

On appelle déclaration d'une fonction son nom, ses arguments formels et le type (type de la valeur retournée) de la fonction.

❖ Arguments formels

Les variables `x` et `y` sont les paramètres ou arguments formels de la fonction *pgcd*. Ce sont bien des arguments formels en ce sens que cette fonction s'applique quelles que soient les valeurs de `x` et `y`. Ces valeurs ne seront en fait connues que lors de l'appel de cette fonction.

❖ Arguments effectifs

Les variables `a`, `b` ainsi que la constante 5 sont les paramètres ou arguments effectifs de la fonction *pgcd* pour cet appel.

© pk-mt/enseirb/2005

❖ Type de la fonction

Cette fonction retourne une valeur de type `unsigned long` qui correspond au pgcd calculé par cette fonction.
La variable `r` est une variable locale à la fonction *pgcd*.

8.2. Appels de fonction et gestion de la pile

Une *pile* (tout comme une pile de linge ou feuille de brouillon) est une structure de données ayant les caractéristiques suivantes:

- on dépose le dernier élément en haut (on dira *sommet*) de la pile et le premier élément que l'on utilisera sera le dernier déposé. Une structure de donnée qui possède le comportement que l'on vient de décrire se nomme *liste LIFO* ou *pile* (Last In First Out).

© pk-mt/enseirb/2005

Les arguments formels et les variables locales d'une fonction sont stockés dans une zone particulière de la mémoire que l'on appelle *segment de pile* ou *pile* ou *stack* (en anglais). Un registre du processeur pointe sur le haut de cette zone (pointeur de pile).

❖ Passage par valeur

Lors de l'appel d'une fonction C, le système alloue une portion de la pile dédiée au programme. Cette zone mémoire est utilisée pour :

- stocker les arguments
- les variables locales
- l'adresse de retour de la fonction.

A chaque appel de fonction, les arguments effectifs sont recopiés à l'emplacement prévu.

© pk-mt/enseirb/2005


```
main() {
    unsigned long a=10, b=15, c;
    c = pgcd(a, b) ;
     ...
}
```

Etat de la pile juste avant l'appel de la fonction pgcd :

	10	5	0	
	a	b	c	

Etat de la pile après appel de la fonction pgcd :
Restitution de l'espace des arguments effectifs et des variables locales. Au retour de cette fonction, les valeurs de *a* et *b* sont inchangées.

	10	15	5	
	a	b	c	

© pk-mt/enseirb/2005

Cette manière de passer les arguments (par recopie des valeurs des arguments effectifs) se nomme **passage d'arguments par *valeur*.**

❖ Passage par adresse

Il est parfois utile de conserver les valeurs calculées par les paramètres formels.

Exemple : fonction qui calcule deux résultats que l'on voudrait récupérer.

☺ passage par adresse

Pour avoir un passage par adresse, on se contente de passer par valeur l'adresse d'une variable.

La fonction ne travaille plus sur une copie de la valeur de l'argument effectif mais directement sur celui-ci.

© pk-mt/enseirb/2005

Exemple :

Modification de la fonction `pgcd` de telle sorte que l'appel de la cette fonction modifie la valeur du premier paramètre effectif.

```
 void pgcd(unsigned long *x, unsigned long y) {
   unsigned long r;

   if (*x < y) {
     r = *x;
     *x = y;
     y = *x;
   }
   do {
       r = *x % y;
       *x = y;
       y = r;
   } while (r != 0);
}
```


© pk-mt/enseirb/2005

L'appel de cette fonction doit se faire de la manière suivante :

```
main(){
    unsigned long a=10, b=15;
    pgcd(&a, b);
    ...
}
```

Etat de la pile juste avant l'appel de la fonction pgcd :

	10	5	0	
	a	b	c	

© pk-mt/enseirb/2005

Etat de la pile après appel de la fonction pgcd :

Restitution de l'espace des arguments effectifs et des variables locales. Au retour de cette fonction, la valeur de *b* est inchangée ; par contre *a* contient à présent la valeur calculée par la fonction.

	5	15	5	
	a	b	c	

A comparer au passage de paramètres par valeurs :

	10	15	5	
	a	b	c	

© pk-mt/enseirb/2005

8.3. Divers

❖ Fonction sans argument et/ou qui ne retourne rien

Le mot clé `void` permet de spécifier que la fonction n'a aucun argument :

```
int fonction(void) {
     ...
}
```

L'appel à une fonction sans argument se fait de la manière suivante :

```
{
 ...
 i = fonction();   Ne pas oublier les parenthèses !!!
 ...
}
```


© pk-mt/enseirb/2005

De même, une fonction qui ne retourne rien (procédure) se note :

```
void fonction(...)
```

❖ Fonction récursive

En C, les fonctions peuvent être utilisées de manière récursive :
une fonction peut s'appeler elle-même.

Lorsqu'une fonction s'appelle récursivement, chaque niveau d'appel possède son propre jeu de variables locales.

Il est à noter que la récursivité peut entraîner des débordements de piles s'il y a trop d'appels imbriqués !
Dans l'embarqué, la récursivité est à éviter car on peut avoir une pile limitée ou pas du tout (microcontrôleur PIC avec une pile à 8 niveaux d'imbrication).

© pk-mt/enseirb/2005

9. VISIBILITE DES DONNEES

Nous avons jusqu'à présent associé aux variables un type particulier. Cet attribut n'est pas le seul que l'on assigne à une variable : il y a également la *classification* qui est un autre attribut important. Cette classification détermine l'emplacement mémoire où la variable sera stockée (voir le langage Java).

La classification concerne aussi les fonctions.

9.1. Variables globales et locales

Selon l'emplacement de la définition d'une variable, celle-ci est soit :
- une variable *locale.*
- une variable *globale*.

© pk-mt/enseirb/2005

Toute variable définie à l'extérieur des fonctions est une variable globale.

Toute variable définie à l'intérieur des fonctions est une variable locale.

Une variable globale est connue donc utilisable dans n'importe quelle partie du fichier où elle est définie en particulier à l'intérieur de n'importe quelle fonction du module.

Cette portée peut même s'étendre à d'autres modules du programme, si le programme est constitué de plusieurs fichiers.

Une variable locale est définie à l'intérieur d'une fonction et au début d'un bloc (c'est le seul endroit où il est permis de définir des variables locales). Une variable locale n'est connue qu'à l'intérieur du bloc dans lequel elle est définie.

© pk-mt/enseirb/2005

Exemple :

```
int g;                    g est une variable globale

main() {
    int i; i est une variable locale
    static int k; i est une variable globale privée
    . . .
}
```

On retiendra que :

- Les variables globales sont `statiques` c'est-à-dire qu'elles sont permanentes. Elles existent toute la durée de l'exécution du programme. Au démarrage du programme, le système alloue l'espace nécessaire et effectue les initialisations. Tout cet espace sera rendu au système lors de la terminaison du programme.
- les variables locales et les arguments formels des fonctions sont `automatiques`. L'espace nécessaire est alloué lors de l'activation de la fonction ou du bloc correspondant. Il est restitué au système à la fin du bloc ou de la fonction.

© pk-mt/enseirb/2005

9.2. Variables globales privées et publiques

Dans cette section, on s'intéresse aux programmes constitués de plusieurs modules.

Toutes les variables globales sont **publiques** c'est-à-dire qu'elles peuvent (moyennant une déclaration spécifique) être utilisées dans les autres modules (fichiers C).

On peut interdire cette facilité en donnant à ces variables le qualificateur `static`. On dit alors qu'il s'agit d'une variable globale **privée**.

Variables externes :

Le qualificateur `extern` placé devant la déclaration d'une variable ou d'une fonction informe que celui-ci sera défini plus loin dans le même module ou dans un autre module.

© pk-mt/enseirb/2005

A chaque variable, il ne peut y avoir qu'une seule définition. Par contre, il peut y avoir plusieurs déclarations dans le même module ou dans d'autres modules.

9.3. Fonctions privées et publiques

Comme pour les variables globales, les fonctions peuvent être privées ou publiques.

Une fonction privée ne peut être appelée que par des fonctions qui figurent dans le même module.
Par contre, une fonction publique peut être appelée par n'importe quelle fonction d'un module du programme.

Toutes les fonctions sont **publiques** c'est-à-dire qu'elles peuvent (moyennant une déclaration spécifique, voir plus loin) être utilisées dans les autres modules.

© pk-mt/enseirb/2005

On peut interdire cette facilité en donnant à ces fonctions l'état `static`. On dit alors qu'il s'agit d'une fonction **privée**.

Fonctions externes :

Le qualificateur `extern` placé devant la déclaration d'une fonction informe que celle-ci sera définie plus loin dans le même module ou dans un autre module.

