

Notes sur le langage C

par Emmanuel Delahaye ([Espace personnel d'Emmanuel Delahaye](#))

Date de publication : 29 avril 2009

Dernière mise à jour : 07 septembre 2010

Ce document est un peu disparate. En effet, il contient une multitude d'articles plus ou moins longs qui sont souvent des réponses que j'ai pu apporter à des questions souvent posées sur les forums.

Je me suis efforcé ensuite de les organiser en chapitres de la façon suivante :

- Généralités
- Eléments du langage
- Fonctions
- Types et Variables
- Pointeurs, tableaux, chaînes
- Entrées / sorties
- Bibliothèque standard
- Codage
- Divers



*Votre avis et vos suggestions sur cet article
nous intéressent !*

Alors après votre lecture, n'hésitez pas :

I - C'est quoi l'algorithmique ?.....	5
II - Déclarations ? Définitions ?.....	6
II-A - Déclaration.....	6
II-B - Définition.....	6
III - Langage C ? Fonctions systèmes ? Je suis perdu !.....	8
III-A - Domaine couvert par le langage C.....	8
III-B - Fonctions système.....	9
III-C - Bibliothèques tierces publiques.....	10
III-D - Bibliothèques tierces privées.....	10
IV - C, UNIX, POSIX ?.....	11
V - Production du code exécutable.....	12
VI - pile, tas ? C'est quoi ?.....	13
VII - Le mode plein écran.....	14
VII-A - Installation de ansi.sys sous Windows XP.....	15
VIII - Se procurer la norme du langage C (C99).....	16
IX - C'est quoi ce 'static' ?.....	17
X - Le type retourné par main().....	18
X-A - Historique.....	18
XI - C'est quoi un prototype ?.....	19
XII - Bibliothèques de fonctions.....	21
XII-A - Bibliothèque statique.....	21
XII-B - Bibliothèque dynamique.....	21
XIII - Pourquoi ma fonction ne modifie pas ma variable ?.....	22
XIV - Les fonctions à nombre d'arguments variable.....	24
XIV-A - Introduction.....	24
XIV-B - Interface.....	24
XIV-C - Implémentation.....	24
XIV-D - Conclusion.....	25
XV - size_t, c'est quoi ?.....	26
XVI - Données.....	27
XVI-A - Initialisation.....	27
XVII - Structures.....	28
XVII-A - Structures visibles.....	28
XVII-B - Structures opaques.....	28
XVII-C - Pseudonymes (ou alias).....	30
XVIII - Variables globales.....	31
XVIII-A - Nommage.....	31
XVIII-B - Organisation.....	31
XVIII-C - Définition.....	31
XVIII-D - Déclaration.....	31
XVIII-E - Utilisation.....	32
XIX - Champs de bits.....	33
XX - Bien utiliser const.....	34
XX-A - Objets simples.....	34
XX-B - Pointeurs.....	34
XX-C - Usage.....	35
XXI - Les pointeurs démythifiés !.....	36
XXI-A - Introduction.....	36
XXI-B - Définition.....	36
XXI-B-1 - Pointeur sur objet.....	36
XXI-B-2 - Pointeur de fonction.....	37
XXI-B-2-a - Remarques importantes.....	38
XXI-C - Du bon usage des pointeurs.....	38
XXII - Les pointeurs, ça sert à quoi ?.....	39
XXII-A - A quoi ça sert?.....	39
XXII-A-1 - Petit rappel.....	39
XXII-A-2 - Comment faire ?.....	39
XXIII - Un tableau n'est pas un pointeur ! Vrai ou faux ?.....	41

XXIV - Passer un tableau à une fonction.....	42
XXIV-A - Introduction.....	42
XXIV-B - Tableau à une dimension.....	42
XXIV-C - Tableau à n dimensions.....	43
XXV - Retourner un tableau.....	44
XXVI - char* char**.....	45
XXVI-A - Introduction.....	45
XXVI-B - Chaîne de caractères.....	45
XXVI-C - Le type char *.....	45
XXVI-D - Le type char **.....	47
XXVII - Initialisation des tableaux de caractères.....	48
XXVIII - Qu'est-ce qu'une chaîne littérale ?.....	49
XXIX - Bien utiliser malloc().....	50
XXIX-A - Suppression du cast.....	50
XXIX-A-1 - Compilateur non ISO.....	50
XXIX-A-2 - Compilateur C++.....	50
XXIX-B - Déterminer la taille sans le type.....	51
XXX - Bien utiliser realloc().....	52
XXXI - Comment créer un tableau dynamique à 2 dimensions ?.....	53
XXXII - Saisie de données par un opérateur (stdin).....	54
XXXII-A - Introduction.....	54
XXXII-B - fgetc(), getc(), getchar().....	54
XXXII-C - gets().....	54
XXXII-D - scanf().....	54
XXXII-E - fgets().....	54
XXXII-F - Ressources.....	55
XXXII-G - Comment fonctionne fgetc(stdin) alias getchar().....	55
XXXII-G-1 - Comportement visible.....	55
XXXII-G-2 - Comportement interne.....	56
XXXII-G-3 - Quelques expérimentations.....	56
XXXIII - Les fichiers.....	58
XXXIII-A - Introduction.....	58
XXXIII-B - Texte ou binaire ?.....	58
XXXIII-B-1 - Fichier texte.....	58
XXXIII-B-2 - Fichier binaire.....	59
XXXIII-B-3 - Modes d'ouverture d'un fichier.....	59
XXXIII-B-4 - Lecture d'un fichier.....	59
XXXIII-B-4-a - fgetc(), getc().....	59
XXXIII-B-4-b - fread().....	59
XXXIII-B-4-c - fscanf().....	60
XXXIII-B-4-d - fgets().....	60
XXXIII-B-4-d-i - Exemple d'utilisation.....	60
XXXIII-B-4-d-ii - Explication.....	61
XXXIII-B-4-d-iii - Critique de cet exemple.....	61
XXXIII-B-4-d-iv - Détection d'une erreur.....	61
XXXIII-B-4-d-v - Gestion des fins de ligne.....	61
XXXIII-B-4-d-vi - Exemple amélioré avec détection de la fin de lecture.....	62
XXXIII-B-4-d-vii - Explication.....	63
XXXIII-B-5 - Écriture dans un fichier.....	63
XXXIII-B-5-a - fputc(), putc().....	64
XXXIII-B-5-b - fwrite().....	64
XXXIII-B-5-c - fprintf().....	64
XXXIII-B-5-d - fputs().....	64
XXXIII-B-6 - Bien utiliser les formats de données.....	64
XXXIII-B-6-a - Format orienté texte.....	64
XXXIII-B-6-b - Format orienté binaire.....	65
XXXIII-C - Supprimer un enregistrement dans un fichier binaire.....	65
XXXIII-D - En guise de conclusion.....	65

XXXIV - Pourquoi fflush (stdout) ?.....	66
XXXV - <time.h> : La gestion du temps.....	67
XXXV-A - Introduction.....	67
XXXV-B - Usage.....	67
XXXV-B-1 - Lire l'heure courante.....	67
XXXV-B-2 - Afficher l'heure courante.....	68
XXXVI - <time.h> : bien utiliser difftime().....	69
XXXVII - rand(), srand()... j'y comprends rien.....	70
XXXVIII - Du bon usage de qsort().....	72
XXXVIII-A - Description de la fonction qsort().....	72
XXXVIII-A-1 - Introduction.....	72
XXXVIII-A-2 - Interface.....	72
XXXVIII-A-2-a - Interface de la fonction de comparaison.....	72
XXXVIII-A-3 - Comportement.....	72
XXXVIII-A-3-a - Comportement de la fonction de comparaison.....	73
XXXVIII-B - Usage de la fonction qsort().....	73
XXXVIII-B-1 - Tri d'un tableau d'entiers.....	73
XXXVIII-B-2 - Tri d'un tableau de chaînes.....	74
XXXVIII-B-3 - Tri d'un tableau de structures.....	75
XXXVIII-B-4 - Tri de nombres réels.....	76
XXXIX - Les identificateurs réservés.....	78
XL - Bien gérer la portée des objets et des fonctions.....	79
XL-A - Fonctions.....	79
XL-B - Objets.....	79
XL-B-1 - Définition hors d'un bloc.....	80
XL-B-2 - Définition dans un bloc.....	80
XL-B-3 - Masquage (Shadowing).....	80
XLI - Du bon usage de assert().....	81
XLI-A - Exemple d'utilisation.....	81
XLII - Comportement indéfini.....	83
XLIII - Les item-lists.....	85
XLIII-A - Introduction.....	85
XLIII-B - Mise en oeuvre.....	85
XLIV - Borland C : "floating point formats not linked".....	91
XLV - Code standard ? Code portable ? Je suis perdu !.....	92
XLV-A - "standard" ?.....	92
XLV-B - "portable" ?.....	92
XLV-B-1 - portabilité absolue.....	92
XLV-B-2 - portabilité relative.....	92
XLV-C - Bon usage.....	93
XLVI - Enregistrer une structure.....	94
XLVI-A - Le format binaire.....	94
XLVI-B - Le format texte.....	95

I - C'est quoi l'algorithmique ?

C'est l'art de traduire un comportement en phrases simples et claires.

Surveiller la température.

Si elle dépasse la consigne, actionner une alarme.

Ces phrases sont ensuite traduites en 'pseudo-code' qui est une sorte de langage de description des comportements (ressemble au Pascal) à base d'actions et de structures de code comme IF-ELSE-ENDIF, SELECT-CASE, REPEAT-UNTIL, WHILE etc.

```
DO
  temperature := read_temperature()
  IF temperature > threshold
    alarm (ON)
  ENDIF
FOREVER
```

Ce pseudo-code est ensuite traduit facilement en langage d'implémentation (par exemple en C).

```
{
  for (;;)
  {
    int temperature = read_temperature();
    if (temperature > threshold)
    {
      alarm (ON);
    }
  }
}
```

 Pour plus d'informations, consultez les  **Cours & Tutoriels** sur l'algorithmique disponibles sur [Developpez.com](#)

II - Déclarations ? Définitions ?

Il y a beaucoup de confusions sur les termes définition et déclaration en C. Voici un petit article qui va s'efforcer de mettre les choses au point.

II-A - Déclaration

Une déclaration permet l'utilisation d'un objet ou d'une fonction. Elle doit être préalable à l'utilisation.

Exemples avec un objet x de type int :

```
extern int x;  
<...>  
{  
    x = 2;  
}
```

ou

```
int x;  
<...>  
{  
    x = 2;  
}
```

ou

```
{  
    int x;  
<...>  
    x = 2;  
}
```

Voici des exemples de déclarations de fonctions

```
int f();  
  
extern int g(void);  
  
static int h();  
  
static int i(int a)  
{  
}  
  
int j(char *b)  
{  
}
```

II-B - Définition

Une définition est l'endroit où l'objet ou la fonction sont réellement définis. De l'espace mémoire est alloué à l'objet, les instructions sont fournies à la fonction (entre des {}). Notons que par conséquent, une définition est aussi une déclaration implicite. Il convient donc d'être prudent sur l'emploi des termes. Voici des définitions d'objets :

```
int a;  
static float b[12];  
{  
    struct xxx c;
```

```
static long d;  
}
```

ou de fonction

```
int f()  
{  
}  
  
int g(void)  
{  
}
```

La dernière forme est une définition avec déclaration implicite sous forme de prototype

III - Langage C ? Fonctions systèmes ? Je suis perdu !

Sur un forum je pose une question sur le langage C et on me répond "va voir sur un forum consacré à ton système". M'enfin, je programme en C, c'est quoi ce cirque ?

III-A - Domaine couvert par le langage C

Le langage C, tel qu'il est défini par la norme, est un ensemble de règles d'écriture (syntaxe, sémantique) définissant les éléments du langage, et un ensemble de fonctions regroupées sous le terme générique de 'bibliothèque d'exécution du [langage] C'.

Les domaines couverts par la bibliothèque sont

- Les flux d'entrée/sorties
- Les traitements de chaînes
- Les conversions chaîne/binaires
- Les fonctions mathématiques
- Les algorithmes génériques (tri, recherche)
- La gestion du temps
- (d'autres qui me reviendront plus tard...)

On constate donc qu'un programme C standard permet d'entrer des données à partir de la ligne de commande (paramètres de main()) ou d'un flux entrant (stdin, fichier en lecture), de les traiter 'silencieusement', ou avec une trace vers un flux sortant (stdout, stderr ou un fichier en écriture) et de sortir des données vers un flux sortant selon le schéma bien connu.

```
+-----+ +-----+ +-----+
| entrée |-->| traitement |-->| sortie |
+-----+ +-----+ +-----+
```

Exemples typiques

- compilateur
- générateur de code
- convertisseur wav -> mp3
- etc.

Si on cherche d'autres domaines d'applications comme

- Gestion de l'écran en mode texte
- Lecture du clavier sans attente
- Impression
- Port série
- Réseau
- Ecran graphique
- Programmation événementielle
- Programmation multitâches
- Interface graphique
- Souris
- etc.

il va falloir utiliser des ressources externes au langage C. Bien que ces ressources disposent (entre autres), d'une interface leur permettant d'être appelées par un programme écrit en C, **elles ne font pas partie du langage C**.

On distingue principalement 3 types de ressources

- Les fonctions fournies par le système
- Les fonctions des bibliothèques publiques (gratuites, payante)
- Les fonctions des bibliothèques privées (personnelles, entreprises)

III-B - Fonctions système

Rappelons qu'un système est un logiciel lancé par la machine au démarrage et qui prend en charge la gestion des ressources matérielles, ainsi que la surveillance de différents événements. D'autre part, il fournit un certain nombre de fonctions utilisables dans les applications, ainsi que le moyen de charger et lancer (exécuter) une ou des applications.

L'ensemble des interfaces des fonctions système utilisables pour développer des applications est décrit dans un document appelé API (Application Programming Interface). Ce document précise pour chaque fonction :

- identificateur
- paramètres
- retour
- comportement

Pour des raisons propres à chaque architecture, l'interface est souvent matérialisée par un 'TRAP' ou une interruption logicielle que l'on ne peut appeler qu'en assembleur. Par exemple en sur un PC/x86, l'interruption BIOS 'Vidéo' :

```
MOV AH, 09h
MOV AL, character
MOV BH, 00h
MOV BL, attributes
MOV CX, 01h
INT 10h
```

ou avec des extensions de très bas niveau comme par exemple ceci en Borland C :

```
#if defined (__BORLANDC__)
#include <dos.h>
#else
#error Undefined for this platform
#endif

<...>

/* -----
VIDEO_putch()
-----
écriture d'un caractère à la position courante du curseur avec
la couleur spécifiée
-----
E : caractère (0-255)
E : couleur texte
E : couleur fond
S :
----- */
void VIDEO_putch (int c, eCOU ct, eCOU cf)
{
#if defined (__BORLANDC__)
union REGS reg;

/* caractère à écrire */
reg.h.al = (uchar) c;

/* putch avec attribut */
reg.h.ah = 0x09;

/* page 0 */
reg.h.bh = 0;

/* attributs 08h : clignotement */
```

```
reg.h.bl = (uchar) (ct | ((cf & ~0x08) << 4));

/* pas de repetitions */
reg.x.cx = 1;




/* Bios Video */
int86 (0x10, &reg, &reg);
#endif
}
```

Afin de faciliter l'appel à partir de différents langages de développement, chaque implémenteur de compilateur (C, C++, Pascal, Ada etc.) ou d'interpréteur (BASIC, Python, Ruby etc.) fournit une interface dans son langage. Les fonctions systèmes apparaissent donc comme une **extension** dudit langage.

