Introduction au langage C - Partie 4

et protection contre quelques bugs

Nancy-Université
Université Henri Poincaré

Brigitte Wrobel-Dautcourt

Octobre 2007

Département Informatique - Faculté des Sciences
Université Henri Poincaré - Nancy 1

Partie 4

14. La gestion de la mémoire
15. Les fonctions
16. Les variables "fichiers"
17. Les directives de compilation
18. Les arguments des commandes shell
19. La fonction *system*
20. Compilation et édition de liens
21. Création de processus

---

14. La gestion de la mémoire

Introduction au langage C
└La gestion de la mémoire

La gestion de la mémoire

1. Laisser faire le compilateur

- l'allocation mémoire est gérée lors de l'exécution du programme, selon les indications données par le compilateur
- la durée de vie des éléments du tableau respecte la durée de vie classique d'une variable

Exemple :

```
void func(int a)
  {
    int b, tab[50], *p ;
    struct noeud n ;

    ...
```

167 / 239

Introduction au langage C
└La gestion de la mémoire

2. Gérer soi-même l'allocation mémoire

(allocation dynamique)

- demande de mémoire dans le tas
- fonctions *malloc(), calloc(), ...*
- inclusion du header **malloc.h**
- profil de la fonction :

```
char *malloc(unsigned size)
```

Paramètre : nombre d'octets à allouer
Résultat :

- pointeur sur la zone mémoire allouée de type `(char *)` car la fonction ne connaît pas le type des objets que l'on va ranger dans la zone allouée.
- pointeur sur `NULL` quand la zone mémoire demandée n'a pu être fournie.

168 / 239

```
int tab[10] ;
```

Le nom `tab` est associé à l'adresse d'implantation de ce tableau statique, il est interdit de souhaiter "déplacer" cette adresse, `tab` ne peut pas être affecté à une valeur :

`tab = ...` est interdit

`tab = malloc(...)` est interdit

`tab = p`, où `p` est déclaré comme `int *p`, est également interdit

Introduction au langage C
└La gestion de la mémoire

Allocation d'une zone de N entiers :

```
int *p ;
p = malloc(4*N) ;
```

qui s'écrit de préférence sous la forme :

```
p = (int *)(malloc( (unsigned)(sizeof(int) * N))) ;
```

Conseil :

- **toujours tester** la valeur du pointeur retourné par ***malloc()***.

```
if (!p) /* équivalent à if (p == NULL) */
  {
    printf("problème d'allocation dynamique\n") ;
    exit(1) ;
  }
```

169 / 239

malloc() alloue size octets, et renvoie un pointeur sur la mémoire allouée. Le contenu de la zone de mémoire n'est pas initialisé.

calloc() alloue la mémoire nécessaire pour un tableau nmemb éléments, chacun d'eux représentant size octets, et renvoie un pointeur vers la mémoire allouée. Cette zone est remplie avec des zéros.

realloc() modifie la taille du bloc de mémoire pointé par ptr pour l'amener à une taille de size octets. realloc() conserve le contenu de la zone mémoire minimum entre la nouvelle et l'ancienne taille. Le con- tenu de la zone de mémoire nouvellement allouée n'est pas initialisé.

sizeof() fait partie du langage C, elle n'a pas de profil défini, ni de manuel associé. Par précaution, on force son résultat à être considéré comme un entier non signé.

La fonction d'allocation donnée en exemple dans la slide peut également s'écrire sous la forme d'une macro-fonction, voir page 159.

Attention :

- la fonction ***strlen()*** ne donne que la longueur "utile" d'une chaîne de caractères :

```
int n ;
char *chaine, *nom ;
...
n = strlen(chaine) ;

nom = malloc(n+1) ;
```

⟵ ☹ `nom = malloc(...)` est faux ; pour stocker une chaîne d'au plus **n** caractères, il faut n+1 cases pour stocker également le marqueur de fin de chaîne : '\0'

```
strcpy(nom,chaine) ;
```


Allocation d'une zone mémoire représentant un tableau bidimensionnel :

- allocation de l'espace pour les données à stocker :

```
int *p ;
p = malloc(nbCols * nbLignes * tailleElement) ;
```


Pour les tableaux monodimensionnels :

- gestion de la mémoire par le compilateur :

```
int tab[N] ;
```

accès à un élément par les deux notations :

```
tab[i]
*(tab+i)
```

- gestion de la mémoire par allocation dynamique :

```
int *tab ;
tab = malloc(N * sizeof(int)) ;
```

accès à un élément par les deux notations :

Pour les tableaux bidimensionnels :

- gestion de la mémoire par le compilateur :

```
int tab[N][M] ;
```

accès à un élément par les trois notations :

```
tab[i][j]
*(tab[i] + j)
*(*(tab[0] + i) + j)
```

- gestion de la mémoire par allocation dynamique :

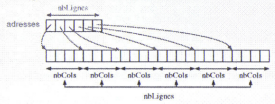
```
int *tab ;
tab = malloc(N * M * sizeof(int)) ;
```

accès à un élément par la **seule** notation :

Si on veut également pouvoir accéder à un élément par la notation indicée :

```
tab[i][j]
```

il faut connaître les adresses de début des tableaux représentant chacune "une ligne" :



on a deux étapes :

- allocation de l'espace pour stocker ces adresses :
- initialisation des éléments de ce tableau d'adresses

Allocation d'une structure :

```
struct point { int x, y ; } ;
struct noeud { int val ;
               struct noeud *fg, *fd ;
             } ;
struct point *p ;
struct noeud *n ;

p = (struct point *) malloc ((unsigned) (sizeof(struct point))
n = (struct noeud *) malloc ((unsigned) (sizeof(struct noeud))
```

Listes chaînées et structures arborescentes :

- création des nœuds un par un par la fonction *malloc()*
- destruction des nœuds un par un par la fonction *free()*

L'accès aux champs d'une structure

notation	*exemple*
nomStructure.champ	`n1.x`
*(pointeurStructure).champ	`*(p).val`
équivalent à pointeurStructure->champ	`p->val`

```
struct noeud n1, *p ;

p = &n1 ;
p->val = 25 ;        ou        *(p).val = 25 ;
p->fg = p->fd = NULL ;
```


Libération d'espace mémoire

```
void free(char *ptr)
```

Libération de l'espace pointé par le pointeur `ptr` dans le cas où le pointeur a été précédemment obtenu par *malloc()*.

Il n'est plus possible d'utiliser un objet après l'avoir libéré, mais on peut à nouveau affecter une autre zone mémoire au pointeur libéré :

```
char *p ;

p = malloc(n) ;
...
free(p) ;
p = malloc(2*n) ;
...
```

Exemple de fuite de mémoire

```
int *tab ;

for (i = 0 ; i < 10 ; i++)
    tab = (int *) malloc(10 * i * sizof(int)) ;

free(tab) ;
```

Il n'existe pas de ramasse-miettes.

15. Les fonctions

Les fonctions

```
type nomFonction (listeArgumentsAvecLeurType)
```

- listeArgumentsAvecLeurType
  passage des arguments par valeur
- type
  type du résultat

Le type du résultat

- type du résultat de la fonction : `int`, `float`, structure, pointeur, ...
- une fonction ne peut retourner (par `return`) qu'une seule valeur.
- la valeur retournée ne peut pas être un tableau ou une fonction, mais peut être un pointeur sur un tableau ou sur une fonction

- quand la fonction n'a aucune valeur retournée : `void`

Passage des arguments par valeur

- valeur d'une variable de type primitif
- valeur de l'adresse d'une variable de type composé (tableau)
- valeur de l'adresse d'une variable de type primitif dont on souhaite modifier le contenu

Passage des arguments par valeur

- valeur d'une variable de type primitif :

```
void impression(int x, int y)
  {
    printf("(%d,%d)\n", x, y) ;
  }
```

appel par :

```
int i, j, t[TAILLE] ;

impression(6, 2) ;
impression(i, j) ;
impression(3*i + j, t[i]) ;
```

Introduction au langage C
└Les fonctions

- valeur d'une variable de type primitif dont on souhaite modifier le contenu :

```
struct point { int x, y ; } ;

void lect(int *, int *) ;

int main ()
   {
     struct point pt ;
     lect([      ], [      ]) ;
     return(0) ;
   }
void lect (int *a, int *b)
   {
     printf("abscisse :  ") ; scanf("%d", [      ]) ;
     printf("ordonnée :  ") ; scanf("%d", [      ]) ;
   }
```

182 / 239

Introduction au langage C
└Les fonctions

- valeur de l'adresse d'une variable de type composé :

```
void lecture(int *px, int t[])
   {
     int *p ;
     scanf("%d", [      ]) ;
     for (p = [      ] ; p < [      ] ; p++)
         scanf("%d", [      ]) ;
   }

int main()
   {
     int nb ;
     int tab[N] ;
     lecture([      ] , [      ]) ;
     return(0) ;
   }
```

183 / 239

La déclaration d'un argument dans une fonction peut être précédée du mot clé **const** précisant ainsi que ce paramètre, qu'il soit une adresse ou une variable, ne peut pas être modifié par la fonction.

Introduction au langage C
└Les fonctions

Tableau en paramètre d'une fonction
Le passage par valeur d'un tableau n'existe pas en langage C.

```
int main(void)
   {
     char chaine[100] ;

     strcpy(chaine,"bonjour") ;
     affiche(chaine) ;
     return(0) ;
   }

void affiche(char ch[100])
   {
     printf("%s\n", ch) ;
   }
```

184 / 239

Introduction au langage C
└Les fonctions

- seule l'adresse de début de la zone mémoire où sont stockées les valeurs est passée à la fonction appelée
- il est inutile de préciser la taille totale du tableau lors de la déclaration de la fonction :

```
void affiche(char ch[])

void affiche(char *ch)
```

185 / 239

Cas d'un tableau à une dimension passé en paramètre d'une fonction

- Si une fonction accepte en paramètre un tableau qui a toujours la même taille, on peut la déclarer sous la forme :

```
void calcul (int tab[TAILLE])
```

- Si la taille du tableau varie d'un appel à l'autre, il faut passer un pointeur sur ce tableau et ajouter éventuellement la taille du tableau en argument :

```
void calcul (int tab[], int nbElements)
void calcul (int *tab, int nbElements)
```

Cas général d'un tableau à n dimensions passé en paramètre d'une fonction

Il faut préciser au moins (n-1) dimensions afin que le compilateur puisse calculer l'adresse d'un élément du tableau (son déplacement par rapport à l'origine de l'implantation du tableau en mémoire).

Par exemple :

```
#define N 10
#define M 100

void calcul (float mat[N][M])
     {
       ...
       t[i][j] = ...  ;
       ...
     }
```

L'accès à l'élément **t[i][j]** est traduit par le compilateur en ***(*t+i*M + j)**, qui a besoin de connaître la valeur de M.

Déclarations correctes (car on donne la valeur d'au moins 1 dimension à la fonction) :

```
void calcul (float mat[N][M])
void calcul (float mat[][M])
void calcul (float mat[N][])
```

Déclaration fausse :

```
void calcul (float mat[][])
```

Introduction au langage C
└Les fonctions

Résultat d'une fonction

Une fonction a :

- aucun résultat : `void`
- un résultat :
  - type primitif : `return`
  - type structuré : `return`
    (adresse d'une zone mémoire contenant plusieurs informations, mécanisme directement géré par le langage)
- plusieurs résultats
  - `return` d'une adresse d'une zone mémoire contenant plusieurs informations (tableau par exemple)
  - adresses mémoire passées en argument de la fonction

186 / 239

Introduction au langage C
└Les fonctions

Résultat d'une fonction (suite)

exemple 1 :

```
int lire_entier()
   { int x ;

     printf("valeur :  ") ;
     scanf("%d",&x) ;
     return(x) ;
   }
```

exemple 2 :

```
struct point lire_point()
   { struct point p ;

     printf("abscisse :  ") ;
     scanf("%d",&p.x) ;
     printf("ordonnee :  ") ;
     scanf("%d",&p.y) ;
     return(p) ;
   }
```

187 / 239

Introduction au langage C
└Les fonctions

Résultat d'une fonction (suite)

exemple 3 :

```
#define TAILLE 100
int *f(int a)
   {
     int table[TAILLE] ; /* incorrect */
     ...
     return(table) ;
   }
```

188 / 239

- soit on déclare le tableau comme une variable statique :

```
#define TAILLE 100
int *f(int a)
   { static int table[TAILLE] ; /* correct */
     ...
     return(table) ;
   }
```

- soit on lui alloue dynamiquement de l'espace si on veut créer un nouveau tableau à chaque appel de la fonction :

```
int *f(int a)
   { int *p ;
     ...
     p = (int*)malloc((unsigned)TAILLE*sizeof(int));
     if (!p) { ...
               exit(...)  ;
             }
     ...
     return(p) ;
   }
```


Fonction passée en paramètre d'une autre fonction

De la même manière que le nom d'un tableau représente son adresse de base, le nom d'une fonction représente l'adresse de celle-ci.