A chaque fonction, il ne peut y avoir qu'une seule définition. Par contre, il peut y avoir plusieurs déclarations dans le même module ou dans d'autres modules.

© pk-mt/enseirb/2005

10. POINTEURS

10.1. Introduction

Un pointeur est une variable qui contient l'adresse d'une autre variable. L'opérateur unaire `&` donne l'adresse mémoire d'un objet.

L'instruction :

```
p = &c;
```

affecte l'adresse de `c` à la variable `p`.

On dit que `p` *pointe sur* `c`. L'opérateur unaire `&` s'applique uniquement sur des objets en mémoire.

© pk-mt/enseirb/2005

En particulier, il ne peut pas s'appliquer sur des constantes ou même à des variables déclarées comme `register` (registres du processeur).

L'opérateur unaire `*` représente l'opérateur d'*indirection* ou de *déréférence*.
Appliqué à un pointeur, il donne accès à l'objet pointé par ce pointeur.

```
int x = 1, y = 2, z[10];
int *pi;        pi pointeur sur un int
pi = &x;        pi pointe sur x
y = *pi;        y est affecté à la valeur de l'objet pointé par pi i.e. x
*pi = 0;        x vaut 0
pi = &z[0];     pi pointe sur le premier élément du tableau z
```


© pk-mt/enseirb/2005

10.2. Pointeurs et tableaux

Les pointeurs et les tableaux sont conceptuellement similaires en C.

Soient `tab` un tableau de `int` de 10 éléments et `pint` un pointeur vers un `int`. L'affectation :

```
pint = &tab[0]; ou pint = tab;
```

fait en sorte que `tab` et `pint` pointe tous deux sur le même élément (le premier élément du tableau).
Si `pint` pointe sur le $i^{ème}$ élément du tableau, alors `*(pint + 1)` désigne respectivement sur le $(i+1)^{ème}$ élément du tableau.

© pk-mt/enseirb/2005

La correspondance entre `pint` et `tab` est si étroite que l'on pourra écrire `tab[i]` ou `*(pint +i)`.

De même, `&tab[i]` ou `pint+i` sont équivalents.

10.3. Le type void

Le type `void` se comporte comme un type fondamental.

- Il sert à préciser qu'une fonction ne renvoie aucune valeur (procédure).
- Il peut être utilisé comme type de base pour des pointeurs sur des objets de type inconnu (pointeur universel ☺).

```
void f();
void * pv;
```


© pk-mt/enseirb/2005

Cette facilité doit être utilisée pour se servir des fonctions génériques qui rendent des pointeurs vers n'importe quel type d'objets (exemple : fonction `malloc`).

Exemple :

```
void * malloc(unsigned taille);
void free(void *);

void une_fonction() {
  int *pi;
  char *pc;
  pi = (int *) malloc(10*sizeof(int)); Noter la conversion explicite
  pc = (char *) malloc(10*sizeof(char));
  ...
  free(pi);
  free(pc);
}
```


© pk-mt/enseirb/2005

11. LES ENTREES SORTIES HAUT NIVEAU FORMATEES

11.1. Les E/S formatés. Les E/S bas niveau

Les E/S sont un point crucial dans le langage C. N'étant pas intégrés au langage comme avec le Pascal (avec *writeln* et *readln*), les appels d'E/S ont longtemps dépendu du type de compilateur C ; ce qui a conduit à des problèmes de portabilité des programmes C. L'arrivée de l'extension C-ANSI a permis de corriger ce défaut (du moins théoriquement…).

Un fichier est constitué d'une suite de données non structuré (*stream*) et est vu par un programme C comme un flot/flux i.e. une suite d'octets structurés ou non qui, selon le cas, constitue soit un *fichier texte* ou un *fichier binaire* ou encore **les données provenant *d'un périphérique matériel.***

On peut alors comparer un fichier C à un tableau d'octets.

© pk-mt/enseirb/2005

ENSEIRB
www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

- Un fichier texte est un fichier structuré en suite de lignes et exploitable par divers programmes (éditeur de texte, logiciel d'impression ...)
- Un fichier binaire n'a aucune structure et est inexploitable par des programmes externes qui ignorent la structure réelle du fichier.

Il est à noter que sous UNIX : *« Tout est fichier (et tout est processus) ».*

Un périphérique est repéré par un point d'entrée dans le système de fichiers (*path*) si bien que :

Lire et écrire sur un périphérique sous UNIX revient à lire et à écrire dans un fichier ☺
Il en est aussi de même pour la programmation socket en mode TCP !

© pk-mt/enseirb/2005

Les périphériques sont représentés traditionnellement par une entrée dans le répertoire `/dev` sous UNIX :
Exemple :

Liaison série : `/dev/ttya` sous Solaris `/dev/ttyS0` sous Linux (port COM1 du PC).

11.2. Le tampon d'entrée sortie des appels d'E/S formatés

Quelle que soit l'opération que l'on veuille effectuer sur un fichier (lecture ou écriture) d'un disque dur, les données transitent par un buffer de la bibliothèque d'E/S (libc) et ensuite par une zone mémoire temporaire appelé *tampon* (*buffer cache* du noyau sous UNIX) avant d'être définitivement rangées dans le fichier.

Le programmeur n'a pas à se soucier de la gestion de ce tampon : c'est le système d'exploitation qui en a la responsabilité.

© pk-mt/enseirb/2005

L'utilisation par l'intermédiaire d'un tampon permet d'optimiser le fonctionnement des E/S et **notamment sur disque** puisque l'écriture ou la lecture dans un tampon de taille N octets (qui réside en mémoire) est bien plus rapide que la même opération sur le fichier directement sous la forme de N lectures/écritures d'un octet.
On minimise ainsi les mouvements de la tête d'écriture d'un disque dur…

L'inconvénient est que l'on ne contrôle plus le moment de lecture/écriture dans le fichier qui est à la charge du système d'exploitation, ce qui est gênant pour l'accès à des périphériques matériels comme une liaison série.

© pk-mt/enseirb/2005

On utilisera alors les appels d'E/S bas niveau non bufferisés pour l'accès à un périphérique matériel (pas de buffer libc et éventuellement pas de tampon noyau)…
On utilisera alors les appels d'E/S de haut niveau formatées et bufferisés (buffer libc et tampon du noyau) pour l'accès à un fichier sur disque (bien que les appels d'E/S bas niveau non bufferisés (pas de buffer libc et tampon noyau) soient aussi utilisables…)

Le passage par un tampon, s'il accélère les opérations d'E/S introduit un problème de décalage dans la chronologie, puisque l'opération effective de lecture ou d'écriture dans le fichier n'est pas prédictible par le programmeur.
Il en découle qu'une interruption brutale d'un programme (ou du système) peut conduire à des pertes d'informations dans les fichiers.

© pk-mt/enseirb/2005

11.3. Fonctions générales sur les flots

Les flots sont désignés par des variables de type `FILE *` (pointeur sur une structure de type `FILE`).

Le type `FILE` est défini dans le fichier d'interface `stdio.h`.

Pour utiliser un fichier dans un programme C, il faut donc inclure le fichier `stdio.h` et définir une variable par fichier :

```
#include <stdio.h>
FILE *fd1, *fd2;
...
```


© pk-mt/enseirb/2005

❖ fopen

Avant qu'un programme puisse utiliser un fichier, il faut ouvrir ce fichier. L'ouverture du fichier initialise tous les champs du descripteur évoqué plus haut. C'est la fonction :

```
FILE *fopen(char *nom, char *mode_d_accès);
```

qui est utilisée pour ouvrir un fichier.
Cette fonction retourne un pointeur vers une structure de type `FILE` qui sera utilisée pour les opérations d'E/S par la suite.

- Le paramètre *nom* est une chaîne de caractères qui désigne le nom du fichier.
- Le paramètre *mode_d_accès* est une également une chaîne de caractères qui spécifie le type d'opération que l'on veut effectuer sur ce fichier :

© pk-mt/enseirb/2005

"r"	Ouvrir le fichier en lecture seule
"w"	Ouvrir le fichier en écriture seule
	(Création du fichier si le fichier n'existe pas)
	(Si le fichier existe, son ancien contenu sera perdu)
"a"	Ouvrir le fichier en ajout i.e. écriture à la fin du fichier
	(Création du fichier si le fichier n'existe pas)
"r+"	Ouvrir le fichier en lecture et écriture
"w+"	Ouvrir le fichier en lecture et écriture
	(Création du fichier si le fichier n'existe pas)
	(Si le fichier existe, son ancien contenu sera perdu)
"a+"	Ouvrir le fichier en lecture et en ajout
	(Création du fichier si le fichier n'existe pas)

Lorsqu'il y a une erreur, le programme ne s'arrête pas (pour arrêter le programme, il faut utiliser l'appel *exit(int cause)*).