Chaque système dispose de sa propre API. Mais certains systèmes offrent des API définies selon la norme **POSIX**, ce qui tend à normaliser au moins une partie des API.

III-C - Bibliothèques tierces publiques

Il est aussi possible d'utiliser des fonctions fournies par des bibliothèques publiques gratuites ou payantes selon les besoins et les licences requises. Ces bibliothèques sont le plus souvent indépendantes de la plateforme. On peut citer (interface C)

-  **GTK+** (GUI)
-  **SDL** (Graphisme 2D simple)
- Fmod (Multimédia)
-  **MySQL** (SGBD)
- etc.



III-D - Bibliothèques tierces privées

Chaque développeur peut, pour lui-même, ou au sein de son entreprise, développer une bibliothèque de fonctions 'métier' qui lui facilitent la réalisation d'applications spécialisées. Il est courant que ces bibliothèques soient partagées dans l'entreprise.

Certaines de ces bibliothèques peuvent ensuite devenir publiques si elles peuvent intéresser d'autres personnes et si la licence le permet.

IV - C, UNIX, POSIX ?


Le langage C n'a à voir ni avec UNIX ni avec POSIX.

UNIX est une norme de spécification de système (actuellement UNIX 03) gérée par l' **Open Group** qui définit un système indépendamment de la machine sur lequel il tourne. Elle s'appuie, entre autre, sur la spécification des fonctions systèmes définie par POSIX.1 sous l'égide de l' **IEEE**.

Cette norme (POSIX.1) comprend à la fois la description d'un certain nombre de fonctions systèmes, et d'au moins deux types d'interfaces :

- Le mode commande (shell) pour l'utilisateur de la machine.
- Une API utile aux programmeurs d'applications.

L'API est décrite sous la forme d'interfaces de fonctions appelables directement en langage C (C99 pour les dernières specs de POSIX.1). Les fonctions standards du C sont reprises telles quelles par POSIX.1.

 *Ce sont ces nombreux emprunts au langage C qui ont créés la confusion entre langage C et POSIX ...*

Cette API est implémentée par une bibliothèque (.a, .so, .lib, .dll, ...) et des fichiers d'entête (.h) livrés avec les systèmes compatibles POSIX.1 (la plupart des unixoïdes, mais aussi certains Windows).

La spécification complète de POSIX.1 est disponible  [ici](#) (s'inscrire, c'est gratuit et sans danger).

Les constructeurs de systèmes et de compilateurs s'efforcent de suivre tout ou partie de ces normes, ce qui permet une portabilité importante dans des domaines qui ne sont pas couverts par la norme du langage C comme

- La gestion des répertoires
- La gestion des processus (*process*)
- La gestion des tâches (*threads*)
- La gestion des réseaux (*sockets*)
- etc.

V - Production du code exécutable

Il existe de nombreuses implémentations du langage C. Certaines sont des interpréteurs, mais la plupart sont des compilateurs.

Compiler un programme consiste à vérifier le code source, puis à le traduire en langage machine de façon à en faire un fichier exécutable qui pourra ensuite être exécuté par la machine.

Les détails de production du code dépendent de l'implémentation, c'est à dire de la machine, du système, des outils utilisés etc. Cependant, il existe une procédure générale commune à toutes ces implémentations :

- Production de modules de code machine intermédiaires par compilation individuelle des codes sources. Il manque les références externes.
- Production du code machine exécutable par résolution des références externes (liens) entre les modules intermédiaires, et l'éventuelle ajout de bibliothèques de fonctions selon les besoins.

Pour cela, on utilise successivement 2 outils :

- Le compilateur (*compiler*), autant de fois que nécessaire, selon le nombre de fichiers sources à traiter.
- L'éditeur de lien (*linker*), une fois pour produire l'exécutable.

VI - pile, tas ? C'est quoi ?

Le langage C définit 3 zones de stockage pour les objets (aussi appelés variables)

- La mémoire statique
- La mémoire allouée
- La mémoire automatique

1 - Les objets définis hors des fonctions et les objets définis avec le mot clé `static` sont placés en mémoire statique. Leur durée de vie est celle de l'exécution du programme. Ils existent et sont initialisés (à 0 par défaut) avant même le lancement de `main()`.

2 - Les objets définis avec les fonctions `malloc()`, `calloc()` et `realloc()` sont placés en mémoire allouée (appelée *heap* ou *tas* sur certaines implémentations). Leur durée de vie est contrôlée par le programme. Ils existent dès que l'appel de `*alloc()` retourne une valeur différente de `NULL` et cessent d'exister dès que `free()` est appelé avec la valeur retournée par `*alloc()`.

3 - Les objets définis dans un bloc, sans qualificateur `'static'`, sont placés en mémoire automatique (appelée *stack* ou *pile* sur certaines implémentations). Leur durée de vie est celle du bloc dans lequel ils sont définis.

VII - Le mode plein écran

En C standard, la notion d'écran (et plus généralement de matériel) n'existe pas. Une application écrite en C est censée tourner sur une machine abstraite dont les interfaces matérielles sont vues comme des flux d'entrée ou de sortie. Ceci dans un souci de portabilité, qui est une des raisons d'être du langage C.

Parmi ces flux, il en existe 3 qui sont ouverts par défaut :

- `stdin` : entrée standard
- `stdout` : sortie standard
- `stderr` : sortie erreur

Une application écrite en C peut être recompilée pour une multitude de plateformes (qui diffèrent selon le processeur, l'architectures, le système, etc.) pour lesquelles `stdin`, `stdout` et `stderr` seront implémentés de manières différentes.

Sur un modem, par exemple, ces flux seront connectés à un port série permettant de brancher une console par laquelle on pourra passer, par exemple, des commandes AT.

Le but est d'instaurer un dialogue 'interactif' dans lequel l'application montre qu'elle attend une commande de l'opérateur à l'aide d'un 'prompt' ou 'invite de commande':

```
modem> _
```

L'utilisateur peut alors passer une commande de numérotation

```
modem> ATD0123456789  
_
```

et le modem répond

```
CONNECTED AT 9600 BAUDS
```

il passe alors en mode transmission de données, et il signale qu'il n'attend plus de commandes par l'absence de prompt.

De même, sur un PC sous Windows ou unixoïde, il est possible d'ouvrir une console localement ou de se brancher localement ou à distance sur la machine via une console (gestion, application, debug...). L'application ne verra pas la différence et continuera à attendre ses commandes de `stdin` et à envoyer ses réponses à `stdout` et/ou `stderr`.

L'avantage d'un tel système est qu'il exige très peu de la console, qui peut se réduire à un simple terminal série, voire à une imprimante série s'il ne s'agit que de traces sur `stdout` ou `stderr` (utile pour mettre au point une application graphique, par exemple).


La gestion plein écran est une autre façon de concevoir la relation entre l'homme et la machine (IHM). En effet, on peut reprocher à l'interface console 'interactive', un certain manque de convivialité (largement compensé par le développement de langages 'scripts' extrêmement puissants), rendant les applications difficiles d'accès à un public non informaticien (direction, comptables, secrétaires, achats, commerciaux etc.)


C'est pour cela qu'il a été développé la notion d'interface 'plein écran' qui permet d'améliorer l'ergonomie.


- Effacement de l'écran
- Placement absolu du curseur
- Contrôle de la couleur du texte et du fond

Malheureusement, il n'a jamais été prévu de norme internationale régissant cette gestion d'écran (probablement parce que faisant trop partie du domaine applicatif, chaque constructeur voulant tirer parti des performances de son matériel et de son logiciel).


Certaines pratiques et normes de fait se sont toutefois imposées.

- Basées sur les ressources de la machine :
 - PC/MS-DOS : Borland et sa bibliothèque conio (Turbo Pascal, C) pour la console intégrée. Windows bénéficie d'un portage réduit de conio pour MinGW ( [voir la documentation](#)).
 - Unixoides : curses et ncurses. (consoles locales)
- Basées sur des ressources console
 - Commandes ANSI (VT-100)
 - Vidéotex (Minitel)
 - HTML (Browser)

Je recommande la bibliothèque  **PDCurses** qui est une version portable et multi-plateforme de [n]curses et qui permet facilement de gérer les entrées/sorties directes à la console, et ce soit d'une manière basique à-la-conio, ou d'une manière plus avancée. Il existe de nombreux tutoriels sur le net.

Mention spéciale pour 'termcaps' qui est une bibliothèque multi-plateforme censée s'adapter à quasiment tous les terminaux supportant des séquences d'échappement. Exemple d'implémentation :  **vv_termcaps**


VII-A - Installation de ansi.sys sous Windows XP


 *A ma connaissance, il n'y a pas de possibilité d'installer un interpréteur de séquences ANSI pour cmd.exe.*


VIII - Se procurer la norme du langage C (C99)

La norme du C est définie par l'ISO. On trouve l'original sur le site de l'ISO, mais elle est très chère. Elle est aussi reprise par l'ANSI à l'identique, pour la somme raisonnable de 20 USD.

Il y a aussi le site de **Dinkumware** qui fournit un excellent résumé des fonctions (C99).

Il est possible de télécharger gratuitement le dernier draft de la norme (N1256) ici : ( **C99**). Ce document est complet et intègre C99, TC1 et TC2 (Technical Corrigendum 1 et 2). C'est probablement le dernier draft du document définitif de l'ultime norme du C qui ne devrait plus évoluer, le comité étant en sommeil.

En fait, le comité continue de travailler, et une nouvelle version du langage C (C1x) est en cours d'élaboration. En voici le dernier draft : ( **n1336**).

Il existe un document quasiment officiel qui commente la norme : ( **Le 'Rationale'**). Très utile pour trouver des explications complémentaires.

IX - C'est quoi ce 'static' ?

static est un qualificateur qui a plusieurs significations selon le contexte.

- Devant une déclaration de fonction :

```
static int f(void)
```

Limite la portée de la fonction à l'unité de compilation courante.

- Devant une variable définie hors de tout bloc (dite 'de portée globale')

```
static int x;
```

Limite la portée de la variable à l'unité de compilation courante.

- Devant une variable définie dans un bloc (dite 'de portée locale')

```
int counter(void)
{
    static int x;
    x++;
    return x;
}
```

Place la variable dans la mémoire statique, la rendant persistante. Usage rarissime (*quick'n dirty*).



A éviter, surtout pour du code réutilisable (bibliothèque).

X - Le type retourné par main()

Bien que main() soit censé retourner un int, on voit quelquefois écrit

```
void main (void)
{
}
```

Qu'en est-il exactement ?

D'après la définition du langage C, dans un programme conforme, main() doit retourner int. D'ailleurs un compilateur comme Borland C 3.1 en mode ANSI refuse void main() (*error*). Dans les mêmes conditions, gcc qui émet un avertissement (*warning*).


X-A - Historique

Dès l'apparition du langage C, une des formes canoniques de main() était

```
main()
{
    return 0;
}
```

l'autre étant la forme qui permet de récupérer les arguments de la ligne de commande.

Il faut bien comprendre qu'à cette époque, une fonction définie sans type de retour explicite, retournait un int (c'est toujours le cas en C90, mais plus en C99 où le type doit être explicite). Le mot clé 'void' n'existait pas.

 *Il n'y avait donc aucune raison d'utiliser une forme void main().*

Ensuite, est venue la normalisation du langage C. (1989 ANSI, 1990 ISO). Dans le texte, les deux formes canoniques sont décrites :

```
int main (void)
```

et

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

Il est précisé en remarque (dans la partie **non normative**) qu'il existe d'autres formes sans autres précisions. Elles ne font donc pas partie de la norme, leur comportement est donc indéfini dans le cadre d'un programme respectueux de la norme (dit 'strictement conforme').

XI - C'est quoi un prototype ?

Lorsqu'on définit une fonction,

```
int f (char *s)
{
    ...
}
```

on définit en même temps son interface, à savoir :

- son nom (ici , f)
- son type de retour (ici, int)
- ses paramètres (ici, s, de type pointeur sur char)

Cet interface est aussi appelé prototype intégré. Il permet une utilisation de la fonction si l'appel est placé **après** la définition (il n'y a pas de raison de faire autrement, sauf cas exceptionnels et souvent douteux...) :

```
int f (char *s)
{
    ...
}

int main (void)
{
    int x = f("hello");
}
```

Maintenant, dans le cas de compilation séparée, on doit 'détacher' le prototype et le placer dans un fichier qui sera inclus à la fois dans le fichier d'implémentation de la fonction et dans le ou les fichiers qui l'utilisent.

```
/* f.h */
int f (char *s);
```

```
/* f.c */
#include "f.h"
int f (char *s)
{
    ...
}
```

```
/* main.c */
#include "f.h"

int main (void)
{
    int x = f("hello");
}
```

Pour compléter, il est fortement recommandé de protéger les fichiers .h contre les inclusions multiples avec une garde (*guard*) :


```
#ifndef H_F
#define H_F
/* f.h */
```

```
int f (char *s);  
  
#endif
```

Détails dans cet article  [ici](#)

XII - Bibliothèques de fonctions

Une bibliothèque (*library*) est une collection de fonctions mise à la disposition des programmeurs. Elle se compose d'une interface matérialisée par un ou plusieurs fichiers d'entêtes (.h), et d'un fichier d'implémentation qui contient le corps des fonctions (.lib, .a, .dll, .so etc.) sous forme exécutable.

Il est aussi possible à un programmeur de 'capitaliser' son travail en réalisant des fonctions réutilisables qu'il peut ensuite organiser en bibliothèques. Cette pratique est courante et encouragée. La création physique d'une bibliothèque est assez simple si on respecte quelques règles de conception, comme l'absence de globales, la souplesse et l'autonomie (voir l'article  **Comment construire sa bibliothèque**). Elle nécessite un outil spécialisé (librarian) généralement livré avec toute implémentation du langage C.

Une bibliothèque peut être statique ou dynamique (partagée).

XII-A - Bibliothèque statique

Une bibliothèque à édition de lien statique (.lib, .a etc.) est liée à l'application pour ne former qu'un seul exécutable. La taille de celui-ci peut être importante, mais il a l'avantage d'être autonome. Cette pratique a l'avantage de la simplicité, et elle ne requiert aucune action particulière de la part d'un éventuel système lors de l'exécution du programme. Elle est très utilisée en programmation embarquée (*embedded*).

XII-B - Bibliothèque dynamique

Une bibliothèque à édition de lien dynamique, est un fichier séparé (.dll, .so, ...) qui doit être livré avec l'exécutable. L'intérêt est que plusieurs applications peuvent se partager la même bibliothèque, ce qui est intéressant, surtout si sa taille est importante. Dans ce cas, les exécutables sont plus petits. Autre avantage, les défauts de la bibliothèque peuvent être corrigés indépendamment des applications. Ensuite, la correction est répercutée immédiatement sur toutes les applications concernées par simple mise à jour de la bibliothèque dynamique.