Usage :

- tableau de fonctions
- passage de fonctions en arguments...

Syntaxe :

```
type (*identificateur)(paramètres);
```

- **type** : type de la valeur renvoyée par la fonction,
- **identificateur** : nom du pointeur de la fonction,
- **paramètres** : liste des types des arguments de la fonction

Déclaration "classique" des fonctions qui sont passées en argument :

```
int f1 (int a, float b)
   {
     ...
   }
int f2 (int a, float b)
   {
     ...
   }
```

Déclaration du profil de la fonction qui reçoit un paramètre fonction (paramètre formel) :

```
void executer(float x, int (*f)(int, float))
   {
     int res, ...
     /* utilisation du paramètre formel */
     res = (*f)(5, x) ;
   }
```


Définir le type "profil de la fonction" qui sera passée en argument :

```
typedef int (*Func)(int, float) ;
```

Appel de la fonction acceptant une fonction passée en paramètre (paramètre effectif) :

```
executer(5.  (Func)f1 ) ;
executer(5.*6., (Func)f2 ) ;
```


Exemple de passage de fonctions en paramètre d'une autre fonction :

```
1  #include <stdio.h>

   int addition(int i, int j)  ;
   int soustraction(int i, int j)  ;
5  int multiplication(int i, int j)  ;
   int division(int i, int j)  ;
```

```c
int executer(int i, int j, int (*f)(int, int)) ;

typedef int (*Func)(int, int) ;

int addition(int i, int j)
{
  return i+j ;
}

int soustraction(int i, int j)
{
  return i-j ;
}

int multiplication(int i, int j)
{
  return i*j ;
}

int division(int i, int j)
{
  return i/j ;
}

int executer(int i, int j, int (*f)(int, int))
{
  return( (*f)(i, j) ) ;
}

int main()
{
  int a, b, res ;
  char ch[100] ;

  printf("entier : ") ; scanf("%d", &a) ;
  printf("entier : ") ; scanf("%d", &b) ;

  printf("fonction (+ - * /) : ") ; scanf("%s", ch) ;

  switch (ch[0])
    {
    case '+' :
      res = executer(a, b, (Func) addition) ;
      break ;
    case '-' :
      res = executer(a, b, (Func) soustraction) ;
      break ;
    case '*' :
      res = executer(a, b, (Func) multiplication) ;
      break ;
    case '/' :
      res = executer(a, b, (Func) division) ;
      break ;
    }

  printf("resultat : %d\n", res) ;
  return(0) ;
```



```c
}
```

On peut supprimer l'instruction **switch** en utilisant un tableau de pointeurs de fonctions :

```c
...
/* debut du programme identique au code precedent */
...

int main()
{
  int a, b, op, res ;

  Func tableau[4] ;

  tableau[0] = &addition ;
  tableau[1] = &soustraction ;
  tableau[2] = &multiplication ;
  tableau[3] = &division ;

  printf("entier : ") ; scanf("%d", &a) ;
  printf("entier : ") ; scanf("%d", &b) ;

  printf("fonction (+:0, -:1, *:2, /:3) : ") ; scanf("%d", &op) ;

  res = executer(a, b, (Func) tableau[op]) ;

  printf("resultat : %d\n", res) ;
  return(0) ;
}
```


Pointeur générique

Pour transmettre l'adresse d'une variable dont on ignore le type, on peut utiliser le pointeur générique :

`void *`

Utilisations :

- paramètre d'une fonction
- valeur de retour d'une fonction

Introduction au langage C
└Les fonctions

Paramètres non modifiables

L'attribut `const` placé devant une déclaration informe le compilateur que les variables déclarées ne peuvent pas être modifiées par une instruction du programme.

```
void f(const int i)
    {
      i = 6 ;      interdit
    }
```

194 / 239

Introduction au langage C
└Les fonctions

Liste variable de paramètres

- une fonction avec liste variable de paramètres doit comporter au moins un paramètre formel fixe.
  Exemple de déclaration du profil d'une telle fonction :

  `void f (char *ch, ...)`

- un appel à une fonction avec liste de paramètres s'effectue comme un appel à n'importe quelle autre fonction. cependant, il n'y a pas de mécanisme permettant de détecter automatiquement la fin de la liste des arguments, c'est à l'utilisateur d'ajouter un "marqueur de fin".

  ```
  f("bonjour", NULL) ;
  f("<<", chaine, ">>", NULL) ;
  ```

- la récupération des paramètres de l'appel est détaillée dans le poly...

195 / 239

Dans la fonction à liste variable de paramètres, la récupération des paramètres de l'appel s'effectue au moyen des trois macro-fonctions ***va_start()***, ***va_arg()*** et ***va_end()***. Ces trois macro-fonctions, ainsi que le type **va_list** qu'elles utilisent, sont définis dans le header **stdarg.h** :

```
void va_start(va_list ap, argN);
void va_copy(va_list dest, va_list src);
type va_arg(va_list ap, type);
void va_end(va_list ap);
```


- Il est nécessaire de déclarer, dans le corps de la fonction, une variable de type **va_list** :

  ```
  va_list maListe ;
  ```

- la macro-fonction ***va_start()*** permet de récupérer, dans cette variable, la liste des paramètres optionnels :

  ```
  va_start(maListe, dpf) ;
  ```

  où **dpf** est le nom du dernier paramètre formel, obligatoirement présent dans la liste des paramètres.

- les éléments de la liste des paramètres sont obtenus en itérant des appels à la macro-fonction ***va_arg()*** :

  ```
  char *p = va_arg(maListe, char*) ;
  ```

  – le deuxième élément de cette fonction est le type du paramètre à récupérer,
  – chaque appel retourne un nouvel élément.

- la fonction ***va_end()*** permet de clore le parcours de la liste des paramètres.

Exemple :

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

    void f(char *ch, ...)
5   {
      va_list maListe ;
      char *p ;
      va_start(maListe, ch) ;
      p = va_arg(maListe, char *) ;
10    while (p)
        {
          printf("parametre trouve : %s\n", p) ;
          p = va_arg(maListe, char *) ;
        }
15  }

    int main()
    {
      f("une", NULL) ;
20    f("ceci", " est", " un", " chat", NULL) ;
      return(0) ;
    }
```

Appel et déclaration d'une fonction (en java)

```
package truc ;
public class MaClasse {
   // champs privés
   protected int valeur ;
   // constructeurs
   protected MaClasse(int val) {
      setValeur(val) ;
   ...
   }
   // fonctions d'observation
   // fonctions de transformation
   void setValeur(int i) {
      valeur = i ;
   }
} // classe MaClasse
```


Appel et déclaration d'une fonction (en C)

```
#include <stdio.h>

int main()
   {
      fonction(i) ;
      ...
   }
void fonction(int a)
   {
      ...
   }
```


Appel et déclaration d'une fonction (suite)

- Le langage C :
  - permet d'appeler une fonction non déclarée
  - mais ne vérifie ni ses arguments, ni son retour
  - car le compilateur n'examine qu'un seul fichier à la fois

  Ceci est source d'erreurs considérables !
- **Pour imposer une vérification :**
  - la fonction doit être **déclarée** (donner son profil)
  - dans le fichier où elle est **appelée** et
  - **avant l'appel**

Fonctions (suite)

Problèmes :

- cohérence de la déclaration et de la définition

Solution :

- inclure la **même déclaration** dans les fichiers
  - où la fonction est définie
  - et ceux où elle est déclarée
- via les fichiers partagés : **headers**

Principe :

- dans le fichier `.c` : définition de la fonction
- dans le fichier `.h` : déclaration de la fonction
- inclure `.h` dans tous les fichiers `.c`

Fonctions (suite)

Headers standards :

- déclaration des fonctions des bibliothèques :
  - **stdio.h** : fonctions d'entrées-sorties
  - **math.h** : fonctions mathématiques
  - **string.h** : fonctions de manipulation de chaînes de caractères
  - ...

Il y a un fichier **header** pour chaque librairie.

Recherche des fichiers headers :

- header utilisateur : `#include "truc.h"`
- header standard : `#include <truc.h>`
  recherche dans :
  - répertoires standards (/usr/include ...)
  - répertoires indiqués en option **-I** du compilateur **gcc**

Fonctions (suite)

Remarque :

- l'édition de liens est indépendante de la compilation
- un programme peut compiler (en trouvant les bonnes librairies) même s'il n'utilise pas les bons headers... !

Ne pas oublier d'inclure les headers, y compris pour les fonctions des librairies !

16. Les variables "fichiers"

Les variables "fichiers"

Dans un programme, à tout fichier manipulé, correspond une variable de type fichier indiquant :

- nom physique
- type d'accès : lecture, écriture, ...
- l'adresse du buffer
- la position des caractères dans le buffer, ...

La déclaration d'une variable de type fichier

```
FILE *fich ;
```

Le type `FILE` est défini dans le fichier **stdio.h**

avec inclusion du fichier **`/usr/include/stdio.h`** :

Extrait du fichier stdio.h

```
...
#  ifndef NULL
#    define     NULL    0
#  endif /* NULL */
...
#  ifndef EOF
#    define     EOF     (-1)
#  endif /* EOF */
...
  typedef struct {
        int              __cnt;
        unsigned char   *__ptr;
        unsigned char   *__base;
        unsigned short   __flag;
        unsigned char    __fileL;
                 /* low byte of file desc */
        unsigned char    __fileH;
                 /* high byte of file desc */
  } FILE;
...
```

Introduction au langage C
└Les variables "fichiers"

L'ouverture d'un fichier

Pour accéder à un fichier, il faut :

- faire le lien entre le nom physique et le nom logique
- l'ouvrir dans le mode choisi : fonction *fopen()*
- se positionner au début du fichier

203 / 239

Introduction au langage C
└Les variables "fichiers"

```
fich = fopen(nomPhysique,mode)
```

nomPhysique : chaîne de caractères représentant le nom du fichier (nom absolu ou relatif).

mode : mode d'ouverture du fichier (`"r"`, `"w"`, ...)

fich : résultat de la fonction :

- soit le pointeur associé au descripteur du fichier si il existe,
- soit le pointeur `NULL` (0) sinon
  - `fopen(nom,"r")` → 0 quand le fichier n'existe pas
  - `fopen(nom,"w")` → 0 quand le chemin d'accès est faux

204 / 239

mode :

`r`	ouverture en lecture
`w`	pour tronquer ou création en écriture
`a`	ouverture en écriture en fin de fichier, ou création en écriture
`r+`	ouverture en mise-à-jour : lecture et écriture
`w+`	pour tronquer ou ouverture pour mise-à-jour
`a+`	mode "ajout" : ouverture ou création pour ajouter en fin de fichier (après l'actuel `EOF`)

nomPhysique : le nom physique est soit :

- une constante chaîne de caractères quand le nom du fichier est figé
  `fich = fopen("donnees.txt","r") ;`
- une variable chaîne de caractères (à condition que cette chaîne soit effectivement initialisée...) :

```
char nom [50] ;
...
fich = fopen(nom,"w") ;
```

Introduction au langage C
└Les variables "fichiers"

exemple :

```
FILE *fich ;
char nom[100] ;

do {
     printf("nom du fichier a lire :  \n") ;
     scanf("%s",nom) ;
     fich = fopen(nom,"r") ;
   }
   while (!fich) ;
```

205 / 239

Le mode `r+` permet d'ouvrir le fichier à la fois en lecture et en écriture, mais il n'est cependant pas possible de mêler librement les opérations de lecture et d'écriture : il faut appeler la fonction *fseek()* (voir cette fonction page 156) entre deux opérations de lecture et d'écriture.

Exemple :

```
while (fread((char *)&rec, sizeof(rec), 1, fp) == 1)
      {
        /* faire un traitement à l'enregistrement rec */
        ...
        if ( ...  /* on souhaite réécrire l'enregistrement rec */ ...)
           {
             fseek(fp, -(long)sizeof(rec), 1) ;
             fwrite((char *)&rec, sizeof(rec), 1, fp) ;
             fseek(fp, 0L, 1) ;
      }
      }
```

- le premier appel de *fseek()* permet de se positionner au début de l'enregistrement à modifier.
- le deuxième appel de *fseek()* ne fait rien : il place simplement le fichier dans un mode où la lecture puisse avoir lieu.

Les 5 types d'entrées-sorties (Inclure le fichier stdio.h)

- e/s caractère par caractère :

```
int getc (FILE *stream) ;        int putc(int c, FILE *stream) ;
int fgetc(FILE *stream) ;        int fputc(int c, FILE *stream) ;
int getchar(void) ;              int putchar(int c) ;
```

- e/s mot par mot (un mot = quelques octets, cela dépend des machines...)