© pk-mt/enseirb/2005

La fonction/appel renvoie la valeur `NULL` (qui vaut 0). C'est au programmeur de tester cette valeur chaque fois qu'il appelle cette fonction.

Bien programmer, c'est toujours tester le code de retour de tous les appels système ! ! !

```
#include <stdio.h>

main() {
    FILE *fd;
    fd = fopen("toto.c", "r");
    if (fd == NULL) {
        printf("Erreur d'ouverture du fichier %s", "toto.c");
        exit(0);
    }
   . . .
}
```


© pk-mt/enseirb/2005

ou plus concis à la sauce UNIX :

```
  #include <stdio.h>

  main() {
      FILE *fd;

      if ((fd = fopen("toto.c", "r")) == NULL) {
          printf("Erreur d'ouverture du fichier %s", "toto.c");
          exit(0);
      }
     . . .
}
```


© pk-mt/enseirb/2005

❖ fclose

Même si le système se charge de fermer tous les fichiers ouverts à la fin de l'exécution d'un programme, il est vivement conseillé de fermer les fichiers dès qu'ils ne sont plus utilisés notamment pour réduire le nombre de descripteurs ouverts et se prémunir contre toute interruption brutale du programme.

C'est la fonction :

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

qui est utilisée pour fermer un fichier.

Le paramètre *flot* (*stream*) est un pointeur vers une structure de type `FILE` qui est la valeur retournée par un appel à la fonction `fopen`.
La fonction `fclose` retourne `EOF` (*End Of File*) si la fermeture du fichier s'est mal passée. Sinon, c'est la valeur 0 qui est retournée.

© pk-mt/enseirb/2005

❖ fflush

Cette fonction provoque l'écriture physique immédiate du tampon à la demande du programmeur et non lorsque le système le décide. Elle retourne `EOF` en cas d'erreur et 0 dans les autres cas.

```
int fflush( FILE *flot);
```

❖ feof

La fonction `feof` teste l'indicateur de fin de fichier du flot spécifié par le paramètre `flot` et retourne une valeur non nulle.

```
int feof(FILE * flot);
```


© pk-mt/enseirb/2005

11.4. Les flots standards d'E/S

Par défaut, lorsqu'un programme débute, il existe trois flots prédéfinis qui sont ouverts :

- `stdin`
- `stdout`
- `stderr`

Ces flots sont connectés sur les organes d'E/S naturels :

- `stdin` : l'unité d'entrée standard (le clavier) dont le descripteur bas niveau est 0.
- `stdout` : l'unité de sortie standard (l'écran) dont le descripteur bas niveau est 1.
- `stderr` : l'unité de sortie erreur (l'écran) dont le descripteur bas niveau est 2.

© pk-mt/enseirb/2005

11.5. Lecture et écriture en mode caractère

❖ fputc

La fonction `fputc` transfère le caractère `car` dans le fichier représenté par le paramètre `flot`. La valeur de retour de cette fonction est le caractère `car` ou `EOF` en cas d'erreur :

```
int fputc(int car, FILE *flot);
```

❖ fgetc

La fonction `fgetc` retourne le caractère lu dans le fichier représenté par le paramètre `flot`. En cas d'erreur ou de fin de fichier, elle retourne la valeur `EOF`.

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


© pk-mt/enseirb/2005

❖ Les macros putc et getc

Les macros :

```
int putc(int car, FILE *flot);
int getc(FILE *flot);
```

font exactement la même chose que `fputc` et `fgetc` mais leur différence vient du fait qu'elles sont des macros (#define …) donc plus efficaces en temps d'exécution.

❖ getchar

`getchar()` est un synonyme de `getc(stdin)`.

```
int getchar(void);
```


© pk-mt/enseirb/2005

❖ putchar

`putchar(c)` est un synonyme de `putc(c, stdout)`.

```
int putchar(int c);
```

11.6. Lecture et écriture en mode chaîne de caractères

❖ fgets

La fonction `fgets` lit une chaîne de caractères dans un flot et le range dans un tampon défini (et alloué) par l'utilisateur.

```
char *fgets(char *s, int taille, FILE *flot);
```


© pk-mt/enseirb/2005

Plus précisément, cette fonction lit `taille-1` caractères dans le fichier de descripteur `flot` ou bien jusqu'au caractère `'\n'`, s'il survient avant les `taille-1` caractères. Le dernier caractère du tampon est `'\0'`.

La fonction rend le pointeur vers le début du tampon ou `NULL` en cas d'erreur ou de fin de flot.

❖ fputs

La fonction `fputs` écrit une chaîne de caractères dans un flot. Cette fonction n'écrit pas le caractère `'\0'` .

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

La fonction retourne une valeur non négative si tout s'est bien passé. La valeur `EOF` indique une erreur.

© pk-mt/enseirb/2005

❖ gets

Comme on peut le voir sur le prototype de la fonction, le flot ainsi que le nombre maximum de caractères à lire ne sont pas spécifiés.

```
char *gets(char *s);
```

Le flot par défaut est `stdin`.

❖ puts

La fonction `puts` écrit sur le flot `stdout` la chaîne de caractères `s`.

```
int puts(char *s);
```


© pk-mt/enseirb/2005

La fonction retourne une valeur non négative si tout s'est bien passé. La valeur `EOF` indique une erreur. Contrairement à la fonction `fputs`, la fonction ajoute le caractère '`\n`' à la fin de l'écriture.

11.7. Lecture et écriture formatée

Les fonctions `fprintf` et `fscanf` permettent d'effectuer des lectures et écritures formatées de données dans un flot. Les fonctions `printf` et `scanf` sont des raccourcis pour les fonctions `fprintf` et `fscanf` lorsque le flot est l'entrée standard ou la sortie standard.

```
int fprintf(FILE *flot, char *format, arg1, ...,  argN);
```


© pk-mt/enseirb/2005

❖ Ecriture formatée avec printf

Nous avons déjà rencontré ce type d'écriture dans les divers exemples précédents.
Le format général de la fonction `printf` est de la forme :

```
int printf(const char *format, arg1, ..., argN);
```

La fonction `printf` affiche les arguments donnés en fonction du format spécifié et ce après conversion et mise en forme si nécessaire.
Cette fonction retourne le nombre de caractères affichés.
Le format est une chaîne de caractères qui contient deux types d'objets :

- des caractères ordinaires, qui sont affichés sans aucune modification
- des spécifications qui déterminent le type de conversion et de mise en forme à effectuer avant l'affichage proprement dit.

© pk-mt/enseirb/2005

Toutes les spécifications commencent par le caractère `%` et se terminent par un caractère de conversion. Entre ces deux caractères, on peut placer dans l'ordre:

1. des drapeaux :

 \+ : impression du signe du nombre

espace **:** place un espace au début

 0 **:** complète (si besoin) le début du champ par des zéros (pour les arguments numériques)

2. un nombre qui précise la largeur minimum du champ d'impression.
3. un point qui sépare la largeur du champ et la précision demandée.
4. un nombre qui spécifie la précision du champ.
5. une lettre qui modifie la largeur du champ (h pour short, l pour long et L pour long double)

© pk-mt/enseirb/2005

d, i	notation décimale signée	int
u	notation décimale non signée	int
o	notation octale non signée	int
x, X	notation hexadécimale non signée sans les caractères 0x ou 0X	int
c	caractère après conversion en unsigned char	int
s	impression d'une chaîne de caractères se terminant pas par le caractère de fin de chaîne	char *
f	notation décimale de la forme *[-]mmm.ddd*	double
%	imprime le caractère %	

```
float x = 123.456;

printf("%f ", x);             123.456000
printf("%.2f ", x);           123.46
```


© pk-mt/enseirb/2005

❖ Ecriture formatée avec scanf

Le format général de la fonction `scanf` est de la forme :

```
int scanf(const char *format, arg1, ...,  argN);
```

Cette fonction lit dans le flot, en fonction des spécifications du format les données pour les affecter aux arguments fournis.
Cette fonction rend EOF en cas d'erreur ou si la fin de flot est atteint. Sinon, elle retourne le nombre d'objets lus.

Le format est le même que celui de `printf`.

© pk-mt/enseirb/2005

12. LES ENTREES SORTIES BAS NIVEAU

Jusqu'à présent, les fonctions d'E/S que avons évoquées (C ANSI) sont de haut niveau (bufferisées et formatées)
Elles sont indépendantes des systèmes d'exploitation que l'on utilise.

Nous allons nous intéresser à des fonctions de plus bas niveau et qui peuvent différer d'un système à l'autre.

Ce qui va suivre s'applique pour tous les systèmes d'exploitation UNIX.