Une application qui utilise une bibliothèque dynamique doit réaliser le lien avec la bibliothèque à l'exécution. Pour cela, elle utilise des appels à des fonctions système spécifiques dans sa phase d'initialisation. Les détails dépendent de la plate-forme.

XIII - Pourquoi ma fonction ne modifie pas ma variable ?

Il y a deux façon de modifier la valeur d'un objet (aka variable) avec une fonction :

- Récupérer la valeur retournée par la fonction :

```
int x = f();
```

ou

```
int x;  
x = f();
```

avec

```
int f(void);
```

- Passer l'adresse de la variable dont on veut modifier la valeur à la fonction :

```
int x;  
f(&x);
```

avec

```
void f(int *);
```

Si l'objet est un pointeur, la règle est la même :

- Retourner une valeur :

```
int *px = f();
```

avec

```
int *f(void);
```

- Passer l'adresse :

```
int *px;  
f(&px);
```

avec

```
void f(int **);
```



Il n'y a aucune chance de modifier l'entier en faisant ceci :

```
int x;  
f(x);
```



De même, il n'y a aucune chance de modifier le pointeur en faisant cela :

```
int *px;
```

```
f(px) ;
```

XIV - Les fonctions à nombre d'arguments variable

XIV-A - Introduction

Bien que leur usage ne soit pas recommandé, car il n'y a pas ou peu de contrôles possibles par le compilateur, le langage C supporte les fonctions à nombre variable d'arguments (variadic).

Attention ! Non seulement le nombre est variable, mais aussi le type.

Un exemple bien connu est `printf()`.

```
printf ("s:%d\n", __FILE__, __LINE__);
```

XIV-B - Interface

- La fonction admet n'importe quel type de retour.
- Elle a obligatoirement un paramètre fixe
- Les paramètres variables marqués par l'ellipse '...', sont obligatoirement les derniers.

Ces prototypes sont valides :

```
void f(int a, ...);  
int f(int a, char *b, ...);  
/* etc. */
```

Le rôle du dernier paramètre fixe est de fournir un moyen de déterminer le nombre et éventuellement le type des paramètres variables (donc passés après).

Exemple :

```
void f (int n, ...);  
  
<...>  
{  
    f(3, 'a', 'b', 'c');  
}
```

Ici, le dernier paramètre fixe indique le nombre de caractères passés.

Un effort particulier doit être fait au niveau de la documentation de la fonction, car il n'y a pas de contrôle possible de la part du compilateur. Si on se trompe de type ou de nombre, c'est le drame...

XIV-C - Implémentation

Elle varie évidemment selon la fonction à réaliser, mais elle s'appuie sur quelques principes communs :

- Inclure `<stdarg.h>` pour accéder aux fonctions et macros de manipulation des paramètres variables
- Définir une variable de traitement des variadics de type `va_list`
- Initialiser cette variable correctement avec `va_start()` et le nom du dernier paramètre fixe
- Récupérer et traiter les paramètres selon la spécification de la fonction
- Terminer l'analyse proprement avec `va_end()`

```
#include <stdarg.h>  
  
void f (int n, ...)
```



```
{
    va_list va;
    va_start (va, n);

    /* traitement des paramètres selon la fonction */

    va_end (va);
}
```

La récupération des paramètres se fait séquentiellement de gauche à droite par appels successifs à `va_arg()` en précisant le type attendu, parmi 'int', 'long', 'double', 'void*' et 'char*' [C90].

```
int x = va_arg (va, int);
double y = va_arg (va, double);
char *z = va_arg (va, char *);
/* etc. */
```

Il doit y avoir correspondance parfaite entre le type de la variable et celui passé en paramètre à la macro `va_arg()`.

Là encore, la vigilance est de rigueur, car aucun contrôle n'est possible de la part du compilateur.

Exemple d'implémentation de la fonction présentée au-dessus.

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

void f (int n, ...)
{
    va_list va;
    va_start (va, n);

    int i;

    for (i = 0; i < n; i++)
    {
        int c = va_arg (va, int);
        putchar (c);
    }
    putchar ('\n');
    va_end (va);
}

int main (void)
{
    f (3, 'a', 'b', 'c');
    f (4, 'x', 'y', 'z', 't');

    return 0;
}
```

```
abc
xyzt
```

```
Process returned 0 (0x0)   execution time : 0.025 s
Press any key to continue.
```

XIV-D - Conclusion

Nous avons jeté les bases de la construction des fonctions variadics. Attention, ces fonctions sont une source importante de problèmes invisibles aux yeux du compilateur.

Ne pas les utiliser dans des projets critiques sans une procédure de validation de l'implémentation et de l'usage extrêmement rigoureuse. Dans certains domaines sensibles (médical, avionique, nucléaire), leur usage est tout simplement interdit.

XV - size_t, c'est quoi ?

size_t est le type retourné par l'opérateur sizeof. C'est un entier non signé. Il est suffisamment grand pour contenir la valeur représentant, en nombre de bytes (ou char), la taille du plus grand objet possible d'une implémentation donnée.

Il convient pour les tailles, les dimensions de tableau, les index croissants et non négatifs...

Ce type est défini dans <stddef.h> qui est inclus dans la plupart des headers standards courants (<stdio.h>, <stdlib.h> <string.h> etc.)

XVI - Données

Le langage C utilise 3 zones mémoire pour implémenter les données.

- La mémoire statique, qui contient les variables permanentes (modifiables ou non)
- La mémoire automatique qui contient les variables locales et les paramètres des fonctions
- La mémoire allouée qui contient des variables dynamiques gérées à l'exécution par le programme (malloc() / free()).

Les données en C sont caractérisées par :

- leur portée
- leur durée de vie.

La portée peut être :

- locale à un bloc
- limitée à une unité de compilation
- illimitée
- contrôlée par l'application.

La durée de vie peut être :

- limitée à un bloc
- permanente
- contrôlée par l'application.

Exemples:

```
/* permanente de portee illimitée */
int a;

/* permanente de portee limitée à l'unité de compilation */
static int b;

<...>
{
/* locale (de portee limitée au bloc) */
int c;

/* permanente de portee limitée au bloc */
static int d;

/* contrôlée de portee limitée à la validité du pointeur (non NULL) */
int *p = malloc (sizeof *p * 3);

free (p), p = NULL;
}
```

XVI-A - Initialisation

Seules les données statiques sont initialisées avant le lancement de main(). Les autres ont une valeur indéterminée. Il est donc nécessaire de les initialiser avant utilisation (lecture). Sauf indication explicite contraire, l'initialisation par défaut des données statiques est 0.

XVII - Structures

XVII-A - Structures visibles


On appelle structure visible une structure dont les éléments sont visibles de l'utilisateur.

Une 'définition de structure' est le moyen par lequel le programmeur indique au compilateur comment est constitué une structure. Cette opération ne réserve aucune mémoire.

```
struct mastructure
{
    type_1 element_a;
    type_2 element_b;
};
```

Ensuite, une structure peut être instanciée, c'est à dire qu'une instance de cette structure est définie en mémoire.

```
struct mastructure mastructure;
```

 *il est autorisé d'utiliser le même nom, même si ce n'est probablement pas le meilleur des choix possible.*

Ces informations suffisent à définir et instancier n'importe quelle structure 'visible'.

XVII-B - Structures opaques

On appelle structure opaque une structure dont les éléments ne sont pas visibles de l'utilisateur.

Pour cela, on utilise une définition réduite (ou incomplète) qui consiste à définir le nom de la structure sans en préciser le contenu.

```
struct mastructure;
```

Cette définition dite incomplète ne permet évidemment pas de créer une instanciation, puisque le compilateur ignore le contenu de la structure. Il n'a donc pas les moyens d'en déterminer la taille.

Par contre, il est possible de créer un pointeur de ce type :

```
struct mastructure *p;
```

Il devient alors possible de créer une fonction qui retourne un pointeur de ce type :

```
struct mastructure *fonction(void);
```

de passer ce pointeur en paramètre une fonction :

```
void fonction(struct mastructure *p);
```

d'en faire un élément de structure etc.

Evidemment, il faudra que la structure soit définie 'quelque part' afin qu'elle soit instanciable et que ses éléments soient manipulables.

On va donc créer un fichier source (.c) séparé d'implémentation contenant les fonctions permettant la création (instanciation) des données, et une interface (header ou .h) ne comportant que la définition incomplète de la fonction et, au minimum, les 2 fonctions permettant la création et la suppression d'une instance de la structure.

Soit la structure 'xxx'. On obtient :

```
/* xxx.h Interface */
#ifndef H_XXX
#define H_XXX

/* definition incomplete de structure */
struct xxx;

/* prototypes des fonctions */
struct xxx *xxx_create (void);
void xxx_delete (struct xxx *p);
/* etc. */

#endif
```

```
/* xxx.c Implementation */
#include "xxx.h"

/* definition de la structure (exemple) */
struct xxx
{
    int a;
    char b[10];
};

/* fonctions publiques */
struct xxx *xxx_create (void)
{
    /* a completer */
}

void xxx_delete (struct xxx *p)
{
    /* a completer */
}
```

Exemple d'utilisation :

```
/* test.c */
#include "xxx.h"

#include <stddef.h>

int main (void)
{
    /* instanciation */
    struct xxx *p = xxx_create ();

    /* tout appel de fonction peut echouer... */
    if (p != NULL)
    {
        /* utilisation de l'objet xxx
         (via de nouvelles fonctions a creer)

        ...
        */

        /* fin d'utilisation : destruction de l'objet */
        xxx_delete (p), p = NULL;
    }
    return 0;
}
```

Je laisse au lecteur le soin de proposer une ou des implémentations de `xxx_create()` et de `xxx_delete()`, sachant qu'on a pas forcément besoin d'un nombre illimité d'instanciations...

XVII-C - Pseudonymes (ou alias)

Il est possible, afin de simplifier l'écriture (notamment pour les interfaces publiques), de remplacer le nom de la structure par un nom différent (alias ou pseudonyme) généralement plus court. Il est recommandé de ne pas abuser de l'abstraction, car les possibilités sont réduites en C et il est bon que le programmeur garde en tête qu'il manipule des pointeurs.

Ceci est possible :

```
typedef struct xxx xxx_s;
```

mais ceci est déconseillé :

```
typedef struct xxx *xxx; /* /\ \ */
```

Détails d'application dans l'article  **Les types abstraits de données (ADT)**

XVIII - Variables globales

Une variable globale est une variable définie en dehors d'une fonction et de portée globale. Sa durée de vie est égale à celle du programme. Elle est initialisée par défaut (0) ou explicitement avant l'exécution de `main()`. Sa valeur est persistante.

Un usage abusif des variables globales est fortement déconseillé pour diverses raisons :

- On ne sait ni comment ni quand ni qui accède à cette variable. Ca rend le code instable et incompréhensible. On n'a aucune certitude sur le code.
- Cela crée une dépendance, ce qui le code rend non modulaire et non réutilisable.
- L'instance étant unique, ça rend le code impropre à la récursion et à l'utilisation dans des threads et même au simple appel imbriqué.

Ceci dit, il est des cas rares (ou plus fréquents s'il s'agit de lecture seule) où les variables globales sont utiles, voire indispensables. Dans le cadre d'une application professionnelle, ces cas doivent être justifiés. Voici comment les définir correctement dans le cadre d'une application composée d'unités de compilations séparées.

XVIII-A - Nommage

Il est recommandé d'utiliser le préfixe `g_` ou `G_` pour signifier qu'une variable est globale.

XVIII-B - Organisation

Il est préférable pour éviter la dispersion, d'utiliser une ou des structures de variables globales regroupées par fonction, plutôt qu'une multitude de variables.

XVIII-C - Définition

Il est recommandé que la définition d'une variable globale soit faite exclusivement dans un fichier source (.c). Ce fichier doit inclure le fichier de déclaration (en-tête).

```
/* data.c */

#include "data.h"

int G_x;

/* la taille du tableau est definie
 * dans la declaration.
 */
double G_a[];

data_s G_data;
```

XVIII-D - Déclaration

Il est recommandé que la déclaration d'une variable globale soit faite exclusivement dans un fichier d'entête (.h). Ce fichier doit être inclus dans le fichier de définition et dans tous les fichiers d'utilisation (.c). Comme tous les fichiers d'en-têtes, celui-ci dispose de protections contre les inclusions multiples.

```
#ifndef H_DATA
#define H_DATA

/* data.h */
```

```
typedef struct
{
    int a;
    char b[123];
}
data_s;

extern int G_x;

/* la defintion de la taille du tableau est unique. Elle est faite ici. */
extern double G_a[12];

extern data_s G_data;

#endif /* guard */
```

XVIII-E - Utilisation

Il est recommandé que le fichier qui utilise une variable globale inclue le fichier de déclaration (.h).

```
/* appli.c */

#include "data.h"

int main (void)
{
    G_x = 123;

    G_data.a = 456;

    G_a[3] = 123.456;

    return 0;
}
```


XIX - Champs de bits

Afin de réduire la taille des objets, il est possible de définir un champ de bits. La définition doit se faire dans une structure. Le type de l'objet unitaire doit être `int` ou `unsigned int` (recommandé) ou `_Bool` (`bool`) en C99.

```
typedef struct
{
    unsigned a:1; /* variable de largeur 1 bit */
    unsigned b:3; /* variable de largeur 3 bits */

    /* etc. */
}
data_s;
```

Il faut garder à l'esprit que l'implantation mémoire des bits n'est pas spécifiée par le langage C. (Et j'ai effectivement constaté sur le terrain des différences selon les implémentations, notamment concernant l'ordre des bits).

Autant une utilisation interne est possible et peut se justifier pour réduire la taille des objets (stockage en mémoire, notamment), extrait de **CLIB** Module DATE (`date.h`) (Les tailles indiquées en commentaire sont les tailles minimales garanties)...

```
typedef unsigned int uint;
<...>
typedef struct
{
    /* 16 bits */
    int year; /* -32767..+32767 */
    uint month:4; /* 0-15 */
    uint day:5; /* 0-31 */

    uint hour:5; /* 0-31 */
    uint minute:6; /* 0-63 */
    uint second:6; /* 0-63 */
}
sDATE;
```

... autant il est illusoire d'utiliser les champs de bits pour créer une interface avec l'extérieur du programme, comme un flux de bytes ou un périphérique en accès direct (mémoire, bus I/O etc.).

Autre pratique non portable, faire une union entre un champ de bits et une variable en s'imaginant pouvoir accéder à la variable, soit d'un bloc, soit bit à bit.

La solution portable pour accéder aux bits d'une variable est d'utiliser les opérateurs binaires (`&`, `|`, `~`, `<<`, `>>`, `^`)

XX - Bien utiliser const

Voici à quoi sert le qualificateur const et comment l'utiliser correctement.

XX-A - Objets simples

Le mot clé 'const' est un qualificateur (qualifier) d'objet. Il lui fait perdre sa qualité par défaut qui est 'accessible en lecture ou en écriture' pour le modifier en 'accessible en lecture seule'. Par exemple :

```
int x = 3;
int const y = 4;

x = 5; /* acces en ecriture possible */
y = 6; /* acces en ecriture interdit */
```

Le compilateur signale l'erreur.

On peut placer indifféremment le qualificateur const avant ou après le type.