```
int getw(FILE *stream) ;         int putw(int w, FILE *stream) ;
```

- e/s chaîne de caractères :

```
char *gets(char *s) ;            int puts(const char *s) ;
char *fgets(char *s, int n, FILE *stream) ;
int fputs(const char *s, FILE *stream) ;
```

- e/s formatées :

```
int fscanf(FILE *stream, format, listeAdresses ...)  ;
int fprintf(FILE *stream, format, listeVariables ...)  ;
```

- e/s binaires bufferisées :

```
size_t fread(void *ptr, size_t size, size_t n, FILE *stream) ;
size_t fwrite(const void *ptr, size_t size, size_t n, FILE *stream) ;
```

Introduction au langage C
└ Les variables "fichiers"

Les entrées/sorties caractère par caractère
getc() *putc()* *fgetc()* *fputc()*

```
int getc (FILE *stream) ;
```

```
int main()
  {
    int c ;
    FILE *fich ;

    fich = fopen("donnees","r") ;
    while ((c = getc(fich)) != EOF)
          {
            ... utilisation de c ...
          }
    ...
  }
```

206 / 239

Il faut déclarer de type `int` et non `char` une variable destinée à recevoir un caractère lu par une fonction de lecture de caractère afin de permettre un traitement correct de la reconnaissance de la constante *finDeFichier* `EOF`, de valeur -1.

`int getc (FILE *stream)` est définie sous forme d'une macro-fonction. Elle retourne un caractère lu (de type `int`) dans le fichier ou la constante `EOF` en cas d'erreur ou de détection de fin de fichier. Elle est équivalente à la fonction ***fgetc()***.

`int getchar(void)` est définie sous forme d'une macro-fonction. Elle est équivalente à la fonction ***getc()*** sur l'entrée standard :

```
# define getchar() getc(stdin)
```

`int fgetc(FILE *stream)` retourne un caractère lu (de type `int`) dans le fichier ou la constante `EOF` en cas d'erreur ou de détection de fin de fichier. Elle est équivalente à la macro-fonction ***getc()*** mais elle est plus lente, prend cependant moins de place à l'appel et son nom peut être passé en argument d'une autre fonction.

`int getw(FILE *stream)` retourne un mot (dont la taille varie en fonction des machines) dans le fichier ou la constante `EOF` en cas d'erreur ou de détection de fin de fichier. À ne pas utiliser !

`char *gets(char *s)` ; lit une ligne sur l'entrée standard, jusqu'à un caractère '\n' ou une fin de fichier, et la recopie à partir de l'adresse `s`, en remplaçant le caractère '\n' par le caractère '\0'. Elle retourne un pointeur ou `NULL` si une erreur s'est produite.

Nous vous recommandons de ne pas utiliser cette fonction qui perd des caractères quand la chaîne lue contient des caractères nuls et qui ne teste pas le débordement mémoire (quand la chaîne lue est plus grande que la zone mémoire réceptrice).

`char *fgets (char *s, int size, FILE *stream)` ; lit au plus `size - 1` caractères depuis `stream` et les place dans le buffer pointé par `s`. La lecture s'arrête après `EOF` ou un "fin de ligne". Si une "fin de ligne" est lue, elle est placée dans le buffer. Un caractère nul '\0' est placé à la fin de la ligne.

`int putc(int c, FILE *stream)` est définie sous forme d'une macro-fonction. Elle écrit sur le fichier le caractère `c` et retourne le caractère écrit ou `EOF` en cas d'erreur. Elle est équivalente à la fonction ***fputc()***.

`int putchar(int c)` est définie sous forme d'une macro-fonction. Elle est équivalente à la fonction ***putc()*** sur la sortie standard :

```
#define putchar(__c) putc((__c), stdout)
```

`int fputc(int c, FILE *stream)` écrit sur le fichier le caractère `c` et retourne le caractère écrit ou `EOF` en cas d'erreur. Elle est équivalente à la macro-fonction ***putc()***, mais elle est plus lente, prend cependant moins de place à l'appel et son nom peut être passé en argument d'une autre fonction.

`int putw(int w, FILE *stream)` écrit un mot (dont la taille varie en fonction des machines) dans le fichier. À ne pas utiliser.

`int puts(const char *s)` recopie la chaîne sur la sortie standard en ajoutant à sa suite le caractère '\n'. Elle retourne `EOF` sur erreur.

Les entrées-sorties formatées

Les fonctions ***fprintf()*** et ***fscanf()*** s'utilisent exactement de la même façon que les fonctions ***printf()*** et ***scanf()***, il suffit d'ajouter un paramètre pointeur de fichier :

```
FILE *fichier ;
int x = 25 ;

fprintf(fichier, "%d\n", x) ;
```

permet d'écrire la valeur de **x** dans le fichier texte préalablement ouvert en écriture.
Cette donnée sera lue dans le fichier par l'instruction :

```
int a ;

fscanf(fichier, "%d", &a) ;
```


Les entrées-sorties binaires bufferisées

Principe : écrire directement la représentation machine des variables (c'est-à-dire leur contenu mémoire)

```
short x ;
x = 1024 ;
```

x :
00000010000000000
x occupe 2 octets

- `fprintf(fich, "%d", x)` écrit les 4 caractères "1024" dans le fichier texte
- `fwrite(&x, sizeof(x), 1, fich)` écrit les 2 octets dans le fichier

```
typedef unsigned int size_t ;
size_t fread(void *ptr, size_t size, size_t n, FILE *stream) ;
size_t fwrite(const void *ptr, size_t size, size_t n, FILE *stream)
```


Inconvénients :

- non portable, car la représentation et la taille des objets diffèrent d'une architecture à l'autre
- non lisible par les outils usuels (**more**, **emacs**, ...)
  il est nécessaire d'écrire un programme C :
  - qui lit par ***scanf()*** et écrit par ***fwrite()*** pour construire un tel fichier,
  - qui lit par ***fread()*** et imprime par ***printf()*** pour afficher un tel fichier

Avantages :

- stockage réduit
- lectures et écritures efficaces

```
char image[NBLIGNES][NBCOLS] ;
fread ( image[0] , sizeof(char) , NBLIGNES*NBCOLS, fich ) ;
```

adresse de la zone	taille de chaque élément	nombre d'éléments	pointeur de fichier

La fin d'un fichier

- certaines fonctions retournent la valeur `EOF` (-1) lorsque la fin du fichier est atteinte :

```
int main(void)
{
  int c ;

  while ((c = getchar()) != EOF)
        /* utilisation de c */...
  return(0) ;
}
```

- la fonction ***feof()*** :

```
int feof(FILE *nomFichier)
```

la fonction retourne : $\neq$ **0** si la fin du fichier est atteinte
0 sinon

La fermeture d'un fichier

```
int fclose(FILE *nomFichier)
```

la fonction retourne : 0 quand la fermeture s'est déroulée sans erreur, EOF sinon.

il est possible de fermer un fichier pour ensuite l'ouvrir dans un autre mode :

```
FILE *fich ;
fich = fopen("donnees","w") ;
...
fclose(fich) ;
fich = fopen("donnees","r") ;
...
```

Par défaut, tous les fichiers ouverts sont automatiquement fermés lors de la sortie (terminaison) du programme.

Fonction d'accès à un emplacement particulier

```
int fseek(FILE *stream, long int offset, int whence)
```

Cette fonction modifie la position courante associée au flot d'entrée/sortie. Le mode de positionnement est spécifié par l'argument **whence** :

- **SEEK_SET** : début de fichier
- **SEEK_CUR** : relativement à la position courante
- **SEEK_END** : fin de fichier

Les fichiers standards

Les pointeurs de fichiers **stdin**, **stdout** et **stderr** sont prédéfinis ; ils peuvent être utilisés partout où l'on peut trouver des objets du type **FILE***.

Extrait de **/usr/include/stdio.h** :

Extrait du fichier stdio.h

```
1  ...
   # define      stdin   (&__iob[0])
   # define      stdout  (&__iob[1])
   # define      stderr  (&__iob[2])
5  ...
```

Ce sont des constantes et non des variables, il est inutile de leur affecter une nouvelle valeur.

Utilisation classique d'un pointeur de fichier :

```
fprintf(stderr,"ouverture impossible du fichier %s\n",nom) ;
```


17. Les directives de compilation

Les directives de compilation

Trois catégories :

- directives de substitution symbolique :
  - `#define`
  - `#undef`
- directives d'inclusion de fichier :
  - `#include`
- directives de compilation conditionnelle :
  - `#if`
  - `#ifdef`
  - `#ifndef`
  - `#elif`
  - `#else`
  - `#endif`

Les directives sont interprétées par le préprocesseur C.

Les directives de substitution symbolique

```
#define nom suiteDeCaractères
```

```
#define nom(listeSymboles) suiteDeCaractères
```

```
#undef nom
```

- définition de constantes :

```
#define PI 3.1415927
```

- définition de symboles :

```
#define BEGIN {
```

Introduction au langage C
└Les directives de compilation

fonction "classique" :

```
int abs (int x)
  {
    return(x > 0 ?  x :  -x) ;
  }


int main(void)
  {
    int a, b ;
    float c, d ;
    ...
    a = abs(b) ;
    a = abs(b-5) ;
    c = [ ??? ](3.*d + 6.)  ;
    ...
  }
```

214 / 239

Introduction au langage C
└Les directives de compilation

définition de macro-fonctions :

```
#define ABS(v) (v > 0 ?  v :  -v) /* incorrect */


int main(void)
  {
    int a, b ;
    float c, d ;
    ...
    a = ABS(b) ;
    a = ABS(b-5) ;
    c = ABS(3.*d + 6.)  ;
    return(0) ;
  }
```

215 / 239

Introduction au langage C
└Les directives de compilation

Effets de bords :

```
ABS(x++)
```

est traduit en :

```
((x++) > 0 ?  (x++) :  -(x++))
```

qui incrémente **deux** fois la variable **x**...

216 / 239

exemple de macro-fonction pour réaliser l'allocation dynamique :

```
#define allocation(pointeur,type) \
    { pointeur = (type*)(malloc((unsigned)(sizeof(type))));\
      if (!pointeur) \
        { printf("probleme d'allocation dynamique\n") ;\
          exit(1) ; \
        } \
    }
```

déclaration d'un pointeur de structure :

```
struct point { int x ; float y ; } *p1 ;
```

allocation d'une structure :

```
allocation(p1, struct p1) ;
```

macro-fonction d'allocation d'une zone de N éléments de taille donnée par le type :