Au démarrage d'un programme, trois fichiers standards sont ouverts : `stdin`, `stdout` et `stderr` dont les descripteurs sont respectivement 0, 1 et 2. On peut donc utiliser ces fichiers sans avoir à les ouvrir comme c'est le cas pour tout autre flot.

© pk-mt/enseirb/2005

❖ open et create

Les fonctions de bas niveau `open` et `creat` permettent d'ouvrir un fichier donc un **périphérique matériel** et éventuellement de le créer :

```
int open(const char *pathname, int flags, mode_t mode);
int creat(const char *pathname, mode_t mode);
```

Ces fonctions retournent un descripteur de fichier (*file descriptor*) qui est un **entier** (int).

Le paramètre `flags` a pour valeur `O_RDONLY`, `O_WRONLY` ou `O_RDWR` selon le l'on veuille utiliser le fichier en lecture seule, en écriture seule ou en lecture et écriture.

© pk-mt/enseirb/2005

Le paramètre `flags` peut également être un entier vu comme une suite de bits que l'on positionne avec des *ou* (`|`).
Les bits à positionner sont :

- `O_CREAT` : Créer le fichier s'il n'existe pas.
- `O_EXCL` : Ouverture du fichier pour un usage exclusif.
- `O_APPEND` : Ouverture du fichier en mode *append* .
- `O_NONBLOCK` ou `O_NDELAY` : Ouverture du fichier en mode non bloquant.

Le paramètre `mode` définit les permissions du fichier (en cas de création). Généralement la valeur de mode est 0.

Pour utiliser ces constantes, il faut inclure le fichier :

```
#include <stdlib.h>
```


© pk-mt/enseirb/2005

La fonction `creat` sert à créer un fichier s'il n'existe pas ou à réécrire sur d'anciens fichiers. Les permissions d'accès sont données en notation octale (Cf permissions d'accès sous UNIX).

❖ read et write

Les E/S de haut niveau sont toutes bâties par des appels systèmes (fonctions de bas niveau) `read` et `write`. Ces appels bas niveau sont non bufferisés (pas de buffer libc, on peut éventuellement passer par le tampon du noyau…) et le flux d'octets est non formaté :

```
int read(int fd, char *buf, int count);
int write(int fd, const char *buf, int count);
```

La fonction `read` lit dans le fichier désigné par le descripteur de fichier `fd`, `count` octets et les range dans le tableau de caractères `buf`.

© pk-mt/enseirb/2005

La fonction `write` écrit dans le fichier désigné par le descripteur de fichier `fd`, `count` octets du tableau de caractères `buf`.

Ces fonctions renvoient le nombre d'octets effectivement transférés. On peut donc demander à lire ou écrire *n* octets mais si le nombre d'octets disponibles est inférieur à *n*.
La valeur 0 est retournée lorsqu'on atteint une fin de fichier et -1 pour toute autre erreur.

© pk-mt/enseirb/2005

<u>Exemple :</u> Programme qui lit une suite de caractères frappés au clavier et qui les réécrit à la console.

```
int main() {
    char buf[128];
    int n;
    while (n = read(0, buf, sizeof(buf)))
        write(1, buf, n);
    return 0;
}
```


© pk-mt/enseirb/2005

❖ ioctl

La fonction `ioctl` permet de changer le comportement par défaut du périphérique ouvert ou du fichier ouvert.
La syntaxe est :

```
      #include <unistd.h>
      #include <stropts.h>

      int ioctl(int fildes, int request, /* arg */ ...);
```

`request` est la commande de reconfiguration et `arg` le ou les paramètres.

© pk-mt/enseirb/2005

<u>Exemple :</u> Etat courant d'une liaison série

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

extern int errno;

main(){
struct termio etat1;
int n;

if((n = ioctl(0, TCGETA, &etat1)) == -1)
        { fprintf(stderr, "echec de ioctl => erreur %d\n", errno);
        exit(1);
}
printf("ioctl reussi => valeur de retour : %d\n", n);
printf("comportement de la liaison : %d\n", etat1.c_line);
printf("valeurs du caractere special <intr> : %d\n", etat1.c_cc[0]);
}
```


© pk-mt/enseirb/2005

❖ close

La fonction `close` permet de libérer le descripteur de fichier et donc de fermer le périphérique matériel ou le fichier.
La syntaxe est :

```
int close(int fildes);
```

`close` renvoie -1 en cas d'échec ou 0 sinon.

© pk-mt/enseirb/2005

13. COMPARAISON ENTRE LES E/S BAS NIVEAU ET HAUT NIVEAU

13.1. Cas de Linux

On se placera dans le cas du système d'exploitation Linux : il faut généralement vérifier comment est mis en œuvre le tampon du noyau…

Les appels de la famille `read()` and Co sont sous Linux comme suit :
- travaillent sur un descripteur de fichier de type *int*.
- ne sont pas standards C ANSI (`_read()` sous Visual C…).
- sont conformes à la norme POSIX.
- sont des wrappers aux appels systèmes *syscalls*.
- sont des E/S non formatées : on a un bloc non formaté de 1 ou plusieurs octets.
- n'utilisent pas de buffer standard *libc* dans l'espace *user*.

© pk-mt/enseirb/2005

- peuvent utiliser les buffers Linux VFS et/ou le tampon du noyau (espace *kernel*). Cela dépend de quoi l'on parle : fichiers, liaison série, FIFO, socket, pty... Le tampon de l'espace *kernel* peut être dévalidé par `fcntl(..., F_SETFL, O_DIRECT)` ou par `open()` avec le flag `O_DIRECT`.
- sont utilisés généralement pour accéder à des données niveau bas : périphérique matériel ou système de fichiers en mode *raw*.

On a donc :
- un buffer utilisateur (paramètre de l'appel) que l'on peut utiliser.
- des buffers Linux VFS et/ou le tampon du noyau mis en œuvre éventuellement.

© pk-mt/enseirb/2005

Les appels de la famille `fread()` and Co sont sont sous Linux comme suit :
- sont un niveau additionnel d'abstraction pour les E/S fichier.
- travaillent sur un descripteur de fichier de type *FILE ** .
- sont standards C ANSI.
- sont des fonctions de la bibliothèque d'E/S standard *libc* (*glibc*).
- utilisent un buffer standard *libc* dans l'espace *user*. La bufferisation *user space* peut être contrôlée (`setvbuf()`) ou dévalidée.
- permettent des E/S formatées pour certains.
- peuvent utiliser les buffers Linux VFS buffers et/ou le tampon du noyau (espace *kernel*). Cela dépend de quoi l'on parle : fichiers, liaison série, FIFO, socket, pty... Le tampon de l'espace *kernel* peut être dévalidé par (`fileno()` avec `fcntl(..., F_SETFL, O_DIRECT)`).
- sont utilisés généralement pour accéder à des flux de données de façon indépendante du périphérique matériel utilisé.

© pk-mt/enseirb/2005

On a donc :
- un buffer utilisateur (paramètre de l'appel) que l'on peut utiliser.
- un buffer standard *libc*.
- des buffers Linux VFS et/ou le tampon du noyau mis en œuvre éventuellement.

13.2. Comparaisons par l'exemple

Les exemples suivantes présentent la mise en œuvre des E/S bas niveau et formatées haut niveau mettant en évidence les bienfaits/méfaits de la bufferisation (adapté de http://www.cs.utk.edu/~plank/plank/classes/cs360/360/notes/Cat/lecture.html) sous UNIX (Solaris 5.7 machine *fakir*).

© pk-mt/enseirb/2005

Fichier C `simpcat1.c` :

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

main()
{
  char c;


  c = getchar();
  while(c != EOF) {
    putchar(c);
    c = getchar();
  }
}
```


© pk-mt/enseirb/2005

Fichier C `simpcat2.c` :

```
main()
{
  char c;
  int i;

  i = read(0, &c, 1);
  while(i > 0) {
    write(1, &c, 1);
    i = read(0, &c, 1);
  }
}
```


© pk-mt/enseirb/2005

Fichier C `simpcat3.c` :

```
#include <stdio.h>

main()
{
  char c[1];
  int i;

  i = fread(c, 1, 1, stdin);
  while(i > 0) {
    fwrite(c, 1, 1, stdout);
    i = fread(c, 1, 1, stdin);
  }
}
```


© pk-mt/enseirb/2005

On crée un gros fichier à lire *large* de 2 Mo et on mesure le temps d'exécution de l'exécutable (-O2) par la commande UNIX `time` :

```
% dd if=/dev/zero of=large  bs=1M count=2
% time simpcat1 < large > /dev/null
real    0m0.375s
user    0m0.320s
sys     0m0.040s
% time simpcat2 < large > /dev/null
real    0m56.183s
user    0m12.530s
sys     0m40.800s
% time simpcat3 < large > /dev/null
real    0m2.959s
user    0m2.730s
sys     0m0.130s
```


© pk-mt/enseirb/2005

- Dans `simpcat1.c`, `getchar()` et `putchar()` sont appelés 2M fois pour 1 octet. Ce sont des macros optimisées des appels de la bibliothèque *libc*.
- Dans `simpcat2.c`, `read()` et `write()` sont appelés 2M fois pour 1 octet. Ils font appel à des appels systèmes plus lents (changement de contexte…).
- Dans `simpcat3.c`, `fread()` et `fwrite()` sont appelés 2M fois pour 1 octet. Ce sont des fonctions de la bibliothèque *libc* plus rapides.