```
int const a = 7;
const int b = 8;
```

mais je conseille néanmoins la première forme, car elle est beaucoup plus claire (notamment avec les pointeurs).

Il est techniquement possible de définir un objet const non initialisé :

```
int const c;
```

évidemment, l'intérêt est limité, mais il a son application dans un contexte particulier : les paramètres de fonctions.

XX-B - Pointeurs

Le cas des pointeurs est un peu plus complexe, puis qu'il y a en quelque sorte 2 objets pour le prix d'un !

- Le pointeur lui même qui peut être qualifié const
- L'objet pointé qui peut lui même être qualifié const

Pour le pointeur, celui-ci étant un objet comme autre, la même règle s'applique, sachant que const doit être placé juste avant l'identificateur, c'est à dire après le dernier * :

```
int a;
int * const pa = &a;

pa++; /* interdit */
*pa = 123; /* OK */
```

Mais un autre qualificateur const peut être utilisé pour préciser les droits du pointeur sur l'objet pointé.


Celui ci se place à la gauche de l', avant ou après le type :

```
int a = 123;
int const * pa = &a;
const int * pb = &a;
```

mais là encore, pour des question de clarté du code, je recommande la première forme. Ce qualificateur interdit la modification de l'objet via le pointeur (mais s'il n'est pas lui même qualifié const, l'objet reste modifiable directement, évidemment).

```
int a = 123;
int const * pa = &a;

*pa = 456; /* interdit */
a = 456; /* OK */
```

 **Par contre, attention.** Il est techniquement possible de définir un pointeur sur un objet qualifié const et de tenter de modifier l'objet. Cela produit un comportement indéfini qui n'est pas forcément signalé par le compilateur.

```
int const a = 123;
int * pa = &a;

*pa = 456; /* comportement indéfini */
```


Cependant, le plus souvent, le compilateur signale un problème au moment de l'affectation du pointeur.

```
int * pa = &a;
```

mais il convient de rester extrêmement prudent. Le C est un langage qui demande rigueur et maîtrise.

 **Bien évidemment, le typecast n'est pas la solution :**

```
int const a = 123;
int * pa = (int *) &a; /* NE PAS FAIRE CECI */
```

 **il ne fait éventuellement que masquer le problème au compilateur ("je sais ce que fais"), mais il ne résout rien et le comportement indéfini est toujours là.**

XX-C - Usage

Le rôle du qualificateur const est particulièrement utile avec les pointeurs, notamment sur des chaînes de caractères, qui, rappelons le, ne sont pas modifiables. Il est fortement recommandé de définir tout pointeur sur une chaîne de caractères avec le qualificateur const :

```
char const *p = "hello";
```

Il est utile aussi pour les pointeurs passés en paramètre à des fonctions. Il permet en effet de restreindre l'accès à la variable pointée à un mode 'lecture seule', ce qui évite bien des erreurs de codage. (La modification d'une variable étant une opération lourde de conséquences si elle est faite au mauvais moment).

Une fonction qui affiche le contenu d'un tableau ou d'une structure, par exemple, n'a pas à la modifier. On fixe donc les règles du jeu dès la définition du prototype :

```
void display (T const *p)
```

Une utilisation astucieuse et intelligente du qualificateur const permet d'écrire du code plus sûr.

XXI - Les pointeurs démythifiés !

XXI-A - Introduction

Le langage C est indissociable de la notion de pointeur. Ce mot mythique en effraye plus d'un, et il est bon de démythifier enfin les pointeurs, sources de tant d'erreurs de codage, de discussions sans fin et de contre vérités...

XXI-B - Définition

Un pointeur est une variable dont la valeur est une adresse.

On distingue 2 grandes familles de pointeurs :

- les pointeurs sur objet
- les pointeurs de fonction

XXI-B-1 - Pointeur sur objet

En C, un objet est essentiellement une variable mémoire (par opposition à registre ou constante), modifiable ou non, mais munie d'une adresse.

Un pointeur sur objet (aussi appelé communément 'pointeur') est une variable qui peut contenir l'adresse d'un objet.

Pour définir un pointeur, on utilise un type, puis le signe '*' et enfin l'identificateur, suivi de ';' si on ne désire pas l'initialiser à la déclaration (peu recommandé). Sinon, on utilise l'opérateur '=', suivi de la valeur d'initialisation. Si le pointeur est déclaré dans un bloc, on peut utiliser la valeur retournée par une fonction.

```
/* Definition d'un pointeur sur int */  
int *p_int;
```

Pour initialiser un pointeur, on peut soit :

- lui donner la valeur 0 ou NULL, qui signifie "invalidé, ne pas utiliser".
- lui donner l'adresse d'une variable,
- lui donner la valeur retournée par les fonctions malloc(), realloc(), calloc().
- s'il est de type void ou FILE, lui donner la valeur retournée par fopen().
- s'il est de même type, ou void, lui donner la valeur d'un autre pointeur réputé correctement initialisé.

Pour accéder à la valeur pointée, le pointeur doit être typé (donc différent de void). Dans ce cas, on peut obtenir la valeur en utilisant l'opérateur de déréférencement '*'.

```
/* Definition de la variable 'a' valant 4 */  
int a = 4;  
  
/* Definition d'un pointeur 'p' initialise' avec l'adresse de la variable 'a' */  
int *p = &a;  
  
/* Definition de la variable 'b' non initialisee */  
int b;  
  
/* recuperation de la valeur de 'a' dans 'b' via le pointeur 'p' */  
b = *p;  
  
/* 'b' vaut maintenant 4 */
```

XXI-B-2 - Pointeur de fonction

Une fonction a une adresse qui est le nom de cette fonction. Un pointeur de fonction peut recevoir cette adresse. Il est possible, via un pointeur de fonction correctement initialisé, d'appeler une fonction.

Cette capacité du langage C lui confère une puissance rarement égalée, qui permet d'écrire du code flexible, dont les adresses des fonctions peuvent être définies à l'exécution. Cela permet de 'personnaliser' des fonctions génériques selon les besoins.

La définition des pointeurs de fonctions est un peu complexe, et a tendance à alourdir le code :

```
/* pointeur sur une fonction avec 2 parametres */
int (*pf) (int, char **);

/* prototype d'un fonction ayant un pointeur de fonction comme parametre */
int fun (int (*pf) (int, char **));
```

Si on doit manipuler des fonctions qui retournent un pointeur de fonction, ou des tableaux de pointeurs de fonction, le code devient rapidement illisible. C'est pourquoi il est fortement conseillé, et ce dans tous les cas, d'utiliser un typedef pour créer un alias sur le type de la fonction.

```
/* definition d'un alias */
typedef int fun_f (int, char **);

/* definition d'un pointeur de fonction de ce type */
fun_f *pf;

/* prototype d'une fonction ayant un pointeur de fonction comme parametre */
int fun (fun_f *pf);

/* tableau de pointeurs de fonction */
fun_f *pf[10];

/* prototype d'une fonction retournant un pointeur de fonction */
fun_f *getfunc (int);
```

La lecture et la maintenance du code s'en trouvent considérablement allégées.

Pour initialiser un pointeur de fonction, on peut soit :

- lui donner la valeur 0 ou NULL, qui signifie "invalide, ne pas utiliser".
- lui donner l'adresse d'une fonction.
- s'il est de même type, lui donner la valeur d'un autre pointeur réputé correctement initialisé.

Pour utiliser le pointeur, il suffit de l'invoquer comme une fonction.

```
/* definition d'un alias */
typedef int fun_f (int);

/* definition d'un pointeur de fonction de ce type */
fun_f *pf;

/* prototype d'une fonction du meme type */
int fonction (int);

/* NOTA : on peut aussi utiliser le type */
fun_f fonction;

/* initialisation du pointeur de fonction */
pf = fonction;

/* appel de la fonction via le pointeur de fonction */
```

```
pf (123);
```

XXI-B-2-a - Remarques importantes

- `void*` n'est pas un type correct pour un pointeur de fonction.
- Il n'existe pas de type générique pour les pointeurs de fonctions.

XXI-C - Du bon usage des pointeurs

Un pointeur est avant tout une variable. Comme toutes les variables, elle doit être initialisée avant d'être utilisée.

Pour un pointeur en général, l'utilisation signifie le passage de sa valeur à une fonction. Pour un pointeur sur objet, c'est le déréférencement par l'opérateur `*`. Pour un pointeur de fonction, c'est l'appel de cette fonction via le pointeur.

Il est recommandé de donner une valeur **significative** à un pointeur. Soit il est invalide, et on lui donne la valeur 0 ou NULL, soit il est valide, et dans ce cas sa valeur est celle de l'adresse d'un objet ou d'une fonction valide. Si le bloc mémoire ou la fonction deviennent invalides, il est recommandé de donner au pointeur la valeur 0 ou NULL.

```
/* Definition du pointeur. Il est initialise a l'etat invalide */
int *p = NULL;

/* le pointeur est initialise avec l'adresse d'un tableau
 * dynamique de 4 elements
 */
p = malloc (4 * sizeof *p);

/* en cas d'échec d'allocation, malloc() retourne NULL */
if (p != NULL)
{
    /* ... */

    /* apres utilisation, l'espace memoire est libere */
    free (p);

    /* le pointeur est force a l'etat invalide */
    p = NULL;
}
```

XXII - Les pointeurs, ça sert à quoi ?

Les **pointeurs** sont un peu l'essence même du C et de tous les autres langages. Sauf que dans beaucoup de langages, cette notion est considérée comme honteuse (ou trop technique), et des astuces sont utilisées pour 'cacher' les pointeurs.

En C, on n'a pas honte des pointeurs et on les affiche ostensiblement.

XXII-A - A quoi ça sert?

XXII-A-1 - Petit rappel.

Un paramètre de fonction est une variable locale de la fonction dont la valeur est donnée par l'appelant. Si on modifie cette valeur, on ne fait que modifier une variable locale. Exemple :

```
{  
    int x = 123;  
    f (x);  
}
```

avec

```
void f (int a)  
{  
    a++;  
}
```

Déroulement des opérations :

- x prend la valeur 123
- appel de la fonction f() avec en paramètre la valeur de x (soit 123)
- Dans f(), la variable locale a prend la valeur 123
- la valeur de a est augmentée de 1 et devient 124
- la fonction se termine
- la valeur de x n'a pas changé, elle vaut toujours 123.

Si on avait voulu modifier x en appelant f(), c'est raté.

XXII-A-2 - Comment faire ?

Tout simplement en donnant un moyen à la fonction qui lui permette de modifier la variable x. Ce moyen est simple. Il suffit de lui donner l'adresse de x et elle pourra alors modifier x grâce à un pointeur et à l'opérateur de déréférencement (*).

En fait, on va faire ceci :

```
{  
    int x = 123;  
    int *p = &x;  
    (*p)++;  
}
```

C'est à dire

- x prend la valeur 123
- p prend l'adresse de x
- x est incrémenté via p et l'opérateur de déréférencement. Il vaut maintenant 124.

sauf que les opérations vont être réparties entre l'appelant et l'appelé comme ceci :

```
{  
    int x = 123;  
    f (&x);  
}
```

avec

```
void f (int *p)  
{  
    (*p)++;  
}
```

- x prend la valeur 123
- l'adresse de x est passée à la fonction
- dans f(), la variable locale p est initialisée avec l'adresse de x
- la valeur de x est incrémentée de 1 via p et l'opérateur de déréférencement. Elle vaut maintenant 124.
- la fonction se termine
- la valeur de x a changé, elle vaut maintenant 124.

Voilà donc un exemple d'utilisation des pointeurs. Pour un tableau, par exemple, il n'y a pas le choix. La seule façon de faire est de passer l'adresse du premier élément du tableau via un pointeur sur le même type que l'élément.

Ensuite, on a accès à tous les éléments du tableau. Non seulement le [0] en *p, mais aussi aux autres, grâce aux propriétés de l'arithmétique des pointeurs.

En effet, le type étant connu, l'adresse des autres éléments est tout simplement p+1, p+2 etc.

L'élément lui même se trouve donc en *(p + 1), *(p + 2) etc. Le langage C définit cette écriture comme strictement équivalente à p[1], p[2] etc. On dit alors que la 'notation tableau' peut s'appliquer aux pointeurs. **Mais cela ne signifie pas qu'un pointeur soit un tableau ni inversement comme on le lit parfois.**

Ce principe est massivement utilisé avec les chaînes de caractères, qui, rappelons le, sont des tableaux de char initialisés avec des valeurs de caractères et terminés par un 0.

XXIII - Un tableau n'est pas un pointeur ! Vrai ou faux ?

Le tableau est probablement le concept le plus difficile à définir correctement en C. Il y a en effet beaucoup de confusion sur les termes : tableau, adresse, pointeur, indices... Ce petit article essaye de tirer les choses au clair.

- Un tableau est une séquence d'éléments de types identiques.
- Le nom du tableau est invariant. Il a la valeur et le type de l'adresse du premier élément du tableau. C'est donc une adresse typée (encore appelée pointeur constant^[1]). Etant de la même nature qu'un pointeur, les mêmes règles d'adressage s'appliquent, à savoir que le premier élément est en `tab`, soit `tab + 0` et que son contenu est donc `*(tab + 0)`. De même le contenu du deuxième élément est `*(tab + 1)` etc.
- Cette syntaxe étant un peu lourde, le langage C définit une simplification qui est `tab[0]`, `tab[1]` etc. Le nombre entre les crochets est appelé indice. Son domaine de définition pour un tableau de taille `N` est 0 à `N-1`.


représentation graphique d'un tableau de 4 éléments :

```
tab      : |-----|
tab[0]   : |---|
tab[1]   :      |---|
tab[2]   :          |---|
tab[3]   :              |---|
```

^[1] C'est là que se situe la difficulté. Le langage C parle de *non-modifiable L-value* ce qui signifie que c'est un objet (il a une adresse), non modifiable. On ne peut pas changer sa valeur. On ne peut changer que la valeur de ses éléments.

XXIV - Passer un tableau à une fonction

XXIV-A - Introduction

 **Rappel** : en langage C, les passages de paramètres se font exclusivement par valeur.

Le langage C n'autorise pas le passage d'un tableau en paramètre à une fonction. La raison est probablement une recherche d'efficacité, afin d'éviter des copies inutiles.

Le but de 'passer un tableau' à une fonction est en fait de permettre à celle-ci d'accéder aux éléments du tableau en lecture ou en écriture. Pour se faire, l'adresse du début du tableau et le type des éléments suffisent à mettre en oeuvre l'arithmétique des pointeurs. Un paramètre 'pointeur' est donc exactement ce qu'il faut.

XXIV-B - Tableau à une dimension

Soit l'appelant :

```
int main (void)
{
    int tab[5];

    clear (tab);

    return 0;
}
```

Le prototype de la fonction appelée doit comporter un pointeur du même type que les éléments du tableau, pour recevoir l'adresse du premier élément de celui-ci, soit :

```
void clear (int *p);
```

La fonction va donc utiliser le paramètre pointeur, dont la valeur est l'adresse du premier élément du tableau, pour accéder aux éléments du tableau. Par exemple, les mettre à 0 :

```
void clear (int *p)
{
    *(p + 0) = 0; /* premier element, */
    *(p + 1) = 0; /* deuxieme element, */
    *(p + 2) = 0; /* troisieme element, */
    *(p + 3) = 0; /* quatrieme element, */
    *(p + 4) = 0; /* dernier element (cinquieme) */
}
```

Afin d'alléger l'écriture, le langage C autorise l'utilisation de la syntaxe des tableaux pour accéder aux éléments :

```
void clear (int *p)
{
    p[0] = 0; /* premier element, */
    p[1] = 0; /* deuxieme element, */
    p[2] = 0; /* troisieme element, */
    p[3] = 0; /* quatrieme element, */
    p[4] = 0; /* dernier element (cinquieme) */
}
```

Cette implémentation est évidemment théorique, car dans la pratique, on utilisera une boucle et un paramètre supplémentaire (nombre d'éléments) afin d'écrire un code plus souple et auto adaptatif.