```
#define alloc_n(pointeur,type,nombre) \
    { pointeur = (type*)(malloc((unsigned)(sizeof(type)*nombre)));\
      if (!pointeur) \
        { printf("probleme d'allocation dynamique\n") ;\
          exit(1) ; \
        } \
    }
```

seul l'argument nombre est ajouté par rapport à la macro-fonction **allocation**.

allocation d'un tableau de structures :

```
alloc_n(tab, struct point, 10) ;
```

Attention, les macro-fonctions ne sont pas des fonctions, les parenthèses et les espaces doivent être placés judicieusement sinon le pré-processeur ne fait pas les substitutions attendues par le programme.

Dans une définition de macro-fonction, toute occurrence d'un paramètre dans le corps de la définition doit être placée entre parenthèses. Lorsque le corps de la macro-fonction définit une expression C, il est lui-même écrit entre parenthèses. Par exemple :

```
#define CARRE(x) ((x)*(x))
```

Il ne faut pas laisser d'espace entre le nom de la macro-fonction et la parenthèse ouvrante définissant la liste des paramètres :

```
#define FONC (y) ((y)*3)
```

l'instruction :

```
z = FONC(x) ;
```

est alors remplacée par :

```
z = (y)((y)*3) (x) ;
```

alors que la définition :

```
#define FONC(y)((y)*3)
```

permet de remplacer l'instruction :

```
z = FONC(x) ;
```

par :

```
z = (( x )*3) ;
```

ce qui est le résultat attendu.

La directive **`#include`** permet d'inclure le texte d'un fichier avant la compilation de l'ensemble. On utilise cette facilité afin d'inclure des fichiers de définitions (de macro-constantes, de macro-fonctions, de définitions de types, de structures, d'énumérations, ...). En aucun cas, il ne faut utiliser cette directive pour inclure des instructions de fonctions (`.c`) afin de faire croire à de la compilation séparée.

Introduction au langage C
└Les directives de compilation

Les directives conditionnelles
Les directives conditionnelles sont évaluées au cours du prétraitement et permettent ainsi de contrôler le fonctionnement du préprocesseur. Elles permettent de tester l'existence et la valeur de macro-constantes et donc de conserver ou d'enlever certaines lignes du fichier source.

= **compilation conditionnelle**

La ligne `#if` évalue une expression entière constante.

217 / 239

Introduction au langage C
└Les directives de compilation

Exemples :

```
...
int i ;
float f, g ;
i = fonction(&f, &g) ;
```

permet de s'assurer que l'on n'inclut qu'une seule fois le fichier `debut.h`.

218 / 239

`#if` - `#endif` : ces directives permettent de choisir le bout de code à inclure dans le fichier source sous certaines conditions.

utilisation principale :

`#if defined (constante)`, raccourci en **`#ifdef constante`**

`#if !defined (constante)`, raccourci en **`#ifndef constante`**

Introduction au langage C
└Les directives de compilation

un.h
```
#include "deux.h"
...
```

deux.h
```
#include "un.h"
...
```

prog.c
```
#include "un.h"
#include "deux.h"
...
```

→ boucle infinie dans les inclusions de fichiers d'en-tête
`macro or '#include' recursion too deep`
La solution :

un.h

deux.h

prog.c
```
#include "un.h"
#include "deux.h"
...
```

219 / 239

Structure d'un fichier header

```
#ifndef FICH_H
#define FICH_H
...
/* liste des définitions */
...
#endif
```


18. Les arguments des commandes

Les arguments des commandes

l'exécution d'un programme dans le shell :

- exécution d'un script shell

```
bash-3.00$ ./commande a1 a2 a3 ...  an
```

- exécution d'un programme Java

```
bash-3.00$ jmlrac pack/TestProg a1 a2 a3 ...  an
```

- exécution d'un programme C

```
bash-3.00$ ./programmeC a1 a2 a3 ...  an
```


Les arguments des commandes (suite)

- exécution d'un script shell

```
./commande a1 a2 a3 ...  an
```


Les arguments des commandes (suite)

- exécution d'un programme Java

```
jmlrac pack/TestProg a1 a2 a3 ...  an
```

Le *shell* fournit au programme Java lancé un tableau permettant d'accéder aux arguments.

- la déclaration dans le code Java :

```
public static void main(String[] arg)
```

Les arguments des commandes (suite)

- exécution d'un programme C

```
./programmeC a1 a2 a3 ...  an
```

Le *shell* fournit au programme C lancé un pointeur **argv** permettant d'accéder aux arguments.

- la déclaration dans le code C :

```
int main (int argc, char *argv[])
```


On peut également écrire le profil :

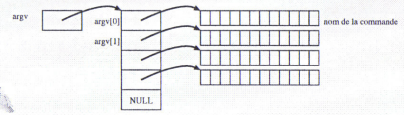
```
main (int argc, char **argv)
```

Remarque :

On peut nommer les paramètres de la fonction ***main()*** comme on veut, mais il est souhaitable de conserver les noms **argc** et **argv** pour ne pas embrouiller le lecteur.

La structure du tableau des arguments

- **argc** : nombre d'arguments, nom de la commande **inclus**
- **argv** : (tableau de pointeurs) tableau de chaînes de caractères



Ne pas confondre avec `$#argv` en *csh* qui contient le nombre d'arguments de la commande, nom de la commande **exclu**.

- **Remarque :** quand un des arguments est un nombre, il faut transformer sa représentation sous forme d'une chaîne de caractères en valeur.

Exemple 1

programme :

```
int main (int argc, char *argv[])
  {
    printf("%s a %d ans\n",argv[2], atoi(argv[1])) ;
  }
```

Commande : `calcul` `25` `albert`
argv[0] argv[1] argv[2]

la ligne de commande a 3 arguments

Résultat :

```
albert a 25 ans
```


Exemple 2

Utilisation de cette fonction pour vérifier une syntaxe de programme de type :

```
prog -in <fichier> -out <fichier>
```

où les arguments `-in` et `-out` sont nécessairements suivis d'un nom de fichier et peuvent se présenter dans n'importe quel ordre sur la ligne de commande

```
int main(int argc, char *argv[])
  { int k ;
    char *in_f = NULL, *out_f = NULL ;
    k = findArg(argc, argv, "-in") ;
    if (k > 0 && k < argc-1) in_f = argv[k+1] ;

    k = findArg(argc, argv, "-out") ;
    if (k > 0 && k < argc-1) out_f = argv[k+1] ;
    ...
  }
```

Exemple 3

Soit la fonction `findArg` qui retourne l'indice de la chaîne `mot` dans la liste de `argc` mots de `argv` ou -1 si cette chaîne n'est pas présente :

```
int findArg(int argc, char *argv[], char *mot)
  {
    while (argc--)
          if (!  strcmp(argv[argc], mot))
                return(argc) ;
    return(argc) ;
  }
```


Exemple 4

Programme qui reconnaît les options :

- -a : ajouter
- -c : changer
- -d : détruire
- -l : lister

```
#include <stdio.h>

void main(int argc, char *argv[])

 {
   if (--argc < 1 || *argv[1] != '-')
          usage() ;

   switch (*++argv)
    {
       case 'a' :
                    ajouter(argc, argv) ;
                    break ;
       case 'c' :
                    changer(argc, argv) ;
                    break ;
       case 'd' :
                    detruire(argc, argv) ;
                    break ;
       case 'l' :
                    lister(argc, argv) ;
                    break ;
       default : usage() ;
    }
 }

void ajouter (int argc, char *argv[])

 {
   if (!--argc)
          usage() ;
   ++argv ;
   ...
 }

void usage()

 {
   printf("usage :\n") ;
   printf("\t commande -a chaine --> [ajouter]\n") ;
   printf("\t commande -c chaine --> [changer]\n") ;
   ...
   exit(1) ;
 }
```

19. La fonction system()

La fonction system()

- transmission d'une commande à l'environnement d'exécution :

```
#include <stdlib.h>
int system(const char *commande) ;
```

  - **commande** est une chaîne de caractères, exprimée dans le langage de programmation de l'environnement (problème de portabilité) :
  - ***ksh*** si l'application est conforme aux standards
  - ***sh*** pour votre machine

Exemple :

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

int main(void)
  {
    system("date") ;
    printf("fini\n") ;
    return(0) ;
  }
```

- le résultat de la fonction `system(NULL) ;` permet de tester l'existence d'un interpréteur
  (code de retour $\neq 0$ si l'interpréteur existe)

L'instruction suivant l'appel de la fonction ***system()*** est exécutée quand la commande *shell* est terminée.

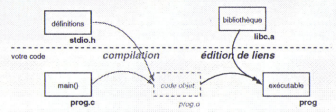
L'instruction `system("ll") ;` ne fonctionne pas si `ll` est défini par un mécanisme d'alias.

Le résultat de la fonction ***system()*** est le code de la commande retourné par ***exit()***.

20. Compilation et édition de liens

Compilation d'un programme C (suite)

```
#include <stdio.h>        inclusion de fichiers de définition
#define N 10

int f (int) ;

int f (int p)
      {
        return(p * N) ;
      }

int main(void)
      {
        int i = 5, j = 6, a ;

        i = f(i) ;
        a = f(j) ;
        printf("résultats :  %d, %d\n", i, a) ;
        return(0) ;
      }
```

aucun changement pour la commande de compilation :

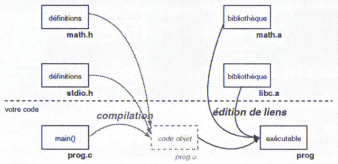
```
gcc prog.c -o prog
```

Utilisation de fonctions de la bibliothèque mathématique

```
bash-3.00$ man sin
Mathematical Library Functions                    sin(3M)

NAME
     sin - sine function

SYNOPSIS
     cc [ flag ... ] file ... -lm [ library ... ]
     #include <math.h>

     double sin(double x);
...
```

l'aide en ligne donne les informations :

- profil de la fonction
- nom du fichier de définitions à inclure pour la compilation
- nom de la bibliothèque à inclure pour l'édition de liens



Ajout de la bibliothèque mathématique pour l'édition de liens :

```
gcc prog.c -o prog -lm
```


Programme C réparti sur plusieurs fichiers

def.h :

```
#include <stdio.h>
#include <math.h>
#define N 10
extern double f (double) ;
```

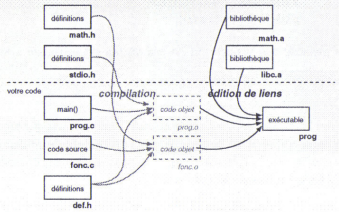
prog.c :

```
#include "def.h"
void main(void)
     { double a, b ;
       b = f(5.)  ; a = f(6.)  ;
       printf("Constante :  %d\n", N) ;
       printf("résultats :  %f, %f\n", b, a) ;
     }
```

fonc.c :

```
#include "def.h"
double f (double p)
```


- La notation **<fich.h>** permet au compilateur de rechercher le fichier de définitions **fich.h** par défaut sous le répertoire **/usr/include**.
- La notation **"fich.h"** permet au compilateur de rechercher le fichier de définitions **fich.h** par défaut sous le répertoire courant.




```
gcc prog.c fonc.c -o prog -lm
```

Résumé :

```
#include <stdio.h>
...
int main(void)
   { ...
     ...
   }
```

```
#include <stdio.h>
#include <math.h>
...
int main(void)
   { ...
     x = sin(y) ;
     ...
   }
```

`# include < ... .h >` ne provoque pas obligatoirement un appel explicite à une bibliothèque lors de la commande de l'édition de liens.