On peut maintenant jouer sur le paramètre de la taille du buffer utilisateur (paramètre de l'appel) :

© pk-mt/enseirb/2005

Fichier C simpcat4.c :

```
#include <stdio.h>

main(int argc, char **argv)
{
  int bufsize;
  char *c;
  int i;

  bufsize = atoi(argv[1]);
  c = malloc(bufsize*sizeof(char));
  i = 1;
  while (i > 0) {
    i = read(0, c, bufsize);
    if (i > 0) write(1, c, i);
  }
}
```


© pk-mt/enseirb/2005

Fichier C simpcat5.c :

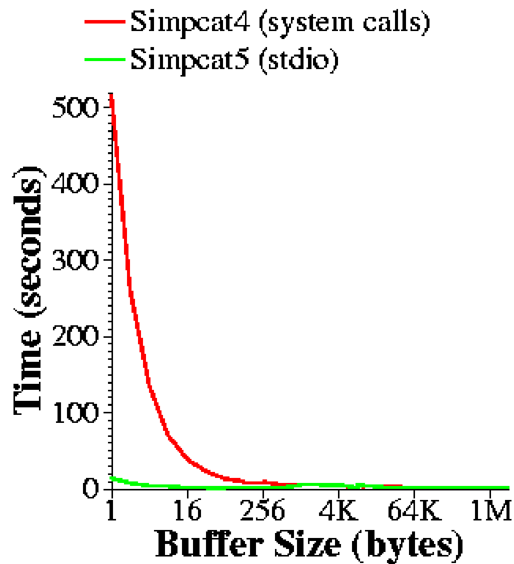
```
#include <stdio.h>

main(int argc, char **argv)
{
  int bufsize;
  char *c;
  int i;

  bufsize = atoi(argv[1]);
  c = (char *)malloc(bufsize*sizeof(char));
  i = 1;
  while (i > 0) {
    i = fread(c, 1, bufsize, stdin);
    if (i > 0) fwrite(c, 1, i, stdout);
  }
}
```


© pk-mt/enseirb/2005

On fait varier ainsi la valeur de `bufsize` et l'on obtient le diagramme similaire à :



On limite ainsi le nombre d'appels système ou d'appels de fonctions de la bibliothèque *libc* dès que l'on augmente la taille du buffer utilisateur.

© pk-mt/enseirb/2005

14. PROGRAMMATION C AVANCEE

Nous venons de voir le langage C dont sa principale faiblesse réside dans sa **non prise en compte au niveau du langage du traitement des E/S**. C'est d'ailleurs pour cela que l'on doit faire appel à des bibliothèques d'appels d'E/S…

Des éléments connexes au langage sont parfois nécessaires comme :

- Les arguments de la ligne de commande et les variables d'environnement
- Signaux
- Communications inter processus
- …

14.1. Arguments passés à un programme

Il est possible de passer des arguments à un programme lors de son exécution par l'intermédiaire de la ligne de commande qui le lance :

© pk-mt/enseirb/2005

```
% mon_exe arg1 arg2 arg3
```

Ici, 3 arguments sont passés au programme. **Ils sont vus comme des chaînes de caractères.**

Au niveau du fichier source C (ici mon_exe.c), la déclaration des arguments se fait comme suit :

```
main(int argc, char *argv[])
{
if(argc != 4) {
   prinft("Erreur d'arguments\n");
   exit(0);
}
…
}
```


© pk-mt/enseirb/2005

- La variable `argc` du type *int* contient le nombre d'arguments plus la commande elle-même (4 sur l'exemple).
- La variable `argv` du type *char* ** est un tableau de chaînes de caractères contenant les arguments. `argv[0]` contient la chaîne de caractères du nom de la commande elle-même (ici "mon_exe"), `argv[1]` contient la chaîne de caractères "arg1"…

Il est à noter que si les arguments sont considérés comme des entiers, une conversion chaîne de caractères / entier est à faire avant leur utilisation. On pourra utiliser l'appel `atoi()` par exemple…

14.2. Options de compilation

Dans le contexte de l'embarqué, il est important de bien connaître les options de compilation du compilateur C. Il y a en fait un compromis à trouver entre la taille de

© pk-mt/enseirb/2005

l'exécutable produit et son temps d'exécution. Un système embarqué est généralement peu performant (par rapport à un PC) et a des ressources mémoire limitées.

Les options de compilations sont liées au compilateur C utilisé. Le compilateur C présenté ici est le compilateur C du projet GNU (*Gnu is Not Unix*) *gcc*.

La compilation standard par gcc est :

```
% gcc -o toto toto.c
```

Le compilateur gcc possède 4 niveaux d'optimisation :

- O0 : niveau par défaut. Implicite.
- O1 : premier niveau. Optimisation du temps de compilation.
- O2 : deuxième niveau. Compromis temps d'exécution et taille du code généré.
- O3 : troisième niveau. Optimisation du temps d'exécution (au détriment de la taille du code généré).
- Os : cas spécial. Optimisation de la taille du code généré.

© pk-mt/enseirb/2005

La compilation O2 par gcc est :

```
% gcc -O2 -o toto toto.c
```

Il est possible d'optimiser le code produit en fonction du type du processeur utilisé (architecture) en mettant en œuvre des instructions assembleur spécifiques.

La compilation par gcc est :

```
% gcc -march=xxx toto toto.c
```

où xxx vaut (famille x86) :

- `i386` : processeur i386 DX/SX/CX/EX/SL
- `i486` : processeur i486 DX/SX/DX2/SL/SX2/DX4 ou 487.
- `pentium` : processeur Pentium.
- `pentium2` : processeur Pentium 2.
- `pentium4` : processeur Pentium 4.
- `c3` : processeur VIA C3.
- `k6` : processeur AMD K6

© pk-mt/enseirb/2005

- `athlon` : processeur Athlon.
- …

Il est possible de préciser quelle unité mathématique le processeur supporte.

La compilation par gcc est :

```
% gcc -mfpmath=xxx toto toto.c
```

où xxx vaut (famille x86) :

- `387` : FPU 387. Processeur i386…
- `sse` : FPU Streaming SIMD Extensions. Processeurs Pentium III, Athlon…
- `sse2` : FPU Streaming SIMD Extensions II. Processeur Pentium IV…

© pk-mt/enseirb/2005

15. LES BIBLIOTHEQUES STANDARDS

Ce paragraphe se propose de présenter les appels/fonctions de quelques bibliothèques généralement fournies avec tout bon compilateur C.

A tout moment, pour avoir de l'aide supplémentaire sous UNIX sur un appel, taper la commande :

```
% man mon_appel
```

Exemple :

```
% man fopen
% man -s 2 read
```


© pk-mt/enseirb/2005

15.1. Entrées sorties `<stdio.h>`

❖ Opérations sur les fichiers

fonction	description
`remove`	destruction de fichier
`rename`	modification de nom de fichier
`tmpfile`	création d'un fichier temporaire
`tmpnam`	génération de nom approprié pour un fichier temporaire

© pk-mt/enseirb/2005

❖ Accès aux fichiers

fonction	description
`fclose`	fermeture de fichier
`fflush`	écriture sur fichier des buffers en mémoire
`fopen`	ouverture de fichier
`freopen`	ouverture de fichier

© pk-mt/enseirb/2005

❖ E/S formatées et bufferisées

fonction	description
`fprintf`	écriture formatée sur flot de données
`fscanf`	lecture formatée sur flot de données
`printf`	écriture formatée sur sortie standard
`scanf`	lecture formatée sur entrée standard
`sprintf`	écriture formatée dans une chaîne de caractères
`sscanf`	lecture formatée depuis une chaîne de caractères
`vfprintf`	variante de `fprintf`
`vprintf`	variante de `printf`
`vsprintf`	variante de `sprintf`

© pk-mt/enseirb/2005

❖ E/S mode caractère

fonction	Description
`fgetc`	lecture d'un caractère
`fgets`	lecture d'une chaîne de caractères
`fputc`	écriture d'un caractère
`fputs`	écriture d'une chaîne de caractères
`getc`	`fgetc` implémenté par une macro
`getchar`	`getc` sur l'entrée standard
`gets`	lecture d'une chaîne de caractères sur l'entrée standard
`putc`	`fputc` implémenté par une macro
`putchar`	`putc` sur la sortie standard
`puts`	écriture d'une chaîne de caractères sur la sortie standard
`ungetc`	refoule un caractère (sera lu par la prochain lecture)

© pk-mt/enseirb/2005

❖ E/S mode binaire

Pour lire et écrire des données binaires, on dispose de deux fonctions : `fread` et `fwrite`.