```
void clear (int *p, size_t nb)
{
    size_t i;

    for (i = 0; i < nb; i++)
    {
        p[i] = 0;
    }
}

int main (void)
{
    int tab[5];

    clear (tab, 5);


    return 0;
}
```

XXIV-C - Tableau à n dimensions

Rappelons que lorsqu'on définit un paramètre, les syntaxes type `*param` et type `param[]` sont sémantiquement équivalentes.

Pour définir un paramètre de type pointeur sur un tableau à 2 dimensions, on serait tenté d'écrire type `p[][]`, ce qui serait une erreur de syntaxe. En effet, la notation `[]` est une notation abrégée de `[TAILLE]` dans les cas où cette taille est ignorée par le compilateur, c'est à dire lorsque la dimension concernée est la plus à gauche. Les syntaxes suivantes sont légales :

```
type_retour fonction (int p[])
type_retour fonction (int p[12])
type_retour fonction (int p[][34])
type_retour fonction (int p[56][78])
etc.
```

 *Pour déterminer le nombre d'éléments d'un tableau, on peut utiliser les propriétés des tableaux. En effet, un tableau est une suite contiguë d'éléments identiques. Le nombre d'éléments est donc tout simplement le rapport entre la taille en bytes du tableau (`sizeof tab`) et la taille d'un élément en bytes (`sizeof tab[0]` ou `sizeof *tab`), que l'on généralise sous la forme d'une macro bien connue :*

```
#define NB_ELEM(a) (sizeof (a) / sizeof *(a))
```

XXV - Retourner un tableau

En C une fonction ne sait pas 'retourner un tableau'.

Ce qu'elle sait faire, c'est retourner une valeur. La pratique courante est de retourner l'adresse du premier élément du tableau. Pour cela, on définit le type retourné comme un pointeur sur le type d'un élément du tableau.

```
T *f();
```



T représente le type d'un élément du tableau

Evidemment, cette adresse doit être valide après exécution de la fonction. Même si c'est techniquement possible, il est donc hors de question de retourner l'adresse d'un élément appartenant à un tableau local.

- Soit on passe à la fonction l'adresse du premier élément d'un tableau existant, et elle peut retourner l'adresse de ce tableau.
- Soit la fonction fait une allocation dynamique et retourne l'adresse du bloc alloué.
- On peut aussi retourner l'adresse du premier élément d'une chaîne littérale. Celle-ci est statique (attention accès en lecture seule, qualificateur 'const' recommandé).
- Enfin, il est techniquement possible de retourner l'adresse du premier élément d'un tableau statique, mais cette pratique est déconseillée, car elle rend la fonction non-réentrante, donc impropre à plusieurs utilisations comme les appels imbriqués ou les threads... Plusieurs fonctions du C sont malheureusement victimes de ce défaut (ctime() asctime(), strtok() etc.)

XXVI - char* char**

XXVI-A - Introduction

En langage C, beaucoup de problèmes de codage et d'exécution proviennent d'une confusion entre tableaux et pointeurs. C'est particulièrement vrai avec les chaînes de caractères, au point d'utiliser le terme 'Char étoile' à la place du terme 'Chaîne de caractères' ou 'Char étoile étoile' à la place de 'Tableau de chaînes'. Qu'en est-il exactement ?

XXVI-B - Chaîne de caractères

Une chaîne de caractères est un tableau de char terminé par un 0. Une chaîne littérale n'est pas modifiable.

```
/* interdit */
"hello"[2] = 'x';

/* autorise' */
char s[] = "hello";

s[2] = 'x';
```

XXVI-C - Le type char *

Un pointeur sur char est une variable qui peut contenir NULL, l'adresse d'un char ou celle d'un élément d'un tableau de char. Si c'est un tableau, on n'a aucune information sur le nombre d'éléments du tableau pointé. Néanmoins, si c'est une chaîne valide, elle est terminée par un 0 qui sert de balise de fin.

```
char c;
char *pa = &c;

char *pb = 0;
char *pc = NULL;

char s[] = "hello";

char *pd = s;
char *pe = s + 3;

/* une chaîne littérale n'étant pas modifiable,
 * il est conseillé de qualifier l'objet avec
 * 'const' (read-only)
 */
char const *pf = "hello";
char const *pg = "hello" + 2;
```

Un petit schéma pour modéliser :

Représentation graphique d'un objet 'c' de type char non initialisé :

```
char c;

:-----:-----:
: adresse : valeur :
:         :         :
:         :         :
: &c      : ???    :
:         :         :
```

Représentation graphique d'un objet 'c' de type char après initialisation :

```
c = 'A';

:-----:-----:
: adresse : valeur :
:         :         :
:-----:-----:
: &c      : 'A'    :
:-----:-----:
```

Représentation graphique d'un objet 'p' de type char * non initialisé :

```
char *p;

:-----:-----:-----:
: adresse : valeur : valeur :
:         :         : pointée :
:-----:-----:-----:
: &p      : ???   : ???   :
:-----:-----:-----:
```

Représentation graphique d'un objet 'p' de type char * après initialisation (NULL) :

```
p = NULL;

:-----:-----:-----:
: adresse : valeur : valeur :
:         :         : pointée :
:-----:-----:-----:
: &p      : NULL  : ???   : --> NULL
:-----:-----:-----:
```

Représentation graphique d'un objet 'p' de type char * après initialisation (adresse d'une variable) :

```
p = &c;


:-----:-----:-----: :-----:-----:
: adresse : valeur : valeur : : adresse : valeur :
:         :         : pointée : :         :         :
:-----:-----:-----: :-----:-----:
: &p      : &c    : 'A'    : --> : &c    : 'A'    :
:-----:-----:-----: :-----:-----:
```

Représentation graphique d'un objet 'p' de type char * après initialisation (adresse d'une chaîne modifiable) :

```
char s[]="ab";
p = s;

:-----:-----:-----: :-----:-----:
: adresse : valeur : valeur : : adresse : valeur :
:         :         : pointée : :         :         :
:-----:-----:-----: :-----:-----:
: &p      : &s[0] : 'a'    : --> : s+0    : 'a'    :
:         : ou s+0 :         : :         :         :
:         : ou s  :         : : s+1    : 'b'    :
:-----:-----:-----: :-----:-----:
:         :         :         : : s+2    : 0      :
:-----:-----:-----: :-----:-----:
```

Le pointeur sur char est principalement utilisé pour les paramètres 'chaînes de caractères' des fonctions et pour des manipulations de chaînes de caractères.

 *Mais cela n'autorise pas à utiliser le terme 'char étoile' à la place de 'chaîne de caractères'.*

XXVI-D - Le type char **

Un pointeur sur pointeur de char est une variable qui peut contenir NULL, l'adresse d'un pointeur de char ou celle d'un élément d'un tableau de pointeurs de char. Si c'est un tableau, on a aucune information sur le nombre d'éléments du tableau pointé. On peut parfois ajouter un élément de valeur NULL pour délimiter le tableau de pointeurs. Un petit schéma pour modéliser :

XXVII - Initialisation des tableaux de caractères

Il est possible d'initialiser un tableau de char au moment de sa définition avec une constante qui ressemble a une chaîne de caractères. Il faut cependant faire attention, car il n'est pas garanti que le tableau ainsi initialisé forme une chaîne C valide (c'est à dire terminée par un 0).

En effet, la liste de caractères placée entre double quotes et servant à initialiser le tableau de char n'est en aucun cas une chaîne de caractères. C'est une simple liste de caractères. Les zéros que l'on voit dans le tableau sont le résultat du comportement standard du C qui complète à 0 les tableaux partiellement initialisés.

Exemples

```
char s[4] = "ab";
```

le tableau est initialisé avec {'a', 'b', 0, 0} : Chaîne valide

```
char s[4] = "abc";
```

le tableau est initialisé avec {'a', 'b', 'c', 0} : Chaîne valide

```
char s[4] = "abcd";
```

le tableau est initialisé avec {'a', 'b', 'c', 'd'} : Chaîne invalide !\


```
char s[4] = "abcde";
```

Ne compile pas (trop d'initialisateurs)

XXVIII - Qu'est-ce qu'une chaîne littérale ?

Une chaîne littérale, telle qu'elle apparaît dans un source C, est une séquence de caractères entourée de guillemets (double quotes)

```
"hello"
```

 Elle ne doit pas être confondue avec la liste de caractères servant à initialiser un tableau de char, par exemple :

```
char s[] = "hello";
```

(les détails sont indiqués [ici](#).)

Une chaîne littérale désigne en réalité l'adresse du premier élément d'un tableau de char anonyme non modifiable, situé en mémoire statique, initialisé avec la séquence de caractères mentionnés et terminé par un 0.

Tout se passe comme si on avait ceci :

```
static char const identificateur_connu_seulement_du_compilateur[] = {'h','e','l','l','o',0};
```

Si la chaîne apparaît dans un paramètre de fonction :

```
f ("hello");
```

c'est cette valeur (l'adresse) qui est passée à la fonction dans son paramètre :

```
void f (char const *s);
```

Si la chaîne sert à initialiser un pointeur :

```
char const *p = "hello";  
  
p = "bye";
```

c'est cette valeur (l'adresse) qui est stockée dans le pointeur.

RAPPEL : le mot clé **const** sert à qualifier l'objet de non modifiable

XXIX - Bien utiliser malloc()

Il est fréquent de rencontrer ce genre de code :

```
#include <stdlib.h>
...
{
    /* creation d'un tableau de 10 int */
    size_t n = 10;
    int *p = (int*) malloc (sizeof (int) * n);

    if (p != NULL)
    {
        ...
    }
}
```

Ce code est correct et ne présente pas de comportement indéterminé. Cependant, il est inutilement compliqué et peut être amélioré de plusieurs façons.

XXIX-A - Suppression du cast

Il est d'usage d'éviter les casts en C. Certains sont indispensables, d'autres non. Ici, par exemple, et contrairement aux idées reçues, le cast est inutile, et on peut parfaitement écrire :

```
{
    int *p = malloc (sizeof (int) * n);
}
```

Il est cependant des cas rares où le cast est indispensable.

- Le compilateur n'est pas conforme à ISO C-90 ou ISO C-99
- Le compilateur n'est pas C mais par exemple pré-C++98

XXIX-A-1 - Compilateur non ISO

Il est rare de nos jours d'utiliser un compilateur datant d'avant la normalisation du langage C (1989 aux USA, 1990 au niveau international). En effet, ces compilateurs ne supportent pas les prototypes, ce qui les rend impropre à produire du code cohérent, à moins d'utiliser un outil de vérification indépendant comme PCLint.

Le cas peut cependant se produire, s'il s'agit de maintenir du code ancien avec une chaîne de compilation ancienne. Dans ce cas, effectivement, le cast est indispensable si le type du pointeur est différent de char*.

L'opportunité de conserver une telle pratique est donc laissée à l'appréciation du programmeur. Il semble cependant assez évident que dans les nouveaux développements utilisant un compilateur ISO, il est inutile d'ajouter le cast.

XXIX-A-2 - Compilateur C++

Il est techniquement possible, à de rares exceptions syntaxiques près, de faire compiler du code C par un compilateur C++. Néanmoins, cette pratique est rarement justifiée et est largement déconseillée.

En effet, en dehors des points syntaxiques évidents (comme par exemple la conversion de type explicite void* <-> type* qui justement oblige à utiliser le cast) plusieurs points de sémantique diffèrent entre les deux langages. En l'état actuel des normes, les spécifications C++98 et C99 ont même plutôt tendance à diverger (cette situation pourrait changer en 2005 avec une nouvelle révision de C++ intégrant les nouveautés de C99).

On peut aussi se demander pourquoi on utiliserait `malloc()` et `free()` en C++, alors que ce langage dispose des opérateurs `new` et `delete`. D'autre part, en C++98, un cast se fait avec `static_cast<...>`.

Lire à ce sujet :  **Incompatibilities Between ISO C and ISO C++**

XXIX-B - Déterminer la taille sans le type

Il est courant de déterminer la taille d'un objet en utilisant son type

```
{  
    int *p = malloc (sizeof (int));  
}
```

Si le type change, on est obligé de modifier 2 fois le code:


```
{  
    long *p = malloc (sizeof (long));  
}
```

Lorsqu'il s'agit d'un pointeur typé, il existe une technique alternative qui consiste à utiliser la taille d'un élément pointé par ce pointeur :

```
{  
    int *p = malloc (sizeof *p);  
}
```

Le changement de type se trouve largement simplifié :

```
{  
    long *p = malloc (sizeof *p);  
}
```

Quelques compléments sur `malloc()` et l'allocation dynamique en général dans l'article  **Description des mécanismes d'allocation dynamique de mémoire en langage C**

XXX - Bien utiliser realloc()

La fonction `realloc()`, bien que souvent décriée pour sa lenteur, offre une alternative intéressante pour gérer des tableaux de taille variable. Bien sûr il ne faut pas allouer les objets un par un, mais par blocs (doublage, par exemple).

Pour utiliser correctement `realloc()`, quelques précautions doivent être prise. Par exemple :

```
#include <stdlib.h>

<...>
{
    /* allocation d'un tableau de 10 int
     * Pour pouvoir gerer la taille, on utilise
     * une variable 'taille'. Une structure comprenant
     * l'adresse du tableau et sa taille est aussi envisageable.
     */

    size_t size = 10;
    int *p = malloc (size * sizeof *p);

<...>

    /* Agrandissement du tableau a 15 int */
    {
        /* reallocation. Le resultat est stocke'
         * dans une variable temporaire
         */
        size = 15;
        type_s *p_tmp = realloc (p, size * sizeof *p_tmp);

        if (p_tmp != NULL)
        {
            /* si la nouvelle valeur est valide,
             * le pointeur original est mis a jour.
             */
            p = p_tmp;
        }
        else
        {
            /* l'ancien bloc est valide, mais il n'a pas ete agrandi */
        }
    }
}
```


Il faut aussi garder à l'esprit que la partie nouvellement allouée n'est pas initialisée.

XXXI - Comment créer un tableau dynamique à 2 dimensions ?

Il y a plusieurs façons de créer un tableau dynamique à deux dimensions de type T. La plus courante consiste à créer un tableau de N lignes contenant les adresses des N tableaux de M colonnes. L'avantage de cette méthode est qu'elle permet un usage habituel du tableau avec la notation `[i][j]`.

Comme le tableau de N lignes contient des adresses, ses éléments sont donc des pointeurs sur T. Il se définit ainsi :