21. Création de processus

1. création d'un processus fils par **duplication** du processus père
2. **remplacement** éventuel du code exécutable

Duplication du processus

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void) ;
```

- le processus fils possède son propre *pid* (le *pid* du fils est initialisé avec le *pid* du père)
- réinitialisation des horloges internes
- ...

Identification du processus qui exécute le programme par le code de retour de la fonction ***fork()***

- processus père : ***fork()*** retourne le *pid* du fils
- processus fils : ***fork()*** retourne 0

```
int pid ;

pid = fork() ;
if (pid == 0)
   printf("je suis le fils\n") ;
else
   printf("je suis le pere\n") ;
```


Remplacement du code exécutable du processus fils

```
#include <unistd.h>
int execl (const char *chemin, const char *arg, ...)
```

Six fonctions pour charger un nouvel exécutable :

`execl, execv, execle, execve, execlp, execvp`

```
int pid ;

pid = fork() ;
if (pid == 0)
   execl("/users/prof/wrobelb/=sys/fils", "fils", NULL) ;
else
   printf("je suis le pere\n") ;
```

"Programmer en C"

Style de programmation

Un programmeur peut grandement faciliter le travail de maintenance en écrivant des programmes faciles à lire et à modifier par d'autres programmeurs. Un bon style de programmation s'acquiert seulement avec de l'expérience et nous vous recommandons d'écrire "spontanément", dès les premiers programmes, du code facile à comprendre par d'autres programmeurs. Il n'existe pas de formule magique qui garantisse l'écriture de programmes lisibles, mais il y a plusieurs règles de bon sens à observer :

1. Découper un programme en modules cohérents.
2. Organiser un programme de manière que sa structure soit claire.
3. Écrire des commentaires intelligents pour expliquer un programme. Décrire clairement et précisément les modèles de données sous-jacents, les structures de données sélectionnées pour les représenter et l'opération accomplie par chaque fonction. Lors de l'écriture d'une fonction, établir les hypothèses faites sur les entrées et dire comment la sortie se rapporte aux entrées.
4. Utiliser des noms ayant un sens pour les fonctions et les variables.
5. Lorsque c'est possible, éviter les constantes explicites, par exemple, ne pas utiliser 7 pour le nombre de nains. À la place, il est préférable de définir une constante comme `NombreDeNains` de façon à ce que l'on puisse facilement changer toutes les utilisations de cette constante en 8 si on décide d'ajouter un autre nain.
6. Éviter d'utiliser des variables globales, c'est-à-dire des variables définies pour tout le programme afin de limiter les effets de bords des fonctions.

Construire un programme robuste qui gère les situations qui ne doivent pas survenir, mais qui peuvent survenir quand même.

Un autre bon exercice de programmation est de maintenir un jeu d'essais destinés à tester chaque ligne de code pendant que vous développez un programme. Quand de nouvelles fonctionnalités sont ajoutées à un programme, la suite de tests peut être exécutée pour s'assurer que le nouveau programme a le même comportement que l'ancien sur les entrées qui fonctionnaient précédemment.

Conseils de programmation

La **portabilité** d'un programme C est une importante qualité ; elle demande d'écrire des programmes C qui peuvent être transférés d'une machine à une autre, sous le système Unix, (presque) sans changement. Cependant, il n'existe pas de règles absolues pour garantir qu'un programme soit portable, juste des conseils de programmation.

La suite de ce document présente donc des conseils, et non des règles, pour écrire des programmes C qui soient **mieux organisés**, plus **faciles à lire** et **à mettre à jour** et qui soient plus **portables**.

Organisation d'un programme C

Le mécanisme de compilation séparée offert par C doit inciter le programmeur à découper son application en plusieurs modules source distincts ; chaque module réalisant une tâche particulière de l'application. La taille d'un module ne doit pas excéder quelques centaines de lignes C.

La structure de chaque fichier source devrait adopter l'organisation proposée ci-dessous en suivant cet ordre chronologique :

1. Un **bloc commentaire** de plusieurs lignes qui décrit le module en essayant de respecter les conventions de présentation suivantes :
   - une identification du module par :
     - son nom
     - son numéro de version
     - sa date de première réalisation
     - sa date de dernière modification
   - une description sommaire de ce que fait le module. Comment et par qui il est utilisé, ses principales caractéristiques, etc.
   - le nom du ou des auteurs ou des personnes qui ont modifié le module.
   - un bref rappel historique des versions qui se sont succédées.
2. Les **inclusions de fichier** (lignes `#include`) qui énumèrent dans l'ordre les fichiers en-tête prédéfinis dans le système de type `<fichier.h>`, puis les fichiers propres à l'utilisateur, de type `"fichier.h"`.

```
#include <stdio.h> /* lib.  standard d'E/S */
#include <ctype.h> /* types des caractères */
#include "global.h" /* variables externes communes */
```

3. Les **définitions de constantes** symboliques et de macro-instructions (lignes `#define`) suivies des définitions de noms de types (lignes `typedef`).
4. Les **définitions des variables globales** au module (classe `static`) et les déclarations des variables externes importées (spécificateur `extern`). Les variables globales externes devraient figurer dans un fichier en-tête spécifique appelé par `#include`. En revanche, les variables globales du module devraient être invisibles des autres modules (fichiers) et dont être explicitement déclarées comme statique.

```
static char ligne[80] ; /* tampon de 80 octets */
```

5. Les **fonctions**. Chaque fonction devrait se plier aux règles de présentation suivantes :
   - un commentaire d'introduction sur plusieurs lignes spécifiant dans l'ordre :
     - le nom de la fonction,
     - une description sommaire du rôle de cette fonction,
     - la description des paramètres fournis en entrée,
     - la nature et le type de la valeur retournée.

     Dans le cas où la fonction n'a pas de résultat retourné par l'instruction `return()`, il faut indiquer `void` comme type de résultat, exemple :

     `void imprimer(int a)`
   - la définition de la fonction : son type, c'est-à-dire celui de la valeur retournée, devant précéder le nom de la fonction, soit sur la même ligne, soit sur la ligne précédente.

     Les arguments de la liste devraient être séparés par une virgule suivie d'un espace pour mieux les dissocier.
   - les définitions des variables locales à la fonction, décalées d'une tabulation par rapport au début de la ligne.

- les instructions de la fonction en s'efforçant d'appliquer les règles d'indentation pour les itérations et les instructions de test.

Remarques :

- Il est important de mettre les définitions des types de données, les constantes symboliques et les définitions de macro-fonctions partagés par plusieurs modules dans un seul fichier, afin que les modifications puissent être faites dans ce seul fichier.
- Toute fonction doit être définie avant d'être utilisée dans une instruction :

```
void imprime(int a)
   {
     ...
     imprime(x) ;
     ...
   }
```

Si cela n'est pas possible (récursivité croisée par exemple) ou non souhaité, il est préférable de spécifier le type de son résultat et la définition de ses paramètres avant son appel en précisant par la clause **extern** que sa définition complète se trouve ailleurs : dans le même fichier ou dans un autre module.

```
extern void imprime(int a) ; /* spécification */
...
   {
     ...
     imprime(x) ;              /* appel */
     ...
   }

void imprime(int a)            /* définition */
   {
     ...
   }
```

Conventions d'écriture en C

Les commentaires

On peut classer les commentaires en 3 groupes :

- les commentaires de type bloc sur plusieurs lignes. Ce type de commentaire est principalement employé pour décrire un fichier source ou une fonction. Il pourrait par exemple débuter par un seul **/*** et se terminer par un seul ***/** sur la dernière ligne. Les lignes intermédiaires constituant alors le corps du commentaire. Chaque ligne pourrait commencer par un caractère ***** aligné verticalement sur ceux du début et de la fin afin de mieux faire ressortir le commentaire du texte des instructions.

```
/*
 * nom du module :  motpasse.c
 * version :  04
 * projet :  messagerie électronique
 *
 * date création :  21/03/83
```



```
 * date modification :  10/05/83
 *
 * Ce module a pour but de créer de nouvelles entrées
 * dans un fichier qui associe un nom d'utilisateur à
 * un mot de passe pour la consultation du courrier
 *
 * auteur :  un-tel
 */
```

- les commentaires sur une seule ligne encadrée de lignes vides pour introduire une séquence de code spécifique.

```
/* ce qui suit sert à la mise au point */
```

- les commentaires en vis-à-vis des déclarations ou des instructions. Ceux-ci doivent être courts et dans la mesure du possible être alignés verticalement sur le premier **/*** :

```
char c ;      /* caractère courant */
int i ;       /* index dans le tampon */

c = getchar() ;      /* lecture d'un caractère */
if (c == EOF)        /* test fin du fichier */
     ...
```

Les déclarations : constantes, macro-fonctions, types, variables

Une déclaration commence et termine sa ligne :

```
int x ; int y = 1 ;      /* incorrect */

int x ;                  /* correct */
int y = 1 ;
```

Le choix des identificateurs de variables et de fonctions est très important pour la lisibilité des programmes. Il convient de respecter les conventions en usage :

- **argc** et **argv** pour les arguments d'appel du programme récupérés en paramètre de la fonction **main()**,
- **i**, **j**, **k**, ... pour les indices de boucle,
- **p**, **q**, **p1**, ... pour les identificateurs de pointeurs,
- **s** pour une variable de type chaîne de caractères,
- **tmp** pour une variable temporaire, ...
- les variables doivent avoir un nom mnémonique, c'est-à-dire qui comprend une signification. Il faut supprimer les noms de variables du style **toto1**, **toto2**, **toto3**, ... comme les noms trop compliqués à retenir : **prmpt** qui pourrait vouloir signifier "premier point",
- les identificateurs comprenant plusieurs mots peuvent être rendus lisibles par le caractère _ ou par une majuscule sur les débuts des mots : **premier_point** ou **premierPoint**,
- ne vous limitez pas sur le nombre de caractères par identificateur,
- chaque définition de variable devrait être commentée,
- les variables débutant par le symbole souligné _ sont habituellement réservées au système,

- les constantes ou macro-fonctions définies par **#define** ainsi que les types définis par **typedef** devraient être en lettres majuscules :

```
#define DIM 120
typedef long ADRESSE ;
```

- les variables servant de masque ou de champ de bits devraient également être en majuscules :

```
#define BIT_PARITE 0x80
```

- les variables globales devraient avoir leur première lettre en majuscule et les autres en minuscules, comme pour un nom propre :

```
char Buffer[256] ;
```

- les variables locales devraient être en minuscules :

```
int i, j, index ;
```

- les membres des structures et des unions devraient commencer par les deux ou trois premiers caractères du nom de la structure ou de l'union, suivi d'un caractère _ et de l'identificateur du champ :

```
int DAT_mois ;
```

Le codage des instructions

Le principe général est d'isoler chaque élément structurel sur une ligne : instructions élémentaires, délimiteurs de blocs, ...

- une seule instruction élémentaire par ligne. Comme pour les déclarations, le point virgule d'une instruction termine sa ligne :

```
i = 0 ; j = 0 ;        /* incorrect */

i = 0 ;                /* correct */
j = 0 ;
```

- une accolade ouvrante termine sa ligne. L'accolade ouvrante du début d'un bloc peut être placée à la fin d'une ligne, ou seule sur la ligne suivante. Par exemple :

```
if (max < x) {
        max = x ;
        nb++ ;
}
```

ou

```
if (max < x)
  {
    max = x ;
    nb++ ;
  }
```

- une accolade fermante est seule sur sa ligne :

```
    }}          /* incorrect */

  }  /* correct */
}
```

- une étiquette est seule sur sa ligne :


```
switch (c) {
        case 'a' :  ajout() ;        /* incorrect */
                    break ;
        ...

switch (c) {
        case 'a' :
                    ajout() ;        /* correct */
                    break ;
        ...
```

Le codage des expressions

- les mots clés suivis d'une expression entre parenthèses devraient être séparés de la parenthèse ouvrante par un espace.

```
if (expr)
for (i=1 ; i < 10 ; i++)
switch (c) ;
```

- la liste des arguments d'une fonction mise entre parenthèses devrait suivre immédiatement le nom de la fonction.

```
printf(stderr, "erreur\n") ;
```

- les éléments d'une liste d'arguments devraient être séparés par une virgule suivie d'un espace.

```
main(argc, argv)
```

- il ne devrait pas y avoir d'espace entre les opérateurs primaires `() [] .->` et leurs opérandes.

```
getchar(c)
tab[i]
date.heure
ptr->nom
```

- il ne devrait pas y avoir d'espace entre un opérateur unaire et son opérande.

```
* p ++        /* incorrect */
*p++          /* correct */
```

- tous les opérateurs binaires devraient être entourés d'un espace de part et d'autre.