❖ Position dans un fichier

fonction	description
`fgetpos`	donne la position courante dans un fichier
`fseek`	permet de se positionner dans un fichier
`fsetpos`	permet de se positionner dans un fichier
`ftell`	donne la position courante dans un fichier
`rewind`	permet de se positionner au début d'un fichier

© pk-mt/enseirb/2005

❖ Gestion des erreurs

fonction	description
`clearerr`	remet à faux les indicateurs d'erreur et de fin de fichier
`feof`	test de l'indicateur de fin de fichier
`feror`	test de l'indicateur d'erreur
`perror`	imprime un message d'erreur correspondant à `errno`

© pk-mt/enseirb/2005

15.2. Mathématiques `<math.h>`

Sous UNIX, on veillera de dire au compilateur C d'inclure la bibliothèque mathématique libm.a en tapant la commande :

```
% cc -o toto  toto.c -lm
```


© pk-mt/enseirb/2005

❖ Fonctions trigonométriques et hyperboliques

fonction	sémantique
`acos`	arc cosinus
`asin`	arc sinus
`atan`	arc tangente
`atan2`	arc tangente
`cos`	cosinus
`cosh`	cosinus hyperbolique
`sin`	sinus hyperbolique
`sinh`	sinus
`tan`	tangente
`tanh`	tangente hyperbolique

© pk-mt/enseirb/2005

❖ Fonctions exponentielles et logarithmiques

fonction	sémantique
`exp`	exponentielle
`frexp`	étant donné *x*, trouve *n* et *p* tels que $x = n * 2^p$
`Ldexp`	multiplie un nombre par une puissance entière de 2
`log`	logarithme
`log10`	logarithme décimal
`modf`	calcule partie entière et décimale d'un nombre

© pk-mt/enseirb/2005

❖ Fonctions diverses

fonction	sémantique
`ceil`	entier le plus proche par les valeurs supérieures
`fabs`	valeur absolue
`floor`	entier le plus proche par les valeurs inférieures
`fmod`	reste de division
`pow`	puissance
`sqrt`	racine carrée

© pk-mt/enseirb/2005

15.3. Manipulation de chaînes de caractères <string.h>

On dispose de fonctions pour :

- copier : `memcpy`, `memmove`, `strcpy`, `strncpy`
- concaténer : `strcat`, `strncat`
- comparer : `memcmp`, `strcmp`, `strcoll`, `strncmp`
- transformer : `strxfrm`
- rechercher : `memchr`, `strchr`, `strcspn`, `strpbrk`, `strrchr`, `strspn`, `strstr`, `strtok`
- initialiser : `memset`
- calculer une longueur : `strlen`
- obtenir un message d'erreur à partir du numéro de l'erreur : `strerror`.

© pk-mt/enseirb/2005

15.4. Manipulation de caractères <ctype.h>

Toutes les fonctions ci-dessous permettent de tester une propriété du caractère passé en paramètre :

fonction	le paramètre est :
`isalnum`	une lettre ou un chiffre
`isalpha`	une lettre
`iscntrl`	un caractère de commande
`isdigit`	un chiffre décimal
`isgraph`	un caractère imprimable ou le blanc
`islower`	une lettre minuscule
`isprint`	un caractère imprimable (pas le blanc)
`ispunct`	un caractère imprimable pas isalnum
`isspace`	un caractère d'espace blanc
`isupper`	une lettre majuscule
`isxdigit`	un chiffre hexadécimal

© pk-mt/enseirb/2005

On dispose également de deux fonctions de conversions entre majuscules et minuscules :

- `tolower` : conversion en minuscule
- `toupper` : conversion en majuscule

© pk-mt/enseirb/2005

15.5. Utilitaires divers `<stdlib.h>`

❖ Conversion de nombres

Les fonctions suivantes permettent de convertir des nombres entre la forme binaire et la forme chaînes de caractères.

fonction	description
`atof`	conversion de chaîne vers `double`
`atoi`	conversion de chaîne vers `int`
`atol`	conversion de chaîne vers `long int`
`strtod`	conversion de chaîne vers `double`
`strtol`	conversion de chaîne vers `long int`
`strtoul`	conversion de chaîne vers `unsigned long int`

© pk-mt/enseirb/2005

❖ Génération de nombres pseudo aléatoires

On dispose de `rand` et `srand`.

❖ gestion de la mémoire

La liste exhaustive est `calloc`, `free`, `malloc` et `realloc`.

© pk-mt/enseirb/2005

❖ Communication avec l'environnement

fonction	Description
`abort`	terminaison anormale du programme
`atexit`	installe une fonction qui sera exécutée sur terminaison normale du programme
`getenv`	obtention d'une variable d'environnement
`system`	exécution d'un programme

❖ Arithmétique sur les entiers

fonction	description
`abs`	valeur absolue
`div`	obtention de quotient et reste
`labs`	idem `abs` sur des `long int`

© pk-mt/enseirb/2005

16. EXEMPLES DE PROGRAMMES C POUR L'EMBARQUE

16.1. Exemple 1 : programmation mémoire EPROM

But :
Création d'un fichier de programmation d'une mémoire EPROM 8 bits avec les 32K échantillons d'une période de sinusoïde.

Système :
UNIX Solaris sur Sun (marche sur tout UNIX).

```
#include <stdio.h>
#include <fcntl.h>
#include <termio.h>
#include <math.h>
extern int errno;
```


© pk-mt/enseirb/2005


```
#define SAMPLE 32768
#define FREQ   1000.0
main()
{
int fd, n, i, total=0;
double w;
char s;

if((fd=creat("sinus.dat", 0600)) == -1) {
        perror("Creation fichier");
        exit(-1);
}

w = 2.0 * M_PI * FREQ;
printf("w=%f rad/s\n", w);

// close(1);
```


© pk-mt/enseirb/2005

```
// dup(fd);

for(i=0; i<SAMPLE; i++) {
   s = (char)(128.0 * (1.0 + cos(2 * M_PI * i / SAMPLE)));
   write(fd, &s, 1);
// printf("%c", s);
}

close(fd);

exit(0);
}
```

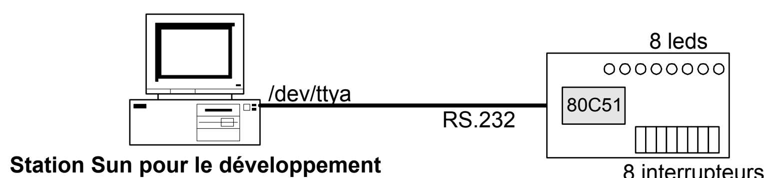
Remarques :
Utilisation des E/S bas niveau.
Les appels `close(1)` et `dup(fd)` permettent de fermer la sortie standard et de dupliquer le descripteur de fichier fd qui prend la plus petite valeur positive libre ;

© pk-mt/enseirb/2005

c'est-à-dire 1 ici. L'appel `printf` permet alors d'écrire directement dans le fichier d'échantillons.

16.2. Exemple 2 : pilotage d'un module par la liaison série

But :
Pilotage d'un module électronique relié au port série d'une station Sun.
Système :
UNIX Solaris sur Sun.



© pk-mt/enseirb/2005

```
#include <stdio.h>
#include <fcntl.h>
#include <termio.h>

extern int errno;

main()
{
int fd;
char commande[80]= "readbp\n";
char reponse[80]="";
int n, i, j;

if((fd=open("/dev/ttya", O_RDWR, 0)) == -1) {
   perror("Ouverture ttya");
   exit(-1);
}
```


© pk-mt/enseirb/2005


```
while(1) {
   printf("Commande No %d\n", ++j);
   write(fd, commande, sizeof(commande));

   bzero(reponse, sizeof(reponse));

   n = read(fd, reponse, sizeof(reponse));
   printf("\tn2=%d BP=%d\n", n, atoi(reponse));

   sleep(1);
}
close(fd);

exit(0);
}
```


© pk-mt/enseirb/2005

Remarques :

Utilisation des E/S bas niveau.

On lit toutes les 1 secondes l'état courant de boutons poussoirs du module électronique relié par la liaison série. La liaison série est déjà configurée par le système d'exploitation en 9600 b/s, 8 bits, sans parité, 1 stop bit.