```
T* *pp = malloc (sizeof (T*) * N);
```

 *T représente le type d'un élément du tableau*

Ensuite, chaque élément reçoit l'adresse du premier élément d'un tableau alloué de M colonnes. Chaque élément est donc de type T :

```
size_t i;
for (i = 0; i < N; i++)
{
    pp[i] = malloc (sizeof (T) * M);
}
```

Bien sûr, pour une utilisation correcte, il faut en plus tenir compte du fait que `malloc()` peut échouer et qu'il faut libérer les blocs alloués après usage.

D'autre part, je rappelle que les valeurs d'un bloc fraîchement alloué sont indéfinies.

Enfin, selon les principes énoncés [ici](#), on peut simplifier le codage comme ceci :

```
T **pp = malloc (sizeof *pp * N);
```

```
size_t i;
for (i = 0; i < N; i++)
{
    pp[i] = malloc (sizeof *pp[i] * M);
}
```

ce qui facilite la maintenance et évite bien des erreurs de type, le choix étant confié au compilateur.

Il va sans dire qu'il faut ensuite libérer le tableau alloué selon le procédé inverse :

```
size_t i;
for (i = 0; i < N; i++)
{
    free(pp[i]), pp[i] = NULL;
}
free(pp), pp = NULL;
```

XXXII - Saisie de données par un opérateur (stdin)

XXXII-A - Introduction

Il est courant en C standard d'utiliser le flux stdin pour acquérir des données en provenance d'un opérateur. (Mode conversationnel). On admettra pour la suite que stdin est connecté à la partie 'clavier' d'un périphérique console.

Le langage C offre plusieurs fonctions permettant de lire des données sur un flux en général et sur stdin en particulier.

- `fgetc()`
- `getc()`
- `getchar()`
- `gets()`
- `scanf()`
- `fgets()`

XXXII-B - `fgetc()`, `getc()`, `getchar()`

Ces trois fonctions extraient **un** caractère du flux entrant (pour `getchar()`, ce flux est stdin). C'est insuffisant pour saisir autre chose qu'un simple <ENTER>. Ces fonctions ne sont absolument pas adaptées à la saisie d'un caractère comme un choix de menu par exemple.

Par contre, ces fonctions peuvent être utilisées pour construire des fonctions d'entrées de plus haut niveau plus ou moins spécialisées.

Détails de fonctionnement de `fgetc()`


XXXII-C - `gets()`

Pour des raisons évidentes de sécurité (pas de limitation du nombre de caractères saisis), la fonction `gets()` ne devrait pas être utilisée. Bien que, à ma connaissance, cette fonction ne soit pas officiellement dépréciée pour des raisons de compatibilité avec le code existant, **il est fortement conseillé de ne pas l'utiliser pour de nouveaux développements**.

XXXII-D - `scanf()`

Malgré ce que l'on constate dans l'abondante littérature consacrée à l'initiation au langage C, **l'utilisation de `scanf()` n'est pas adaptée**.

En effet, le 'f' de `scanf()` est là pour nous rappeler que l'entrée doit être formatée (*formatted*), ce qui n'est évidemment pas le cas avec un opérateur humain qui peut entrer n'importe quoi. D'autre part, `scanf()` gère difficilement le '\n', ce qui entraîne des comportements aberrants dans les saisies si on ne prend pas certaines précautions d'usage.

L'utilisation correcte et sûre de `scanf()` est complexe, et n'est pas à la portée d'un débutant (ni même à celle de la plupart des programmeurs expérimentés). Néanmoins, il est possible d'utiliser correctement `scanf()` si on se forme correctement. En lisant par exemple l'article  **Scanf démythifiée**.

XXXII-E - `fgets()`

Cette fonction est parfaitement adaptée à la saisie d'une ligne, (même de 1 caractère). Son usage est recommandé.

S'il faut saisir une valeur numérique, celle-ci sera d'abord saisie sous forme de ligne, puis traduite par la fonction appropriée (`strtol()`, `strtoul()`, `strtod()` ou `sscanf()`) avec le filtre approprié :

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

int main(void)
{
    int ret;
    char temp[20];

    do
    {
        char saisie[20];

        printf("Entrez un nombre : ");
        fflush (stdout);

        fgets (saisie, sizeof saisie, stdin);

        /* Filtrage des caracteres (entier decimal)
         * Nota : la saisie s'arrete a la premiere erreur.
         * Ce qui est saisi avant est considere comme valide.
         *
         * "123a" -> "123" : ret = 1
         *
         * "a123" -> ""    : ret = 0
         */
        ret = sscanf (saisie, "%[0-9-]s", temp);
    }
    while (ret != 1);

    {
        long n = strtol (temp, NULL, 10);

        printf ("La chaine est '%s', soit %ld\n", temp, n);
    }
    return 0;
}
```

D'autres exemples dans le chapitre sur les **fichiers**

XXXII-F - Ressources



Comment se fabriquer des entrées solides en C

XXXII-G - Comment fonctionne `fgetc(stdin)` alias `getchar()`

Cette fonction d'apparence simple a en fait un comportement plus complexe qu'il n'y parait. En effet, elle regroupe un certain nombre de comportements non triviaux qui sont rarement expliqués dans la littérature C.

XXXII-G-1 - Comportement visible.

L'appel de cette fonction provoque une suspension de l'exécution du programme. Durant cette suspension, il est possible de rentrer des caractères (par exemple à l'aide du clavier) et même éventuellement de supprimer le ou les derniers caractères saisis à l'aide de la touche 'BackSpace'. La fin de saisie (et la reprise de l'exécution du programme) est marquée par la frappe de la touche <enter>.

XXXII-G-2 - Comportement interne.

Les caractères saisis sont stockés dans le flux stdin. Lorsque l'on frappe la touche <enter>, le caractère '\n' est aussi placé dans stdin, et l'exécution reprend. Le caractère le plus ancien est alors extrait du flux et il est retourné. En cas d'erreur de lecture ou d'entrée d'un caractère spécial dit 'de fin de fichier' (Ctrl-D, Ctrl-Z etc. selon le système), la valeur EOF (int < 0) est retournée.

Ensuite, si on rappelle fgetc(), deux cas sont possibles. Soit le flux est vide, soit il ne l'est pas. Si le flux est vide, la fonction fgetc() suspend l'exécution, et on retrouve le comportement précédent. S'il n'est pas vide, l'exécution n'est pas suspendue, et le caractère le plus ancien est extrait et retourné.

Dans la grande majorité des cas la lecture du '\n' signifie que la ligne saisie a été complètement lue.

XXXII-G-3 - Quelques expérimentations.

A l'aide de simples programmes, il est possible de vérifier un certain nombre de comportements décrits précédemment :

```
#include <stdio.h>

int main (void)
{
    int x = fgetc(stdin);

    printf ("x = %d ('%c')\n", x, x);

    return 0;
}
```

Quelques essais de saisie :

```
<enter>
x = 10 ('
')
```

On voit que le caractère extrait est '\n' (ici, LF, soit le code ASCII 10)

```
a<enter>
x = 97 ('a')
```

On voit que le caractère extrait est 'a' (ici, le code ASCII 97). Le <enter> ('\n') n'a pas été extrait. Si on appelait fgetc() une nouvelle fois, il n'y aurait pas de suspension.

```
a<backspace>b<enter>
x = 98 ('b')
```

On constate que, bien que le premier caractère saisi fut 'a', le caractère extrait est 'b' (ici, le code ASCII 98). En effet, la touche <backspace> a permis de corriger la dernière saisie.

```
abcd<enter>
x = 97 ('a')
```

On voit que le caractère extrait est 'a', bien que d'autres caractères aient été saisis après. C'est donc bien le plus ancien caractère qui est extrait. Les autres caractères sont en attente de lecture. Une boucle de fgetc() permettrait de les extraire.

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



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

    do
    {
        x = fgetc(stdin);
        printf ("x = %d ('%c')\n", x, x);
    }
    while (1);

    return 0;
}
```

Je laisse au lecteur le soin de refaire les expériences précédentes et d'en tirer les conclusions qui s'imposent.

XXXIII - Les fichiers

XXXIII-A - Introduction

Le langage C n'offre pas, à proprement parler, de gestion de fichiers. Il définit plutôt des flux d'entrées / sorties (*O streams*) sur lesquels il peut agir (ouverture/fermeture, lecture/écriture). L'unité d'information gérée par un flux est le byte.

Certains de ces flux sont connectés à des périphériques permettant par exemple de réaliser une interface entre la machine et l'utilisateur (IHM) en mode texte. Mais la plupart du temps, le nom associé au flux est en fait un 'fichier', c'est-à-dire une sorte de mémoire (disque, flash) accessible en écriture et en lecture par l'intermédiaire du système. L'avantage évident est que les données sont permanentes, même après mise hors tension de la machine.

En conséquence, dans la pratique, les termes flux et fichiers sont souvent confondus.

XXXIII-B - Texte ou binaire ?

Le langage C fait la distinction entre les fichiers binaires et les fichiers textes. Cette distinction est historique. Elle dépend en fait du système utilisé. Sur certains systèmes, il n'existe aucune différence physique entre les fichiers textes et les fichiers binaires. Sur d'autres systèmes, il existe une différence. Par souci de portabilité, il est recommandé de respecter cette distinction.

Le choix entre fichier texte ou binaire provient du contenu de ce fichier.

XXXIII-B-1 - Fichier texte

On appelle fichier texte un fichier qui contient des informations de type texte, c'est à dire des séquences de lignes.

Une ligne est une séquence de caractères imprimables terminée par une marque de fin de ligne.

Selon le système, la marque de fin de ligne est composée de un ou plusieurs caractères de contrôle (par exemple, CR, LF, ou une séquence de ces caractères)

Système	Fin de ligne	Fin de fichier
Unix		
Mac X	0x0A LF	Sans objet
Linux		
Mac (non unix)	0x0D CR	Sans objet
MS-DOS	0x0D CR	0x1A
Windows	0x0A LF	^Z
Windows NT		
VMS STREAM_CR	0x0D CR	Sans objet
VMS STREAM_LF	0x0A LF	Sans objet
VMS STREAM_CRLF	0x0D CR	Sans objet
	0x0A LF	

L'ensemble des valeurs numériques des caractères (charset) dépend du système. La plupart du temps, il s'agit du codage ASCII (0-127) avec des extensions plus ou moins standards au delà de 127. Il existe d'autres codes, comme EBCDIC utilisé sur certains mainframes IBM.

Pour écrire une fin de ligne dans un fichier texte, il suffit d'écrire le caractère '\n'. Celui-ci sera alors automatiquement traduit en marqueur de fin de ligne.

De même, lors de la lecture d'un fichier texte, le marqueur de fin de ligne est automatiquement traduit en '\n', quel qu'il soit.

Nota : Certains systèmes marquent la fin des fichiers textes d'un caractère spécial. Par exemple MS-DOS ajoute un code 26 (^Z). Cela signifie que, pour ce système, la lecture d'un fichier texte s'arrête dès la rencontre de ce caractère.

XXXIII-B-2 - Fichier binaire

N'importe quel fichier, y compris un fichier texte, peut être considéré comme binaire. Dans ce cas, l'écriture et la lecture des caractères se fait sans interprétation.

Par exemple, sur une plateforme utilisant le jeu de caractères ASCII, CR vaut 13 ou 0x0D ou '\r'. De même, LF vaut 10 ou 0x0A ou '\n'.

XXXIII-B-3 - Modes d'ouverture d'un fichier

La fonction d'ouverture de fichier est `fopen()`. Comme pour les autres fonctions de gestion des fichiers, le fichier d'interface est `<stdio.h>`.

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

Le mode d'ouverture est déterminé par une chaîne de caractère. Voici les chaînes correspondant aux principaux modes :

```
"r" : mode texte en lecture
"w" : mode texte en écriture (création)
"a" : mode texte en écriture (ajout)

"rb" : mode binaire en lecture
"wb" : mode binaire en écriture (création)
"ab" : mode binaire en écriture (ajout)
```

XXXIII-B-4 - Lecture d'un fichier

Le langage C offre plusieurs fonctions permettant de lire les données d'un fichier.

- `fgetc()`
- `getc()`
- `fread()`
- `fscanf()`
- `fgets()`

XXXIII-B-4-a - `fgetc()`, `getc()`

Ces fonctions sont identiques. Elles permettent de lire un caractère.

XXXIII-B-4-b - `fread()`

Cette fonction permet de lire un bloc de caractères d'une longueur donnée. Elle est tout à fait adaptée à la lecture des données binaires brutes (non interprétées).

XXXIII-B-4-c - fscanf()

Cette fonction permet de lire des données 'texte' formatées. Cette fonction est d'une utilisation complexe et son usage est peu recommandé.

XXXIII-B-4-d - fgets()

Cette fonction permet de lire une ligne de texte. Elle est tout à fait adaptée à la lecture d'un fichier texte ligne par ligne.

Sa simplicité d'utilisation et sa robustesse en font la fonction préférée des programmeurs qui doivent analyser des fichiers textes.

XXXIII-B-4-d-i - Exemple d'utilisation

Soit le fichier texte :

```
Ceci est un simple fichier
texte de 2 lignes.
```

et un petit programme permettant de lire ces 2 lignes

```
/* fichier1.c */
#include <stdio.h>

int main (void)
{
    /* ouverture du fichier en mode texte */
    FILE *fp = fopen ("data.txt", "r");

    /* L'ouverture du fichier est-elle realisee ? */
    if (fp != NULL)
    {
        /* definition d'un tableau de char destine a recevoir la ligne
         * La taille est arbitraire. Elle doit etre cependant adaptee * aux besoins courants.
         * Pour les grandes tailles (disons > 256 char),
         * il est preferable d'utiliser une allocation dynamique.
         */
        char ligne[32];

        /* lecture de la premiere ligne */
        fgets (ligne, sizeof ligne, fp);

        /* Affichage de la premiere ligne */
        printf ("1: %s\n", ligne);

        /* lecture de la deuxieme ligne */
        fgets (ligne, sizeof ligne, fp);

        /* Affichage de la deuxieme ligne */
        printf ("2: %s\n", ligne);

        /* Fermeture du fichier */
        fclose (fp);
    }
    else
    {
        printf ("Erreur d'ouverture du fichier\n");
    }
    return 0;
}
```

On doit obtenir ceci sur la sortie standard (stdout):

```
1: Ceci est un simple fichier
2: texte de 2 lignes.
```

XXXIII-B-4-d-ii - Explication

La ligne lue est stockée dans la variable ligne, y compris le '\n'. La fonction d'affichage printf() affiche le numéro de ligne, suivit de ': ', la ligne (avec son '\n') et un '\n' en plus, ce qui explique la présence de lignes "vides".

XXXIII-B-4-d-iii - Critique de cet exemple

Cet exemple de codage 'naïf' souffre d'un défaut majeur : Il fait l'hypothèse que le fichier fait 2 lignes, et il continue à lire le fichier même si une erreur de lecture s'est produite. En fait, tout simplement, il ne gère pas les erreurs de lecture.

Il est facile de gérer les erreurs de lecture. Toutes les fonctions de lecture retournent une valeur. Celle-ci peut prendre une valeur particulière qui signifie 'Arrêt de la lecture'. La cause n'est pas précisée. Ça peut être à cause d'une erreur (support en panne, données corrompu, fichier inexistant etc.) ou tout simplement par ce que la fin de fichier a été atteinte.

XXXIII-B-4-d-iv - Détection d'une erreur

La fonction fgetc() retourne une valeur de type char *. Si la lecture a réussi, la valeur retournée est l'adresse du tableau de char passé en paramètre. En cas d'échec, la valeur NULL est retournée. Il suffit donc de surveiller cette valeur pour savoir si on peut continuer ou non. Comme une des causes d'échec est la "fin de fichier atteinte", on peut donc parfaitement intégrer ce test dans une boucle de lecture "ligne par ligne".

Une fois l'échec de la lecture constaté, il est possible d'en identifier la cause. Le langage C met à disposition les deux fonctions feof() et ferror() qu'il faut appeler après la boucle de lecture, mais avant la fermeture du fichier.