```
a=b*(c+d)          /* incorrect */
a = b * (c + d)    /* correct */
```

- les expressions d'un opérateur conditionnel devraient être parenthésées.

```
max = a > b a :  b          /* incorrect */
max = (a > b) ?  (a) :  (b) /* correct */
```

- le terminateur d'instruction ; devrait être précédé d'un espace.

Indentation du programme

La partie code du module doit mettre en évidence sa structure en blocs en utilisant l'**indentation** et permettre ainsi sa compréhension :

- l'indentation de chaque instruction dépend de son niveau de profondeur dans le programme. Les décalages doivent être constants.

```
while (!  arret)
      {
        ...
        if (x == y)
           {
             ...
           }
        else
           {
             ...
           }
        ...
      }
/* suite des instructions */
x += 3 ;
...
```

- si la ligne est trop longue, il faudrait la découper en plusieurs parties logiques, les lignes suivantes étant alors indentées par rapport à la première.

```
/* incorrect */
x=((y-1)+(6*((i/5)-1)))/((float)((k-1)*k)-((i-
  j)/6)) ;

/* correct (par exemple) */
x = (
      (y-1)
      +
      (6*((i/5)-1))
    )
    / ((float)((k-1)*k) - ((i-j)/6)) ;
```

- si une instruction `if` comporte deux blocs "alors" et "sinon" de tailles très inégales, il sera plus lisible de commencer par écrire la plus petite partie, quitte à inverser la condition.

```
if (x > 0)
   {
     /* calcul de la longueur */            /* incorrect */
     y = z / x ;
     ...
     printf("valeur de z :  %f\n",z) ;
   }
else
   printf("pas de solution\n") ;

if (!  (x > 0)) /* ou if (x <= 0) */        /* correct */
      printf("pas de solution\n") ;
else
   {
     /* calcul de la longueur */
```



```
     y = z / x ;
     ...
     printf("valeur de z :  %f\n",z) ;
   }
```

Quelques pièges classiques ..... à éviter

Test d'égalité

Le langage C se distingue des autres langages de programmation en utilisant le symbole `==` pour effectuer un test d'égalité au lieu du simple signe `=`. L'utilisation du symbole d'affectation `=` dans un test ne génère aucune erreur de syntaxe :

```
if (x = y)        /* incorrect */
          printf("bonjour\n") ;
```

L'expression `(x = y)` affecte à la variable x la valeur de la variable y puis teste si cette valeur est égale à 0 (signifiant "faux"). L'appel à la fonction `printf()` est effectué chaque fois que la valeur de `y` est différente de 0, avec en plus un effet de bord qui modifie la valeur de la variable `x`.

Excès nuisible des points virgules

Il est possible d'insérer des instructions vides dans le code ; par exemple, le fragment de code suivant recherche la fin d'une chaîne de caractères sur laquelle pointe la variable `pc` :

```
char *pc ;
...
while (*pc++) ;        /* correct */
```

Or, les compilateurs C ne détectent pas les instructions `if` inutiles du style :

```
if (x > max) ;        /* incorrect */
             max = x ;
```

qui sera exécuté comme suit :

```
if (x > max) ;
max = x ;
```

Même conclusion pour ce code qui n'en finira pas de boucler :

```
while ( i < 100 ) ;       /* incorrect */
      {
        tabb[i] = 0 ;
        i++ ;
}
```

Conversion automatique de type

L'itération suivante n'est jamais exécutée :

```
unsigned int i ;

for (i = 10 ; i > -1 ; i--)       /* incorrect */
    ...
```

Le compilateur convertit automatiquement la constante -1 en un entier non signé, tout en conservant sa représentation hexadécimale (`0xffff` sur 16 bits). Aucun entier non signé ne pourra être supérieur à cette valeur (`0xffff` correspond à la valeur 65536 sur 16 bits, soit le maximum des entiers non signés).

Définition de macro-fonctions

La définition :

```
#define carre(a) a * a /* incorrect */
```

semble calculer le carré de son argument. Toutefois, la ligne :

```
x = carre(a+b) ;
```

donnera après substitution :

```
x = a+b * a+b ;
```

Il convient alors d'enfermer l'argument entre parenthèses :

```
#define carre(a) (a) * (a) /* incorrect */
```

Les priorités des opérateurs peuvent provoquer d'autres erreurs ; par exemple, l'instruction :

```
x = (short) carre(a) ;
```

donnera après substitution :

```
x = (short)(a) * (a) ;
```

Il est alors préférable d'enfermer également l'ensemble de la définition de la macro-fonction entre parenthèses :

```
#define carre(a) ( (a) * (a) ) /* correct */
```

Les tableaux

La déclaration :

```
int tab[10] ;
```

permet de définir un tableau de 10 entiers, d'indices allant de 0 à 9. L'instruction d'initialisation du tableau à 0 :

```
for (i = 1 ; i <= 10 ; i++)
    tab[i] = 0 ;        /* incorrect */
```

ne sera pas détectée comme incorrecte alors que les indices d'accès aux élements du tableau sont faux.

Les chaînes de caractères

- La déclaration :

  ```
  char chaine[10] ;
  ```

  permet de définir un tableau de 10 caractères ; ce tableau pourra être manipulé comme une chaîne de caractères (par les fonctions `strcpy()`, `strlen()`, ...) si le caractère nul `'\0'` apparaît dans le tableau "après" les caractères utiles de la chaîne. Dans ce cas, n'oubliez pas de réserver une place dans le tableau pour ce caractère.

- La fonction `strlen()` donne la longueur "utile" de la chaîne de caractères qui lui est passée en argument, sans tenir compte du caractère nul `'\0'` qui la termine :

  ```
  char *r ;            /* incorrect */
  ...
  r = (char *) (malloc((unsigned int)(strlen(s1)))) ;
  strcpy(r,s1) ;
  ```

  L'appel à la fonction `malloc()` ne réserve pas assez de mémoire pour permettre de stocker le caractère nul (`'\0'`) de fin de chaîne. Il faudrait écrire :

  ```
  char *r ;            /* incorrect */
  ...
  r = (char *) (malloc((unsigned int)(strlen(s1) + 1 ))) ;
  strcpy(r,s1) ;
  ```

Les pointeurs

Les pointeurs représentent un outil puissant mais potentiellement très dangereux.

- Ne pas retourner en résultat d'une fonction un pointeur sur une zone mémoire dont le programme n'a plus le contrôle :

  ```
  int *f(int a)
     {
       int v[TAILLE] ;        /* incorrect */
       ...
       return(v) ;
     }
  ```

  Le tableau `v[]` est déclaré comme une variable locale de la fonction `f()` et sa classe de stockage est donc `auto` par défaut. La zone de mémoire correspondante est donc réservée (en général sur la pile) lors de l'entrée de la fonction `f()`, puis libérée dès la fin de la fonction, c'est-à-dire lors de l'exécution de l'instruction `return()`. Le contenu de `v[]` est donc perdu.

  Il suffit alors de déclarer le tableau `v[]` comme une variable statique :

  ```
  int *f(int a)
     {
       static int v[TAILLE] ;      /* correct */
       ...
       return(v) ;
     }
  ```

  L'espace mémoire alloué à `v[]` ne sera alors jamais utilisé par d'autres variables, bien sûr même après le retour de la fonction `f()`.

  Si on souhaite qu'une nouvelle zone mémoire soit créée à chaque appel de fonction, il est nécessaire d'effectuer une allocation dynamique par un appel à la fonction `malloc()` :

  ```
  int *f(int a)
     {
       int *v ;
       ...
       v = (int *) (malloc((unsigned)(TAILLE * sizeof(int)))) ;
       if (v != NULL)
          ...
     }
  ```

- Être vigilant lors de l'utilisation de fonctions prédéfinies demandant des pointeurs comme arguments. Dans la documentation du compilateur, on trouve la déclaration suivante pour la fonction **gets()** :

```
char *gets(char *buffer)
```

Il est cependant incorrect d'écrire le code :

```
char *p ;
gets(p) ;        /* incorrect */
```

car, d'une part, la variable **p** n'est pas initialisée avant l'appel à **gets()** et cela signifie qu'elle pointe sur une adresse quelconque en mémoire ; et d'autre part, aucun espace mémoire n'a été alloué à **p** pour que la fonction **gets()** y stocke les caractères saisis par l'utilisateur. Il faut écrire :

```
char *p ;        /* correct */

p = (char *)(malloc((unsigned)(TAILLE))) ;
if (p != NULL)
   gets(p) ;
```

ou

```
char p[TAILLE] ;
gets(p) ;
```

Allocation dynamique

Il est indispensable de tester systématiquement la valeur renvoyée par **malloc()** pour déterminer si la mémoire demandée a pu être allouée : dans le cas contraire, le pointeur obtenu est égal à **NULL**, son utilisation est illégale et il est préférable d'arrêter le programme (par **exit()**) plutôt que de continuer son exécution de manière incorrecte et/ou de le faire planter...

Pour écrire un programme "assez" portable

Utilisation de lint

L'utilitaire **lint** effectue de nombreuses vérifications syntaxiques et sémantiques dans le programme et montre les constructions non portables. La réciproque est cependant fausse : un programme ne générant aucun message au **lint**, n'est pas nécessairement portable...

lint permet donc d'aider à l'écriture de programmes portables, plus généralement, il aide au débuggage.

Contrôle des paramètres des fonctions

- Contrôler que les paramètres passés lors des appels des fonctions sont en accord avec la définition des paramètres dans les fonctions : même nombre, même ordre, même type. Il faut donner une valeur à tous les paramètres définis, même dans le cas où on souhaite en passer moins : on donne alors des valeurs quelconques aux paramètres inutiles, **NULL** par exemple.
- Pour créer, de façon portable, une fonction à nombre de paramètres variable, comme la fonction **printf()** par exemple, il faut utiliser les mécanismes déclarés dans le fichier **/usr/include/varargs.h**.

Appel du système par les fonctions de la bibliothèque

- Les fonctions la bibliothèque système Unix fournissent de nombreuses fonctions standards qu'il est recommandé d'utiliser afin d'isoler l'application écrite des modifications du système.

Ne pas dépendre du compilateur

Certains détails ne sont pas spécifiés dans la définition du langage. Ces détails provoquent des petites différences dans les implantations des différents compilateurs. Bien que "petites", ces différences sont source d'erreurs lors du transfert de programme d'une machine à une autre.

- Par exemple, l'ordre d'évaluation des arguments d'une fonction n'est pas précisé. Il est alors préférable de ne pas utilisé d'opérateurs unaires dans les paramètres d'appel des fonctions :

```
a = fonction(i++, t[i]) ; /* déconseillé */
```

- Ou encore, l'ordre d'évaluation des opérandes de la majorité des opérandes binaires, comme l'addition ou la multiplication, ne sont pas précisés. Ceci peut provoquer de redoutables effets de bords. Il est préférable de sur-parenthéser les expressions pour lever toute ambiguïté d'évaluation :

```
if ((i > 0) && (i < 10)) ...
a = ((b+c)*e)/f ;
```

Ne pas dépendre de la machine

Ne pas dépendre de la taille des mots de la machine

- La taille du type *int* dépend de la taille des mots de la machine et varie donc en fonction des machines. Dans le doute du résultat d'une opération sur des entiers, il est préférable d'utiliser le type *long* pour éviter d'éventuels problèmes de débordement.
- Il n'y a aucune garantie que la taille d'un *int* soit égale à la taille d'un mot de la machine. Il est seulement spécifié dans la définition du langage C que la taille d'un *short* est inférieure ou égale à la taille d'un *int*, elle-même inférieure ou égale à la taille d'un *long*.
- La taille des mots influe sur les constantes et peut provoquer des erreurs si la constante est par exemple utilisée comme un masque de bits :

```
#define MASK 0177770      /* incorrect */
int x ;
x &= MASK ;
```

Cette séquence d'instructions met à zéro les 3 bits de poids faibles (les plus à droite) de **x**, seulement dans le cas où la taille d'un *int* est de 16 bits. Dans le cas contraire, si la taille d'un *int* est supérieure à 16 bits, les bits de poids forts (les plus à gauche) de **x** sont également mis à zéro. Solution :

```
#define MASK ( 07)      /* correct */
int x ;
x &= MASK ;
```

Rappelons que l'opérateur unaire ~ donne le complément à 1 d'un entier, ~07 donne un entier où tous les bits sont à 1 sauf les trois bits de poids faibles.

Cet exemple fonctionne sur toutes les machines, quelle que soit la taille d'un *int*.

Attention aux opérations de décalage

- Le nombre maximal de bits pouvant être décalés à droite ou à gauche dépend de la machine. Une demande de décalage d'un nombre de bits supérieur à ce maximum provoque des erreurs.

- Il est préférable de transformer par **cast** les entiers en *unsigned* avant de les décaler. Sur certaines machines, le décalage est logique et les bits vacants sont mis à zéro. Sur d'autres machines, le décalage est arithmétique, les bits vacants (à gauche bien sûr) sont remplacés par le bit de signe. La définition du langage C garantit cependant le décalage logique pour les entiers *unsigned*.

Utilisation de `#define` pour les constantes

- C'est une "lamentable" pratique d'utiliser des constantes numériques, surtout quand leur signification n'est pas particulièrement évidente. Les constantes numériques sont beaucoup mieux représentées par des constantes symboliques qui leurs sont associées par **#define**. Cette méthode permet également de trouver plus facilement les valeurs des constantes, leur signification, si elles sont écrites à une place standard, comme au début du programme.