© pk-mt/enseirb/2005

16.3. Exemple 3 : bibliothèque de contrôle d'une carte VME

But :

Pilotage d'une carte VME dans un châssis VME sous UNIX System V.

Système :

UNIX System V sur carte 68040.

```
struct cbuf cmic_d_ini(tampon_dem)
struct cbuf tampon_dem;
{
struct cbuf tampon_con;
int semid, id;
int fd;
int i;
int j;
struct params params;
int status;
```


© pk-mt/enseirb/2005

```
if((semid = semget(CMICSEMKEY, CMICNBRSEM, 0)) == -1) {
   tampon_con.code = CMIC_C_INI;
   tampon_con.par0 = B_NOK;
   tampon_con.par1 = E_DBA;
   return(tampon_con);
}

/* On prend le semaphore              */
if((id = Psem(semid)) == -1) {
   tampon_con.code = CMIC_C_INI;
   tampon_con.par0 = B_NOK;
   tampon_con.par1 = E_DBA;
   return(tampon_con);
}
```


© pk-mt/enseirb/2005


```
if((fd=open("/dev/cmic",O_RDWR)) == -1) {
   Vsem(semid);
   tampon_con.code = CMIC_C_INI;
   tampon_con.par0 = B_NOK;
   tampon_con.par1 = E_DRI;
   return(tampon_con);
}

params.vme_address = DUMMYADD;
params.value = DUMMYVAL;
params.indice = DUMMYVAL;
ioctl(fd, CMIC_RESET, &params);
. . .
params.vme_address  =  (char  *)(CMICVMELOW  +  ACFAXC1  +
i*ACFAVMEOFFSET);
params.value = 0xFE;
params.indice = 7 + i*ACFAINDOFFSET;
```


© pk-mt/enseirb/2005

```
ioctl(fd, DB_VME_WRITE, &params);
. . .
Vsem(semid);
close(fd);

tampon_con.code = CMIC_C_INI;
tampon_con.par0 = B_OK;
tampon_con.par1 = B_OK;

return(tampon_con);
}
```

Remarques :
La carte VME est repérée sous UNIX par le fichier `/dev/cmic`.
Utilisation des E/S bas niveau.
On utilise l'appel `ioctl()` pour la configuration et le contrôle de la carte VME.

© pk-mt/enseirb/2005

On a créé une bibliothèque de fonctions pour le pilotage de la carte VME en ayant une approche de conception modulaire.

16.4. Exemple 4 : pilotage de la liaison série d'un microcontrôleur

But :
Pilotage de la liaison série du microcontrôleur 68HC11. Programme d'écho.
Système :
Aucun. Boucle infinie.

Le programme principal :

```
#include "libhc11.h"

unsigned char c;
char *text = "Programme ping pong \r\n";

void main (void) {
```


© pk-mt/enseirb/2005

```
int i = 0;


baudrate(9600);
while(*(text+i) != '\n') {
        write_com (*(text+i));
        i++;
}
write_com ('\n');

while(1) {
        if (read_com (&c)) {
                if(c == '\r') {
                        write_com ('\r');
                        write_com ('\n');
                }
        else
                write_com(c);       /* echo */
        }
}
}
```


© pk-mt/enseirb/2005

Remarques :

Le programme réalise l'écho de tout ce qui est reçu depuis la liaison série.

Le programme est bâti autour d'une boucle infinie.

Le port série du microcontrôleur est configuré à l'aide de la fonction `baudrate()` fourni par la bibliothèque `libhc11`. L'approche modulaire a été choisie.

On utilise les fonctions spécifiques `read_com()` et `write_com()` pour lire le caractère reçu ou émettre un caractère.

Il semblerait que l'on ne connaisse pas les `printf()` et `scanf()`. C'est une particularité du langage C pour l'embarqué. Ces fonctions sont en fait fournies avec le compilateur C croisé (équivalent de la `libc` UNIX) et il convient d'écrire les fonctions de base `getch()` et `putch()` qui collent au matériel…

La fonction `main()` apparaît comme une simple fonction banale. Il faudra généralement renseigner le vecteur reset du microcontrôleur par l'adresse de son point d'entrée.

© pk-mt/enseirb/2005

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

Bibliothèque libhc11. Extrait :

```
void baudrate(unsigned int bps_rate) {

SCCR1 = 0x00;   /* 1 bit de start, 8 bits de donnees, 1 stop, pas de
parite */
SCCR2 = 0x0C;   /* pas d'interruption, modes de Tx et de Rx valides */
BAUD=0x30;      /* 9600 bps  valeur par defaut, Q = 8 MHz        */

if (bps_rate == 9600)
        BAUD = 0x30;
if (bps_rate == 4800)
        BAUD = 0x31;
if (bps_rate == 2400)
        BAUD = 0x32;
if (bps_rate == 1200)
        BAUD = 0x33;
}
```


© pk-mt/enseirb/2005


```
unsigned char read_com (unsigned char *caractere) {
if (SCSR & 0x20) {    /* si RDFR=1 alors on a recu un caractere   */
   *caractere = SCDR;
   return (1);
}

return (0);                        /* si pas de caractere, 0
*/
}
```

Remarques :

Il faut connaître bien sûr le fonctionnement du périphérique à contrôler.

Script de compilation et d'édition de liens :

```
@echo off
cx6811 -v -e ping1.c
if errorlevel 1 goto bad
:clink
```


© pk-mt/enseirb/2005

```
echo.
echo Linking ...
clnk -o ping1.h11 -m ping1.map ping1.lkf
if errorlevel 1 goto bad
:chexa
echo.
echo Converting ...
chex -o ping1.s19 ping1.h11
if errorlevel 1 goto bad
:cobj
echo.
echo Section size ...
cobj -n ping1.h11
if errorlevel 1 goto bad
echo.
echo.
echo       OK.
goto sortie
:bad
echo.
echo.
```


© pk-mt/enseirb/2005


```
echo       BAD.
:sortie
```

Fichier d'édition de liens :

```
#       link command file for test program
#       Copyright (c) 1995 by COSMIC Software
#
+seg .text -b 0x2000 -n .text   # program start address
+seg .const -a .text            # constants follow program
+seg .data -b 0x3000            # data start address bss
ping1.o                         # application program
libhc11.o                       # 68hc11 enseirb lib
d:/logiciel/68hc11/cx32/lib/libi.h11            # C library
d:/logiciel/68hc11/cx32/lib/libm.h11            # machine library
```

Remarques :

Le code est placé en mémoire à partir de l'adresse 0x2000 (`.text`).

© pk-mt/enseirb/2005

Les données du programme (initialisées ou non `.data`) sont placées à partir de l'adresse 0x3000.
Les modules objets sont relogés suivant un ordre précis le premier étant le programme principal. Le point d'entrée du programme est donc à l'adresse 0x2000 ce qui correspond à l'adresse de la fonction `main()`.

Table des symboles. Mapping :

```
Segments:

start 00002000 end 000021b9 length   441 segment .text
start 000021b9 end 000021d0 length    23 segment .const
start 00003000 end 00003002 length     2 segment .data
start 00003002 end 00003003 length     1 segment .bss

Modules:

ping1.o:
start 00002000 end 00002049 length    73 section .text
start 00003000 end 00003002 length     2 section .data
start 00003002 end 00003003 length     1 section .bss
```


© pk-mt/enseirb/2005


```
start 000021b9 end 000021d0 length    23 section .const

libhc11.o:
start 00002049 end 0000219f length   342 section .text

(d:/logiciel/68hc11/cx32/lib/libm.h11)ilsh.o:
start 0000219f end 000021b9 length    26 section .text

Stack usage:

_AppTickInit  >    2   (2)
_baudrate          6   (6)
_get1A        >    7   (7)
_get1E        >    7   (7)
_get8E        >    2   (2)
_main         >   12   (6)
_put1A        >   10   (10)
_put8A        >    6   (6)
_read_com          6   (6)
_tempo        >    8   (8)
_write_com         6   (6)

Symbols:
```


© pk-mt/enseirb/2005

```
_AppTickInit   0000218f   defined in libhc11.o section .text
                          *** not used ***
_baudrate      00002110   defined in libhc11.o section .text
                          used in ping1.o
_c             00003002   defined in ping1.o section .bss
_get1A         0000207d   defined in libhc11.o section .text
                          *** not used ***
_get1E         0000209d   defined in libhc11.o section .text
                          *** not used ***
_get8E         000020bd   defined in libhc11.o section .text
                          *** not used ***
_main          00002000   defined in ping1.o section .text
                          *** not used ***
_put1A         000020d5   defined in libhc11.o section .text
                          *** not used ***
_put8A         000020c5   defined in libhc11.o section .text
                          *** not used ***
_read_com      0000215c   defined in libhc11.o section .text
                          used in ping1.o
_tempo         00002049   defined in libhc11.o section .text
                          *** not used ***
_text          00003000   defined in ping1.o section .data, initialized
```