```
while (fonction_de_lecture(fp) != ERREUR)
{
    ...
}

if (feof(fp))
{
    /* la fin de fichier a ete detectee */
    puts ("EOF");
}

if (ferror(fp))
{
    /* une erreur s'est produite */
    perror (NOM_DU_FICHER);
}

fclose (fp);
```

XXXIII-B-4-d-v - Gestion des fins de ligne

On constate que lorsque fgetc() lit une ligne entière, un '\n' se retrouve à la fin de la chaîne saisie. La présence de '\n' est gênante ou non selon l'application.

Ceci dit, dans tous les cas, il est conseillé d'en détecter la présence. En effet, sa présence indique que la ligne a été lue entièrement, alors que son absence indique que la ligne a été tronquée, et que d'autres caractères (au minimum un '\n') attendent pour être lus. Il est donc conseillé d'écrire ces quelques lignes après un fgets() pour clarifier la situation :

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

...

{
    char ligne[123];

    /* test d'erreur omis */
    fgets (ligne, sizeof ligne, fp);

    {
        /* chercher le '\n' */
        char *p = strchr(ligne, '\n');

        if (p != NULL)
        {
            /* si on l'a trouve, on l'elimine. */
            *p = 0;
        }
        else
        {
            /* Le traitement depend de l'application.
             * Par exemple, ici, on choisi d'ignorer
             * les autres caracteres.
             */

            /* sinon, on lit tous les caracteres restants */
            int c;

            while ((c = fgetc(fp)) != '\n' && c != EOF)
            {
                ;
            }
        }
    }
}
```

Il est clair que dans la pratique, l'ensemble de ce code devra être intégré dans une fonction unique de lecture d'une ligne à partir d'un flux.

XXXIII-B-4-d-vi - Exemple amélioré avec détection de la fin de lecture

```
/* fichier2.c */
#include <stdio.h>

int main (void)
{
    FILE *fp = fopen ("data.txt", "r");

    if (fp != NULL)
    {
        char ligne[32];

        /* definition d'un compteur de lignes et initialisation */
        int cpt = 0;

        /* lecture des lignes */
        while (fgets (ligne, sizeof ligne, fp) != NULL)
        {
            /* Mise a jour du compteur */
            cpt++;

            /* Affichage des lignes */
            printf ("%d: %s\n", cpt, ligne);
        }
    }
}
```

```

/* On peut ajouter ici la detection de la cause
 * de l'erreur decrite ci-dessus
 */

fclose (fp);
}
else
{
    printf ("Erreur d'ouverture du fichier\n");
}

return 0;
}

```

Cet exemple met en oeuvre un mécanisme qui s'adapte automatiquement au nombre de lignes du fichier. Cependant, attention, le fonctionnement, bien qu'il reste sûr, risque d'être surprenant si la longueur de la ligne est supérieure à celle du tableau 'ligne'.

Par exemple, si on diminue la taille de 'ligne' à 16 au lieu de 32,

```

<...>
char ligne[16];
<...>

```

on obtient :

```

1: Ceci est un sim
2: ple fichier

3: texte de 2 lign
4: es.

```

XXXIII-B-4-d-vii - Explication

Rappelons que la taille du tableau de char a été transmise à la fonction fgets().

Celle-ci tente de lire la ligne, mais celle-ci est trop longue pour tenir dans le variable 'ligne'. fgets(), qui connaît la taille de la variable 'ligne', applique alors une stratégie d'adaptation qui consiste à stocker ce qui est possible dans la variable, en laissant une place pour le 0 final. En effet, fgets() a pour obligation de produire une chaîne de caractères valide dans tous les cas.

C'est pourquoi la première ligne est partiellement lue ainsi :

```

0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 : Indice
'c' 'e' 'c' 'i' ' ' 'e' 's' 't' ' ' 'u' 'n' ' ' 's' 'i' 'm' 0 : Données

```

Mais les caractères manquants ne sont pas perdus, et ils sont lus par l'appel suivant:

```

0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 : Indice
'p' 'l' 'e' ' ' 'f' 'i' 'c' 'h' 'i' 'e' 'r' '\n' 0 : Données

```

Cette fois, la place est suffisante, et l'ensemble de la chaîne est lue, y compris le '\n'.

XXXIII-B-5 - Écriture dans un fichier

Le langage C offre plusieurs fonctions permettant d'écrire des données dans un fichier.

- `fputc()`
- `putc()`
- `fwrite()`
- `fprintf()`
- `fputs()`

XXXIII-B-5-a - `fputc()`, `putc()`

Ces fonctions sont identiques. Elles permettent d'écrire un caractère.

XXXIII-B-5-b - `fwrite()`

Cette fonction permet d'écrire un bloc de caractères d'une longueur donnée. Elle est tout à fait adaptée à l'écriture de données binaires brutes (non interprétées).

XXXIII-B-5-c - `fprintf()`

Cette fonction permet d'écrire des données 'texte' formatées. Elle comporte de nombreuses possibilités de conversion de valeurs numériques en texte. (Entiers, flottants etc.)

XXXIII-B-5-d - `fputs()`

Cette fonction permet d'écrire une chaîne de caractères.

XXXIII-B-6 - Bien utiliser les formats de données

Il n'est pas rare que des données enregistrées dans un fichier par une machine soient lues par une autre machine, ou par autre programme ou par le même programme mais compilé avec des options différentes. Pour pouvoir récupérer les données, il faut qu'en aucun cas, le format des données enregistrées ne dépende de l'implémentation.

XXXIII-B-6-a - Format orienté texte

Le format texte est un bon choix, car il utilise une séquence de caractères simple et évidente (chronologique) et un codage très répandu (ASCII). Il peut y avoir quelques problèmes de transcodage pour les valeurs de 128 à 255 (ANSI, OEM etc.), mais rien qui ne soit insurmontable. D'autre part, la conversion ASCII/EBCDIC est triviale.

Il subsiste le problème des fins de ligne qui sont différentes d'un système à l'autre. Il existe des utilitaires bien connus (`dos2unix`, `unix2dos` etc.) généralement fournis avec ces systèmes qui font les conversions. Rappelons que la fonction `system()` permet d'appeler une commande extérieure. Si néanmoins, cet utilitaire n'existait pas, il serait facile de le faire soi-même. Bien sûr, il faudrait travailler en mode binaire de façon à contrôler les données du fichier de manière 'brute' (`raw`).

Les chaînes et les valeurs numériques sont encodées et éventuellement formatées avec `fprintf()`. Une organisation en ligne est souhaitable. Elles sont ensuite lues ligne par ligne avec `fgets()` et analysées soit par `strtol()`, `strtoul()` ou `strtod()` pour les cas les plus simples (valeurs numériques pures), soit par `sscanf()` pour les cas plus complexes, à condition que le formatage soit clairement défini. Il est souhaitable d'utiliser des formats simples à analyser et surtout sans ambiguïté quant aux séparateurs. Le format CSV est recommandé.

XXXIII-B-6-b - Format orienté binaire

Une mauvaise utilisation des formats binaires (raw) peut apporter des problèmes de portabilité. Il est recommandé d'utiliser des formats indépendants comme XDR ( **RFC 1832**).

XXXIII-C - Supprimer un enregistrement dans un fichier binaire

Pour supprimer un enregistrement, le plus simple est de procéder ainsi:

- Le fichier original est ouvert en lecture. Un nouveau fichier est ouvert en écriture. L'original est lu enregistrement par enregistrement (`fread()`), et recopié dans le nouveau fichier (`fwrite()`) en omettant l'enregistrement à supprimer (`if ...`).
- Par un jeu subtil de suppression et de renommage (`remove()`, `rename()`), on se retrouve avec une copie de l'original (genre `.old` ou `.bak`) et le nouveau fichier qui a maintenant le nom de l'ancien. L'opération reste simple, et a l'avantage de permettre l'annulation (par renommage de l'ancien fichier).

Toute autre opération basée sur l'écriture/lecture dans le même fichier est dangereuse, non portable et se traduit souvent par la destruction du fichier original sans recours possible.

XXXIII-D - En guise de conclusion

Il ne faut pas se tromper d'outil. Les flux du C sur disque sont très pratiques pour enregistrer quelques données statiques dans un fichier texte. En binaire, c'est déjà plus risqué à moins de passer par un format indépendant comme XDR. Pour gérer des enregistrements, les fichiers C sont trop rustiques. Il faut une véritable base de données (comme **SQLite** ou **MySQL** par exemple).

XXXIV - Pourquoi fflush (stdout) ?

Il arrive parfois de rencontrer ce genre de code ...

```
printf("Entrez un nombre : ");  
fflush (stdout);
```

... et on se demande alors à quoi peut bien servir ce fflush (stdout).

Le printf() précédent envoie une chaîne de caractères à stdout. Or cette chaîne n'est pas terminée par un '\n'.

Il faut savoir que stdout est souvent un flux "bufferisé", ce qui signifie, en bon français, que les caractères sont placés dans un tampon (*buffer*) de sortie avant d'être réellement émis.

Il y a trois critères qui déclenchent l'émission réelle des caractères :

- Le tampon d'émission est plein (incontrôlable)
- Un '\n' a été placé dans le tampon^[1]
- La commande de forçage a été activée

La commande de forçage est activée par l'appel de la fonction fflush (stdout), ce qui explique sa présence dans le code mentionné.

^[1] sauf en cas de redirection dudit flux vers un fichier.

XXXV - <time.h> : La gestion du temps

XXXV-A - Introduction

Le langage C fournit un certain nombre de fonctions permettant de lire l'heure/date courante et de convertir la valeur lue en structure ou en chaîne de caractères. Ces éléments sont définis et déclarés dans le header standard <time.h>

Élément	Nature	Description
struct tm	structure	Structure de date et heure
time()	fonction	Lecture de l'heure courante
time_t	type	Type retourné par time()
localtime()	fonction	Charge la structure tm avec l'heure locale
gmtime()	fonction	Charge la structure tm avec l'heure GMT
mktime()	fonction	Normalise la structure tm Retourne le temps en time_t
strftime()	fonction	Crée une chaîne formatée à partir d'une structure tm
difftime()	fonction	Calcule la différence entre 2 dates

XXXV-B - Usage

XXXV-B-1 - Lire l'heure courante

On utilise la fonction time(). On peut soit passer l'adresse d'un time_t, soit récupérer la valeur dans un time_t. Si on utilise pas l'adresse d'un time_t, il faut le préciser à la fonction en passant NULL :

```
#include <time.h>

int main (void)
{
    time_t ta;
    time_t tb;

    time (&ta);
    tb = time (NULL);

    return 0;
}
```

Telle quelle, la valeur contenue dans `ta` et `tb` n'a pas grande signification. La norme dit que c'est un grand entier qui représente un nombre d'élément de temps depuis une date donnée. Sous POSIX.1, c'est un nombre de secondes depuis le 1er janvier 1970. (Epoch). En principe, on ne manipule pas la valeur directement, mais via sa représentation en struct `tm`.


Cette valeur a quand même un intérêt pratique. Il y a de fortes chances qu'elle soit différente d'un lancement de programme à l'autre. On l'utilise donc couramment pour initialiser le générateur pseudo-aléatoire du C avec une valeur différente à chaque lancement :

```
srand((unsigned) time(NULL));
```

ce qui permet de démarrer le cycle de la séquence pseudo-aléatoire à un endroit difficilement prévisible, renforçant ainsi l'effet 'aléatoire' pour les jeux et autres programmes statistiques.

XXXV-B-2 - Afficher l'heure courante

L'affichage de la date ou de l'heure de fait grâce à la fonction `strftime()` qui a été conçue pour ça. Elle attend l'adresse d'une structure `tm`, qu'il va donc falloir mettre à jour au préalable. On utilise pour ça la valeur retournée par `time()`, qui donne la datation courante sous la forme d'un entier et une des fonction `localtime()` ou `gmtime()` selon que l'on veut convertir en heure locale ou en heure GMT (Greenwich). En France, il y a une heure de différence.

L'usage précis de `strftime()` et de ses nombreux paramètres de formatage doit être étudié dans un  **document de référence**.

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

int main (void)
{
    /* lire l'heure courante */
    time_t now = time (NULL);

    /* la convertir en heure locale */
    struct tm tm_now = *localtime (&now);

    /* Creer une chaine JJ/MM/AAAA HH:MM:SS */
    char s_now[sizeof "JJ/MM/AAAA HH:MM:SS"];

    strftime (s_now, sizeof s_now, "%d/%m/%Y %H:%M:%S", &tm_now);

    /* afficher le resultat : */
    printf ("%s\n", s_now);

    return 0;
}
```

```
'29/01/2009 02:34:00'
```

```
Process returned 0 (0x0)   execution time : 0.054 s
Press any key to continue.
```

XXXVI - <time.h> : bien utiliser difftime()

Les fonctions de <time.h> offrent une interface assez complexe. Voici un exemple qui rassemble l'usage de la plupart de ces fonctions.

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

int main (void)
{
    time_t now = time (NULL);
    struct tm tm_now = *localtime (&now);
    char s[64];

    strftime (s, sizeof s, "%d/%m/%Y", &tm_now);
    printf ("Aujourd'hui : %s\n", s);

    /* prochain Noel */
    {
        struct tm tm_xmas =
            {0};

        tm_xmas.tm_year = tm_now.tm_year;
        tm_xmas.tm_mon = 12 - 1;
        tm_xmas.tm_mday = 25;

        /* ajustement */
        {
            time_t xmas = mktime (&tm_xmas);

            strftime (s, sizeof s, "%d/%m/%Y", &tm_xmas);
            printf ("Noel : %s\n", s);

            {
                time_t diff = difftime (xmas, now);
                struct tm tm_diff = *gmtime (&diff);

                printf ("Plus que %d jours avant Noel\n", tm_diff.tm_yday);
            }
        }
    }

    return 0;
}
```

XXXVII - rand(), srand()... j'y comprends rien...

Un générateur pseudo-aléatoire est une machine qui génère une séquence de nombres déterminée, cyclique, mais difficile à prévoir pour un humain. De plus, la répartition des valeurs (histogramme) est supposée être équilibrée.

La génération n'est pas 'spontanée', mais 'à la demande' (par appel de la fonction rand()). Les valeurs produites sont comprises entre 0 et RAND_MAX inclus.