```
#define LARGEUR_ECRAN 80
```

  Ceci permet d'utiliser la constante **LARGEUR_ECRAN** au lieu de 80. C'est plus long à écrire... mais c'est plus rapide à comprendre, à modifier...

Utilisation de `sizeof()`

- Une constante est souvent utilisée pour représenter la taille d'un objet... ceci est une pratique non portable ! Il est préférable d'utiliser l'opérateur **sizeof()**.

```
#define NB_EL(T) (sizeof(T)/sizeof(*(T)))
```

  Ceci permet de définir le nombre d'éléments d'un tableau de façon portable.

Ne pas utiliser de constantes caractères

- Les constantes caractères sont réellement des objets de type *int* ; aussi la définition du langage C permet la création de constantes caractères multi-caractères. Cependant, l'ordre dans lequel les caractères sont rangés dans un mot dépend de la machine.

Ne pas dépendre de la représentation interne des entiers

- La plupart des machines utilisent une représentation des entiers en complément à 2, certaines cependant utilisent le complément à 1. De ce fait, il ne faut pas écrire des instructions code qui prennent en compte les avantages du complément à 2. Par exemple, le décalage de 1 bit à gauche d'un entier négatif dans le but de le multiplier par 2 est faux sur une machine en complément à 1.

Ne pas oublier que la représentation en virgule flottante et la précision varient d'une machine à l'autre

- La représentation interne des nombres réels dépend de la machine, les réels ont donc des précisions différentes. Il faut alors éviter, ou tout au moins souligner, les dépendances de certains formats utilisés.

Ne pas dépendre du nombre d'octets d'un mot, ni de l'ordre des octets dans le mot

- Le nombre d'octets d'un mot et l'ordre dans lequel ceux-ci sont stockés dans un mot sont fonction de la machine.

  Dans la fonction suivante, le caractère *nul* `'\0'` est écrit si l'octet de poids faible n'est pas l'octet le plus faible de l'adresse de la variable **c**.


```
#define SORTIE 1
putchar(c) ;      /* incorrect */
int c ;
   {
     write(SORTIE, (char *)&c, 1) ;
   }
```

  Dans cet exemple, la variable **c** devrait, bien sûr, être déclarée comme un *char* et dans ce cas la transformation par **cast** serait inutile.

Ne pas dépendre du nombre de bits dans un octet

- Le nombre de bits d'un octet varie d'une machine à l'autre... Comme preuve voici le contenu du fichier **/usr/include/values.h** :

```
#if gcos
#define BITSPERBYTE     9
#else
#define BITSPERBYTE     8
#endif
```

Faire attention lors de l'utilisation de caractères signés

- Sur certaines machines, les *char* sont signés, ils sont alors étendus avec leur signe lors de l'évaluation d'une expression. Selon les cas, il est préférable d'utiliser des variables *unsigned char* ou de forcer par **cast** le type des caractères en *unsigned char*, ou alors d'utiliser un type autre que *char*.
- Par exemple, il est dangereux qu'un indice puisse pointer avant le début d'un tableau sur une machine représentant les caractères signés :

```
#define TAILLE 256
char c ;
extern char table[TAILLE]
c = table[c] ;         /* incorrect */
```

  Pour interdire cette possibilité, il faut soit déclarer c de type *unsigned char*, soit forcer l'indice à être en *unsigned char* par **cast**.

- Cet autre fragment de programme ne permettra jamais de détecter la fin d'un fichier sur une machine qui ne représente que des caractères non signés :

```
#include <stdio.h>
char c ;
if ( (c = getchar()) != EOF)          /* incorrect */
```

  Sur les machines où les caractères sont non signées, la valeur de **c** ne sera jamais égale à **EOF** qui est défini par -1. La fonction **getchar()** de la bibliothèque rend un *int*, il faut alors déclarer **c** de type *int*.

Ne pas combiner des champs de bits séparés

- Il est possible de faire des champs de bits "portables" si on ne combine pas plusieurs champs séparés, car la taille maximale d'un champ de bit dépend de la taille d'un mot machine et les champs de bits ne sont pas cadrés sur les frontières des mots. De plus, l'ordre d'allocation des champs de bits dans un mot dépend de la machine : de la droite vers la gauche ou inversement.

- L'utilisation de champs de bits pour représenter des données externes n'est pas portable. Pour minimiser les problèmes, il est préférable de placer la structure des champs de bits en en-tête du fichier à lire et de souligner leur dépendance de la machine.

Utilisation de la clause cast pour les conversions de pointeurs

- Généralement, les conversions de pointeurs ne sont pas portables. Cependant, il est possible de convertir un pointeur en un type entier quelconque, si la taille de ce dernier est suffisamment grande pour contenir le pointeur, et de pouvoir retrouver ensuite le pointeur initial. De même, il est possible de transformer par **cast** un pointeur sur un objet en un pointeur sur un objet de plus petite taille, et de faire ensuite la transformation inverse.

Faire attention aux restrictions d'alignement lors des affectations de pointeurs

- L'exécution de la conversion d'un pointeur sur un certain type en un pointeur sur un autre type, peut provoquer un *"addressing exception"* si le pointeur modifié n'est plus "aligné". Ceci provient des contraintes d'"alignement" des mots de certaines machines (exemple : les mots de 16 bits doivent commencer à une adresse paire, alors que les octets peuvent commencer à toute adresse). Il faut alors utiliser la fonction **malloc()** qui rend un pointeur de type caractère sur une zone de stockage, laquelle est toujours correctement "alignée" ; ce pointeur peut alors pointer sur n'importe quel autre type de données.

Faire attention aux comparaisons entre pointeurs signés

- Certaines machines effectuent des comparaisons signées entre les pointeurs, d'autres font des comparaisons non signées... Cette différence ne pose pas de problèmes si on compare des pointeurs générés à partir d'adresses valides. Affecter -1 à un pointeur lui donne la plus grande valeur possible ou la valeur la plus petite de toutes les valeurs valides possibles.
- La seule constante "correcte" que l'on peut affecter à un pointeur est **NULL** (0), quand **NULL** est transformé par **cast** dans le type du pointeur.

Faire attention aux débordements de pointeurs

- Les pointeurs arithmétiques peuvent provoquer des débordements, par exemple dans le cas où un objet, comme un tableau, est proche du début ou de la fin de la mémoire. L'exemple suivant illustre le cas où un débordement peut arriver :

```
struct large x[TAILLE], *p ;        /* incorrect */
for (p = x[TAILLE - 1] ; p >= x ; p--)
```

Si le tableau **x** se trouve à une faible adresse mémoire, il est alors possible que **x-1** ne soit pas plus petit que **x**, mais plus grand, à cause du modulo. Par exemple, soit **x** à l'adresse **0x000006** ; l'adresse **x-1** (atteinte dans l'itération par la variable **p** lors de sa décrémentation) devient l'adresse de **x** à laquelle on soustrait la taille d'un objet pointé par **x** (par exemple 10 octets). Dans ce cas, le calcul **0x000006 - 0x00000a** donne **0xfffffc** qui est une adresse largement supérieure à l'adresse **x**. L'itération ne s'arrête jamais.

Ne pas dépendre d'un ensemble particulier de caractères

- Ne pas écrire d'instructions qui supposent qu'un ensemble de caractères possède des représentations successives et contiguës.


```
char c ;
if ((c >= 'a') && (c <= 'z'))      /* incorrect */
```

Ce test pour savoir si **c** est une lettre minuscule n'est pas portable. Il est préférable d'écrire :

```
char c ;
if (islower(c))      /* correct */
```

La fonction **islower()** de la bibliothèque a une définition qui dépend de la machine. Le test devient alors portable, puisque l'écriture de la fonction est, elle, spécifique à la machine.

- Ne pas prendre la différence entre deux lettres pour calculer leur écart dans un ensemble des caractères.

Ne pas utiliser d'astuces qui dépendent du matériel

- Chaque petite astuce permettant au programme d'aller plus vite et dépendant du matériel utilisé représente un important handicap pour la portabilité du programme.

Isoler les types qui dépendent de la machine

- La clause **typedef** permet de cacher des types qui dépendent de la machine. Lors de la modification d'un type, toutes les variables de ce type vont également changer. Le fichier **/usr/include/types.h** fournit un ensemble de types systèmes standards, par exemple :

```
typedef unsigned char ino ; /* numéro i-node */
```

Isoler les instructions qui dépendent de la machine

- Isoler les fonctions qui dépendent de la machine, les mettre chacune dans un fichier et regrouper éventuellement ces fichiers dans un répertoire séparé du reste de l'application.
- Entourer les sections d'instructions qui dépendent de la machine par des clauses de compilation conditionnelle testant le préprocesseur de la machine :

```
int *sommet_pile ;
#ifdef MACHINE1
   *--sommet_pile = donnee ; /* diminue */
#else
   *sommet_pile++ = donnee ; /* augmente */
#endif
```

- Les macro-fonctions utilisant **#define** peuvent être utilisées pour cacher les instructions qui dépendent de la machine :