© pk-mt/enseirb/2005


```
_write_com     0000217a   defined in libhc11.o section .text
                          used in ping1.o
c_ilsh         0000219f   defined in
(d:/logiciel/68hc11/cx32/lib/libm.h11)ilsh.o section .text
                          used in libhc11.o
```

Fichier de programmation :

```
S00C000070696E67312E6831311C
S12320003C3C304F5FED00CC2580BD2110EC00200B4FBD217AEC00C30001ED00F330001884
S12320208F18E600C10A26E9CC000ABD217ACC3002BD215C5D27F7F63002C10D2608CC0064
S12320400DBD217A20E24F20E23C37363C30EC02ED00201718CE100018EC0EE30018ED1895
S1232060181C2340181F2340FBEC021AEE02830001ED02188C000026DB383838393C373696
S12320803430C601E700A6022704584A26FCE70018CE100018E600E400313838393C373651
S12320A03430C601E700A6022704584A26FCE70018CE100018E60AE400313838393CCE10B6
S12320C000E60A38393C373630E60118CE100018E7003838393C37363C30CC0001ED00E688
S12320E003188FEC00BD219FED00E6095A260C18CE100018E600EA0118E700E609260D1849
S1232100CE1000E6015318E40018E700383838393C3736305F18CE100018E72CC60C18E79D
S12321202DC63018E72BEC001A8325802607C63018E72BEC001A8312C02607C63118E72B2A
S1232140EC001A8309602607C63218E72BEC008304B02605C63318E72B3838393C37363047
S123216018CE1000181F2E200D18E62F1AEE0018E700C60120015F3838393C37363018CEF0
S12321801000181F2E80FBE60118E72F3838393CCE1000EC0EC309C4ED161C22803839189F
S11C21A08C00082508175F188FC008188F188C0000270505180926FB3985
S11A21B950726F6772616D6D652070696E6720706F6E67200D0A0088
```


© pk-mt/enseirb/2005

```
S105300021B9F0
S903FFFFFE
```

Remarques :

C’est un fichier ASCII de type SRecord.

Il est téléchargé via la liaison série pour programmer la mémoire (RAM) à l’aide d’un programme moniteur.

Le lancement du programme se fait à l’aide d’une commande du moniteur (`G 2000`).

© pk-mt/enseirb/2005

16.5. Exemple 5 : pilotage d’un périphérique d’E/S. Structure C

But :

Création d’une structure C qui « mappe » les registres de contrôle, d’état et de données d’un périphérique d’E/S (PIT 68230).

```
typedef volatile struct
{
  unsigned char  PGCR; /*   Port general Control Register */
  unsigned char  vide1;
  unsigned char  PSRR;  /*  Port service request register */
  unsigned char  vide2;
  unsigned char PADDR;  /*  Port A Data Direction Register */
  . . .
  unsigned char  vide5;
  unsigned char PIVR; /*   Port Interrupt Vector Register */
  unsigned char  vide6;
  unsigned char PACR; /*   Port A Control Register */
```


© pk-mt/enseirb/2005

```
  unsigned char  vide7;
  unsigned char PBCR; /*  Port B Control Register */
  unsigned char  vide8;
  unsigned char PADR; /* Port A Data Register */
  unsigned char  vide9;
  . . .
  unsigned char PSR; /* Port Status Register */
  unsigned char  vide14;
  . . .
  unsigned char TSR ; /*Timer Status register  */
} PIT_68230;

/* carte mémoire */

#define ADR_PIT  ((PIT_68230 *) 0xF40081)
```

<u>Remarques :</u>
La structure C « colle » aux différents registres du périphérique qui apparaît dans l'espace d'adressage à l'adresse impaire 0xF40081.

© pk-mt/enseirb/2005

Les registres ne sont qu'aux adresses impaires (voir les `videxx`).
La structure est définie comme `volatile` : on ne demande pas d'optimisation au compilateur C quand on accède aux différents champs de la structure donc aux registres du périphérique.

© pk-mt/enseirb/2005

17. QUELQUES PIEGES CLASSIQUES

17.1. Erreurs sur l'affectation

Ce que le programmeur a écrit	`a == b;`
Ce qu'il aurait dû écrire	`a = b;`

17.2. Erreurs avec les macros

❖ Un `#define` n'est pas une déclaration

Ce que le programmeur a écrit	`#define MAX 10;`
Ce qu'il aurait dû écrire	`#define MAX 10`

Cette erreur peut provoquer ou non une erreur de compilation à l'utilisation de la macro :

© pk-mt/enseirb/2005

- L'utilisation `x = MAX;` aura pour expansion `x = 10;;`

  ce qui est licite : il y a une instruction nulle derrière `x = 10;`.
- L'utilisation `int t[MAX];` aura pour expansion `int t[10;];` ce qui génèrera un message d'erreur.

❖ Un `#define` n'est pas une initialisation

Ce que le programmeur a écrit	`#define MAX = 10`
Ce qu'il aurait dû écrire	`#define MAX 10`

© pk-mt/enseirb/2005

17.3. Erreurs avec l'instruction `if`

Ce que le programmeur a écrit	`if ( a > b) ;` `    a = b;`
Ce qu'il aurait dû écrire	`if ( a > b)` `    a = b;`

17.4. Erreurs avec les commentaires

Il y a deux erreurs avec les commentaires :

1. Le programmeur oublie la séquence fermante `/*`. Le compilateur ignore donc tout le texte jusqu'à la séquence fermante du prochain commentaire.
2. On veut enlever (en le mettant en commentaire) un gros bloc d'instructions sans prendre garde au fait qu'il comporte des commentaires.

© pk-mt/enseirb/2005

17.5. Erreurs avec les priorités des opérateurs

Les priorités des opérateurs sont parfois surprenantes. Les cas les plus gênants sont les suivants :

- La priorité des opérateurs bit à bit est inférieure à celle des opérateurs de comparaison.

Le programmeur a écrit	il désirait	il a obtenu
`x & 0xff == 0xac`	`(x & 0xff) == 0xac`	`x & (0xff == 0xac)`

- La priorité des opérateurs de décalage est inférieure à celle des opérateurs arithmétiques.

Le programmeur a écrit	il désirait	il a obtenu
`x << 4 + 0xf`	`(x << 4) + 0xf`	`x << (4 + 0xf)`

© pk-mt/enseirb/2005

- La priorité de l'opérateur d'affectation est inférieure à celle des opérateurs de comparaison. Dans la séquence ci-dessous, très souvent utilisée, toutes les parenthèses sont nécessaires :

```
while ((c = getchar()) != EOF){
     ...
}
```

17.6. Erreurs avec l'instruction `switch`

❖ Oubli du **`break`**

© pk-mt/enseirb/2005

❖ Erreurs sur le **`default`**

L'alternative à exécuter par défaut est introduite par l'étiquette `default`. Si une faute de frappe est commise sur cette étiquette, l'alternative par défaut ne sera plus reconnue : l'étiquette sera prise pour une étiquette d'instruction sur laquelle ne sera fait aucun `goto`.

```
switch(a){
   case 1 : a = b;
   defult : return(1);      /*   erreur non détectée    */
}
```


© pk-mt/enseirb/2005

17.7. Erreurs sur les tableaux multidimensionnels

La référence à un tableau `t` à deux dimensions s'écrit `t[i][j]` et non pas `t[i,j]`.

© pk-mt/enseirb/2005

18. BIBLIOGRAPHIE

[1] Brian W. Kernighan and Dennis M. Ritchie. *The C programming language*. Editions Prentice Hall, seconde édition, 1988. Le livre sur C écrit par les concepteurs du langage. La seconde édition est conforme au standard ANSI alors que la première édition définissait le langage C dans sa version dite ``Kernighan et Ritchie''.

[2] C - Guide complet. Claude Delannoy. Editions Eyrolles.

19. WEBOGRAPHIE

[1] *Le langage C*. M. Touraïvane, Maître de conférence à l'ESIL.
Ce support de cours est très largement inspiré du travail de ce collègue qui a su synthétiser l'essentiel à savoir sur le langage C. http://www.esil.univ-mrs.fr/~tourai/

© pk-mt/enseirb/2005

[2] *Le langage C adapté aux microcontrôleurs*. Jean Luc PADIOLLEAU.
http://www.ac-orleans-tours.fr/sti-gel/MICROCONTROLEUR/Accueil_Micro.htm

© pk-mt/enseirb/2005