```
#define BITSPERBYTE 8
#define BITS(TYPE) (sizeof(TYPE) * BITSPERBYTE)
```

La spécification de la constante appelée **BITSPERBYTE** est portable, l'implantation ne l'est pas. En revanche, la spécification et l'implantation de la macro-fonction **BITS** sont portables.

Portabilité des fichiers de données

- Les fichiers de données ne sont pas portables puisque les machines utilisent des représentations différentes des données. Malheureusement, il n'existe pas de solution simple pour transformer les fichiers de données. Si deux machines rangent les octets des mots dans un ordre différent, elles ne peuvent pas communiquer entre elles en se transférant des entiers octet par octet. De plus, les ensembles de caractères sont différents d'une machine à une autre.
- Une solution consiste à écrire des programmes spéciaux de conversion qui connaissent exactement les formats des données. Une autre approche consiste à écrire les octets des données dans un ordre prédéfini et indépendant des machines et à relire ces octets dans le même ordre. Il est alors préférable d'utiliser les fonction bibliothèques **`printf()`** et **`scanf()`** qui transmettent des caractères plutôt que les entrées-sorties binaires, sauf si cela n'est pas pratique.

Pour en savoir plus

Concepts fondamentaux de l'Informatique. A. Aho, J. Ullman. Dunod, 1993.

Lint: A C Program Checker. Unix System Programmer's Manual. Volume 2. S.C. Johnson. AT&T Bell Laboratories. Publié par Holt, Rinehart et Winston, New-York, 1983.

Unix Time-Sharing System: Portability of C Programs and the Unix System. S.C. Johnson, D.M. Ritchie. *The Bell System Technical Journal.* volume 57, n. 6, partie 2. AT&T Bell Laboratories, Murray Hill, NJ, 07974.

C Programming Guidelines. T. Plum. Plum-Hall, Cardiff, NJ, 1984.

Pour éviter les pièges du C. Revue PC EXPERT, pages 287–295, avril 1992.

C Traps and Pitfalls. Andrew Koenig, Addison Wesley Publishing Company, 1989.

Index

`!`, non logique booléen, 54
`!=`, différent, 54
`<`, inférieur, 54
`<=`, inférieur ou égal, 54
`<<`, décalage à gauche, 50
`>`, supérieur, 54
`>=`, supérieur ou égal, 54
`>>`, décalage à droite, 50
`\0`, 98
`\n, \t, \f, \b, ...` fonction `printf()`, 64
`|`, ou logique bit à bit, 50
`||`, ou logique booléen, 54
`()`, liste des paramètres d'une fonction, 91
`*`
  multiplication, 50
  "pointeur", 91, 113
`+`, addition, 50
`++`, incrémentation, 53
`+=, -=, *=, /=, ...`, affectation, 51
`,`, virgule, 78
`-`, soustraction, 50
`->`, champ d'une structure pointée, 87
`--`, décrémentation, 53
`.`, champ d'une structure, 87, 91
.c fichier de code C, 9, 15
.h fichier de déclaration, 9, 18, 147, 161, 170
.o fichier objet, 12, 15
`/`, division, 50
`:`, opérateur ternaire conditionnel
  `(?:)`, 73
`;`, terminaison d'instructions, 22, 71, 181
`=`, affectation, 51
`==`, égalité, 54, 181
`?:`, opérateur ternaire conditionnel, 73
`[ ]`, spécification de l'indice d'un tableau, 91
`#define`, 44, 157, 159, 182
`#else`, 157
`#endif`, 157, 160
`#if`, 157, 160
`#ifdef`, 157
`#ifndef`, 157
`#include`, 147, 157, 161, 170, 175
`#undef`, 157
`%`
  format d'affichage, 62
  opérateur modulo, 50
`&`
  et logique bit à bit, 50
  opérateur "adresse", 91, 112
`&&`, et logique booléen, 54
édition de liens, 13
`^`, ou exclusif (xor), 50
`/*` commentaire `/*`, 22
`{` bloc d'instructions `}`, 10, 22
`~`, complément à 1 (not), 50
addition, +, 50
adresse, 112
  d'un tableau, 112
  d'une variable, 112
  opérateur `&`, 91, 112
affectation
  `=`, 51
  `+=, -=, *=, /=, ...`, 51
aide en ligne, 92
allocation mémoire
  dynamique, 85, 128, 183
  statique, 97
  structure, 132
  tableau, 130
appel/déclaration d'une fonction, 146
`argc, argv`, 162, 167
argument
  `const`, 144
  commande *shell*, 162
  fonction, 135
  fonction passée en argument, 140
  liste variable d'arguments, 144
  `main()`, 21, 162, 164, 167
associativité des opérateurs, 57
`atof()`, `atoi()`, 106, 164
`auto` (classe), 35

bibliothèque, 169
bloc d'instructions `{ }`, 10, 22
booléen, 29
`break`, 81
`bus error`, 18

`calloc()`, 128
caractère
  format d'affichage, 62
  type de variable, 28
`cast`, 59
chaîne de caractères, 98
  format d'affichage, 62
  initialisation, 98, 120
  type de variable, 28, 182
champ
  d'union, 88
  de bits, 92
  de structure, 86
`char`, 28
classe de variable
  `auto`, 35
  `const`, 44
  `extern`, 40, 176
  `register`, 37
  `static`, 37, 42
  `volatile`, 43
commande *shell* `system()`, 168
commentaire entre `/*` et `*/`, 22, 176

comparaison ==, !=, <, ..., 54
compilation, 11, 15, 169
    conditionnelle, `#if, #else, ...`, 157
    directive, 44, 157, 175, 182
    options, 13, 15, 40, 44
complément à 1, ~, 50
`const`, 44, 136
constante, 44, 182
    chaîne de caractères, 28
    hexadécimale, 26
    octale, 26
constructeur
    de structure `struct`, 86
    de type `typedef`, 47, 87
`continue` *(utilisation interdite ...)*, 82
conversion de type, 59
`core dumped`, 18
`ctype.h`, 106

décalage gauche, <<, droite, >>, 50
déclaration d'une variable, 31, 47, 177
décrémentation pré-, post-, 53
définition
    d'un type, 47, 87
    d'une fonction, 157, 159
    d'une macro-constante, 44, 157, 182
    d'une macro-fonction, 182
`ddd`, 18
debugger, 18
`default`, 74
différent, !=, 54
directive de compilation, 157
    `#define`, 44, 182
    `#include`, 175
    conditionnelle, 160
division, /, 50
`do ... while`, 76
`double`, 28

écriture, 62, 63, 151, 152
égalité, ==, 54
Endian
    big, 28
    little, 28
entier, 26
    court, 26
    format d'affichage, 62
    long, 26
entrée-sortie, 62, 151, 152
`enum`, 45
énumération, 45
`EOF`, 155
et logique
    bit à bit, &, 50
    booléen, &&, 54
exécution d'un programme, 17
`execl()`, 172
`exit()`, 85
expression, 50
    évaluation, 55
    présentation, 179
`extern` (classe), 40, 176

`fclose()`, 155
`feof()`, 155
fermeture d'un fichier, 155
`fflush()`, 69
`fgetc()`, 152
`fgets()`, 152
fichier, 149, 151
    de code (.c), 9
    de déclaration (.h), 9, 18, 147, 161, 170
    fermeture, 155
    fin, `EOF`, 155
    ouverture, 149
`FILE`, 149
fin du fichier, 155
`float`, 28
fonction, 134, 175
    appel/déclaration, 146
    classification de caractères, 105
    conversion de chaînes de caractères, 106
    définition, 157, 159
    en argument d'une autre fonction, 140
    `extern`, 41
    gestion des fichiers, 149, 151
    macro-fonction, 182
    manipulation de chaînes, 99, 100, 105
    paramètre, 135
`fopen()`, 149
`for`, 78
`fork()`, 172
format d'affichage, `printf()`, 62
`fprintf()`, 152, 153
`fputc(), fputs()`, 152
`fread()`, 152, 154
`free()`, 133
`fscanf()`, 152, 153
`fseek()`, 151, 156
`fwrite()`, 151, 152, 154

`gcc`, 11, 13, 15, 40
`getc()`, 152
`getchar()`, 152, 155
`gets()`, 68
`gets(), getw()`, 152
`goto` *(utilisation interdite...)*, 83

hexadécimal
    constante, 26
    format d'affichage, 62

identificateur, 24, 31, 177
identification, 32
`if ... else`, 72
inclusion de fichier, 18, 147, 157, 161, 170, 175
incrémentation pré-, post-, 53
indentation d'un programme, 180
inférieur, <, inférieur ou égal, <=, 54

initialisation
    chaîne de caractères, 98, 120
    tableau, 97, 109
    variable, 31, 42
instruction, 71
    terminaison ;, 22, 71, 181
`int`, 26
`isalpha(), isdigit(), islower(),...`, 105

lecture, `scanf()`, 62, 66, 151, 152
libération de mémoire, `free()`, 133
lint, 184
`long`, 26
longueur d'une chaîne, `strlen()`, 99

mémoire
    allocation dynamique, 85, 128, 183
    allocation statique, 97
    libération, 133
macro-constante (`#define`), 44
macro-fonction (`#define`), 157, 182
`main()`, 21, 162, 164, 167
`malloc()`, 85, 128, 159
`malloc.h`, 128
`man`, 92
manuel C, 92
`memory violation`, 18
modulo, %, 50
mot réservé, 25
multiplication, *, 50

non logique
    bit à bit, ~, 50
    booléen, !, 54
nul \0 caractère de fin de chaîne, 98
`NULL`, 115, 149, 151

octal
    constante, 26
    format d'affichage, 62
opérateur
    addition, +, 50
    "adresse" &, 91, 112
    affectation, `+=, -=, *=, /=, ...`, 51
    affectation, =, 51
    arithmétique, 50
    associativité, 57
    booléen et &&, ou ||, non !, 54
    champ d'une structure, ., 87, 91
    champ d'une structure pointée, ->, 87, 91
    comparaison, 54
    complément à 1, ~, 50
    décalage gauche, <<, droite, >>, 50
    décrémentation, - -, 53
    différent, !=, 54
    division, /, 50
    égalité, ==, 54
    et logique bit à bit, &, 50
    gestion des adresses, 91
    incrémentation, ++, 53
    inférieur, <, inférieur ou égal, <=, 54
    modulo, %, 50
    multiplication, *, 50
    non logique, ~, 50
    ou exclusif, ^, 50
    ou logique bit à bit, |, 50
    "pointeur", *, 91, 113
    priorité, 57
    soustraction, -, 50
    supérieur, >, supérieur ou égal, >=, 54
    sur les pointeurs, 116
    ternaire conditionnel, ?:, 73
    virgule, ,, 78
options du compilateur, 13, 15, 18, 40, 44
ou
    exclusif, ^, 50
    logique bit à bit, |, 50
    logique booléen, ||, 54
ouverture d'un fichier, 149

paramètre
    `const`, 144
    fonction, 135
    fonction passée en paramètre, 140
    liste variable de paramètres, 144
    `main()`, 21, 162, 164, 167
pointeur, 113, 117, 121, 123, 183
    générique, 143
    tableau, 117
portable (programme portable), 174
pré-, post-
    décrémentation, 53
    incrémentation, 53
précompilation, 12, 44
préprocesseur C, 44, 157, 182
`printf()`, 62, 63
priorité des opérateurs, 57
processus, 172
`purify`, 18
`putc(), putchar()`, 152
`puts(), putw()`, 152

réel
    court, 28
    format d'affichage, 62
    long, 28
résultat
    d'un programme, `exit()`, 85
    d'une fonction, 138
`register` (classe), 37
`return`, 83

`scanf()`, 66
`segmentation fault`, 18
shell, 162
`short`, 26
`sizeof()`, 30, 154
soustraction, -, 50
`sprintf()`, 102
`sscanf()`, 102

`static` (classe), 37, 38, 42
status, 85
`stderr`, 156
`stdin`, 62, 69, 156
`stdio.h`, 62, 149, 156
`stdlib.h`, 107
`stdout`, 62, 156
`strcmp()`, `strncpy()`..., 99, 100, 105
`strcpy()`, `strlen()`..., 99, 100, 105
`string.h`, 100
`struct`, 86
structure, 123
    accès aux champs, 132
    allocation mémoire, 132
    d'un bloc, 10
    d'un fichier C, 18
    d'une application C, 9
    d'une fonction, 10
    type `struct`, 86
suffixe
    .c, 9, 15
    .h, 9, 18, 147, 161, 170
    .o, 12, 15
supérieur, >, supérieur ou égal, >=, 54
`switch ... case`, 74
`system()`, 168

tableau, 96, 123, 182
    allocation mémoire, 130
    de chaînes de caractères, 121
    dynamique, 110
    flexible, 110
    multidimensionnel, 107
    paramètre d'une fonction, 136
    statique, 110
taille d'une variable, 26, 30
terminaison d'instructions ;, 22, 71, 181
`toascii()`, `toupper()`, ..., 106
type
    d'une fonction, 134
    d'une variable, 26
    définition par `typedef`, 47, 87
    `enum`, 45
    `struct`, 86
`typedef`, 47, 87

`union`, 88
`unsigned`, 26

variable
    `auto`, 35
    classe, 34, 37, 38, 40, 42, 176
    `const`, 44
    déclaration, 31
    `extern`, 40, 176
    fichier, 149
    globale, 38, 40
    initialisation, 31, 42
    locale, 35, 37
    modifiable (`volatile`), 43
    nom, 24
    non modifiable (`const`), 44
    permanente, 37, 38, 40, 42
    `register`, 37
    `static`, 37, 38, 42
    temporaire, 35, 37, 42
    `volatile`, 43
`void *`, 143
`void`, 134
`volatile`, 43

`while`, 76

`xddd`, 18