A chaque fois que l'on appelle rand(), une nouvelle valeur sort :

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

int main (void)
{
    int i;
    printf ("Les valeurs vont de 0 a %d\n", RAND_MAX);
    for (i = 0; i < 10; i++)
    {
        int val = rand ();
        printf ("%d ", val);
    }
    printf ("\n");
    return 0;
}
```

Par exemple :

```
Les valeurs vont de 0 à 32767
41 18467 6334 26500 19169 15724 11478 29358 26962 24464
```

Mais on constate que si on lance le programme plusieurs fois, la séquence est toujours la même.

On peut modifier l'origine de la séquence avec srand(), en passant une valeur comprise entre 0 et RAND_MAX. Par exemple 10 :

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

int main (void)
{
    int i;

    srand(10); /* MODIF */

    printf ("Les valeurs vont de 0 a %d\n", RAND_MAX);
    for (i = 0; i < 10; i++)
    {
        int val = rand ();
        printf ("%d ", val);
    }
    printf ("\n");
    return 0;
}
```

On constate que les valeurs sont différentes du tirage précédent, mais que si on relance le programme, elles restent identiques :

```
Les valeurs vont de 0 à 32767
71 16899 3272 13694 13697 18296 6722 3012 11726 1899
```

Pour avoir une séquence différente à chaque lancement du programme, il faut donc trouver un moyen de passer une valeur 'changeante' à `srand()`, d'où l'idée d'utiliser la valeur retournée par `time()`, qui change une fois par seconde, indépendamment du programme (c'est une valeur gérée par le système).

Une seconde, c'est long, mais ça suffit pour les besoins courants.

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

int main (void)
{
    int i;


    srand((unsigned) time(NULL)); /* MODIF */

    printf ("Les valeurs vont de 0 a %d\n", RAND_MAX);
    for (i = 0; i < 10; i++)
    {
        int val = rand ();
        printf ("%d ", val);
    }
    printf ("\n");
    return 0;
}
```

Si on lance plusieurs fois le programme (au moins une seconde entre chaque lancement), on obtient maintenant des séquences différentes :

```
25985 16903 23861 29724 17917 23752 17039 25712 20507 30816
26197 27421 5384 20976 236 16846 22741 32047 12417 24408
26433 14874 13653 16830 21990 4658 17461 17892 21603 7731
```

etc. Je précise que pour que les valeurs changent dans le programme, **il faut évidemment que `srand()` ne soit appelé qu'une seule fois au début du programme.**

Après, il y a des astuces arithmétiques pour réduire la plage de valeurs... C'est du bête calcul entier... Détails dans la FAQ de f.c.l.c., notamment  [ici](#). D'autre part, [ceci](#) peut aider.

XXXVIII - Du bon usage de qsort()

XXXVIII-A - Description de la fonction qsort()

XXXVIII-A-1 - Introduction

La fonction `qsort()` implémente un algorithme de tri non spécifié qui permet de trier tout ou partie de n'importe quel tableau de données, du moment qu'il existe un critère de tri dans les données. Elle s'appuie sur une fonction utilisateur qui se charge d'exprimer le critère de tri.

C'est l'implémentation qui décide quel algorithme est utilisé. En général, l'algorithme est choisi pour sa performance. Il n'est pas rare que ce soit un Quick Sort.

XXXVIII-A-2 - Interface

Le prototype de la fonction `qsort()` est

```
void qsort (void *tableau
           , size_t nb_elem
           , size_t taille_elem
           , int (*compare) (void const *a, void const *b));
```

Les paramètres sont :

- `void *tableau` : adresse du premier élément du tableau à trier. La partie à trier doit être modifiable
- `size_t nb_elem` : nombre d'éléments du tableau à trier (à ne pas confondre avec sa taille) : Par exemple, tout le tableau : `sizeof tab / sizeof *tab`
- `size_t taille_elem` : taille d'un élément du tableau : `sizeof *tab`
- `int (*compare) (void const *a, void const *b)` : adresse de la fonction de comparaison (pointeur de fonction). Cette fonction est fournie par l'utilisateur

XXXVIII-A-2-a - Interface de la fonction de comparaison

Cette fonction dispose de 2 paramètres :

- `void const *a` : adresse d'un des éléments du tableau en cours d'évaluation par l'algorithme. Il est qualifié 'const', car la fonction de comparaison ne doit **en aucun cas** en modifier le contenu sous peine de comportement indéfini.
- `void const *b` : adresse d'un autre élément du tableau en cours d'évaluation par l'algorithme. Il est qualifié 'const', car la fonction de comparaison ne doit **en aucun cas** en modifier le contenu sous peine de comportement indéfini.

de plus, elle doit retourner un `int` qui prend la valeur suivante (tri croissant) :

- 0 si le critère de a est égal au critère de b
- < 0 si le critère de a est inférieur au critère de b
- > 0 si le critère de a est supérieur au critère de b

XXXVIII-A-3 - Comportement

La fonction `qsort()` effectue le tri du tableau selon le critère indiqué. La valeur retournée par la fonction de comparaison permet à l'algorithme de prendre les décisions qui s'imposent.

XXXVIII-A-3-a - Comportement de la fonction de comparaison

La fonction de comparaison reçoit l'adresse des 2 éléments en cours d'évaluation. Par l'intermédiaire de pointeurs locaux typés et initialisés avec ces paramètres, elle doit évaluer le critère de tri et retourner un int de la valeur résultant de l'évaluation. Pour un entier, une simple différence suffit. Pour une chaîne, `str[n]cmp()` a été conçue pour ça. Pour un réel, c'est plus délicat, car la notion d'égalité est soumise à l'appréciation d'un EPSILON qui dépend de la précision recherchée.

XXXVIII-B - Usage de la fonction `qsort()`

Afin de dédramatiser l'usage de `qsort()`, qui fait parfois un peu peur, je fournis ici quelques exemples d'utilisation.



Attention, je suppose que les bases du C sont connues (tableaux, pointeurs, fonctions, I/O)



Ces exemples ne sont pas des suites de validation d'algorithmes de tri ! Ce ne sont que de simples exemples.

XXXVIII-B-1 - Tri d'un tableau d'entiers

Soit un tableau de 5 entiers à trier : 4 , 6 , -3 , 4 , 7 ,

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

/* affichage du tableau */
static void aff (int *a, size_t n)
{
    size_t i;
    for (i = 0; i < n; i++)
    {
        printf ("%3d", a[i]);
    }
    printf ("\n");
}

/* fonction utilisateur de comparaison fournie a qsort() */
static int compare (void const *a, void const *b)
{
    /* définir des pointeurs type's et initialise's
       avec les paramètres */
    int const *pa = a;
    int const *pb = b;

    /* évaluer et retourner l'état de l'évaluation (tri croissant) */
    return *pa - *pb;
}

int main (void)
{
    /* tableau a trier */
    int tab[] = { 4, 6, -3, 4, 7 };

    /* affichage du tableau dans l'état */
    aff (tab, sizeof tab / sizeof *tab);

    qsort (tab, sizeof tab / sizeof *tab, sizeof *tab, compare);

    /* affichage du tableau après le tri */
    aff (tab, sizeof tab / sizeof *tab);

    return 0;
}
```

Ce qui donne :

```
 4  6 -3  4  7
-3  4  4  6  7
```

Press ENTER to continue.

XXXVIII-B-2 - Tri d'un tableau de chaines

Il s'agit maintenant de trier un tableau de chaines constantes. Ici, il est particulièrement important de bien comprendre le sens de 'const'. Le tableau est modifiable, mais il est composé de pointeurs sur des chaines non modifiables. Ce que va faire `qsort()`, c'est de réorganiser le tableau de pointeurs de façon à ce lorsqu'on lira le tableau en séquence, les chaines pointées apparaissent comme 'triées'. Ce concept est assez subtil, car le tri est indirect. En effet, si on regardait la valeur des pointeurs dans le tableau, ils ne seraient probablement pas triés. De même, les chaines en mémoire non modifiable sont, par définition, restées à leur place.

Dans la fonction de comparaison, comme d'habitude, on récupère les adresse des éléments en cours d'évaluation. Ici, les éléments sont de type `char const *`. Leur adresse est donc de type `char const **`. Mais comme on a pas le droit de modifier les données pointées, on doit ajouter un 'const' pour être conforme aux paramètres : `char const * const *`

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

/* affichage du tableau */
static void aff (char const **a, size_t n)
{
    size_t i;
    for (i = 0; i < n; i++)
    {
        printf ("%s\n", a[i]);
    }
    printf ("\n");
}

/* fonction utilisateur de comparaison fournie a qsort() */
static int compare (void const *a, void const *b)
{
    /* definir des pointeurs type's et initialise's
       avec les parametres */
    char const *const *pa = a;
    char const *const *pb = b;

    /* evaluer et retourner l'etat de l'evaluation (tri croissant) */
    return strcmp (*pa, *pb);
}

int main (void)
{
    /* tableau a trier (tableau de pointeurs sur char const) */
    char const *tab[] = { "world", "hello", "wild" };

    /* affichage du tableau dans l'etat */
    aff (tab, sizeof tab / sizeof *tab);

    qsort (tab, sizeof tab / sizeof *tab, sizeof *tab, compare);

    /* affichage du tableau apres le tri */
    aff (tab, sizeof tab / sizeof *tab);

    return 0;
}
```

malgré tout, le tri a eu lieu :

```
world
hello
wild

hello
wild
world
```

Press ENTER to continue.

XXXVIII-B-3 - Tri d'un tableau de structures

Soit une structure comprenant {nom, prenom, age}. On crée un tableau de 5 éléments de ce type que l'on remplit "à la main", puis, on fait un tri par prenom, par age décroissant, puis par age croissant.

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

struct fiche
{
    char nom[11];
    char prenom[11];
    int age;
};

/* affichage du tableau */
static void aff (struct fiche const *a, size_t n)
{
    size_t i;
    for (i = 0; i < n; i++)
    {
        /* pointeur intermediaire pour alléger l'écriture */
        struct fiche const *p = a + i;
        printf ("%10s %-10s %d ans\n", p->nom, p->prenom, p->age);
    }
    printf ("\n");
}

/* fonction utilisateur de comparaison fournie a qsort() */
static int compare_prenom (void const *a, void const *b)
{
    /* definir des pointeurs type's et initialise's
       avec les parametres */
    struct fiche const *pa = a;
    struct fiche const *pb = b;

    /* evaluer et retourner l'etat de l'evaluation (tri croissant) */
    return strcmp (pa->prenom, pb->prenom);
}

/* fonction utilisateur de comparaison fournie a qsort() */
static int compare_age (void const *a, void const *b)
{
    struct fiche const *pa = a;
    struct fiche const *pb = b;

    return pa->age - pb->age;
}

/* fonction utilisateur de comparaison fournie a qsort() */
static int compare_age_dec (void const *a, void const *b)
{
    struct fiche const *pa = a;
    struct fiche const *pb = b;

    return pb->age - pa->age;
}
```

```
int main (void)
{
    /* tableau a trier (tableau de pointeurs sur char const) */
    struct fiche tab[] = {
        {"Simpson", "Homer", 36},
        {"Bouvier", "Marge", 34},
        {"Simpson", "Bart", 10},
        {"Simpson", "Lisa", 8},
        {"Simpson", "Maggie", 2},
    };

    /* affichage du tableau dans l'etat */
    aff (tab, sizeof tab / sizeof *tab);

    qsort (tab, sizeof tab / sizeof *tab, sizeof *tab, compare_prenom);

    /* affichage du tableau apres le tri */
    aff (tab, sizeof tab / sizeof *tab);

    qsort (tab, sizeof tab / sizeof *tab, sizeof *tab, compare_age);

    /* affichage du tableau apres le tri */
    aff (tab, sizeof tab / sizeof *tab);

    qsort (tab, sizeof tab / sizeof *tab, sizeof *tab, compare_age_dec);

    /* affichage du tableau apres le tri */
    aff (tab, sizeof tab / sizeof *tab);

    return 0;
}
```

```
Simpson   Homer   36 ans
Bouvier   Marge   34 ans
Simpson   Bart    10 ans
Simpson   Lisa    8 ans
Simpson   Maggie  2 ans
```

```
Simpson   Bart    10 ans
Simpson   Homer   36 ans
Simpson   Lisa    8 ans
Simpson   Maggie  2 ans
Bouvier   Marge   34 ans
```

```
Simpson   Maggie  2 ans
Simpson   Lisa    8 ans
Simpson   Bart    10 ans
Bouvier   Marge   34 ans
Simpson   Homer   36 ans
```

```
Simpson   Homer   36 ans
Bouvier   Marge   34 ans
Simpson   Bart    10 ans
Simpson   Lisa    8 ans
Simpson   Maggie  2 ans
```

Press ENTER to continue.

XXXVIII-B-4 - Tri de nombres réels

Pour trier les nombres réels (float, double), il y a quelques précautions à prendre, car la notion d'égalité n'existe pas. Il faut travailler par plages :

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

double random_range (double min, double max)
```

```
{
    return min + ((max - min) * (rand () / (double) RAND_MAX));
}

static int cmp (void const *a, void const *b)
{
    int ret = 0;
    double const *pa = a;
    double const *pb = b;
    double diff = *pa - *pb;
    if (diff > 0)
    {
        ret = 1;
    }
    else if (diff < 0)
    {
        ret = -1;
    }
    else
    {
        ret = 0;
    }

    return ret;
}

int main (void)
{
    double tab[100];
    int i;
    srand ((int) time (NULL));
    for (i = 0; i < sizeof tab / sizeof *tab; i++)
    {
        tab[i] = random_range (0, 1);
    }

    qsort (tab, sizeof tab / sizeof *tab, sizeof *tab, cmp);

    for (i = 0; i < sizeof tab / sizeof *tab; i++)
    {
        printf ("%8.4f", tab[i]);
    }

    return 0;
}
```

XXXIX - Les identificateurs réservés

Le document de définition du langage C a réservé un certain nombre d'identificateurs réservés. Ils peuvent être utilisés par les implémenteurs (ceux qui écrivent les compilateurs) ou pour des extensions du langage.

Ces identificateurs peuvent apparaître dans les interfaces publiques ou pour réaliser certaines parties des fichiers d'entête (macros, paramètres...). Ils sont choisis de façon à ne pas interférer avec les identificateurs des utilisateurs, sous réserve, bien sûr, que ceux-ci ne les utilisent pas, d'où l'intérêt de cet article.

Les identificateurs réservés sont

Les identificateurs commençant par	suivi de	Exemple valide	Exemple Reservé
"_"	"_A-Z"	_123	_ABC
"is"	"a-z"	is_abc	isabc
"mem"	"a-z"		
"str"	"a-z"		
"to"	"a-z"		
"wcs"	"a-z"		
"E"	"A-Z" ou "0-9"	Eabc	E123 EABC
"LC_"	"A-Z"	LC_abc LC_123 LCABC	LC_ABC
"SIG"	"_A-Z"	SIGabc	SIGABC SIG_ABC

XL - Bien gérer la portée des objets et des fonctions

Le langage C offre par nature un contrôle assez fin de la portée des objets et des fonctions. Cette caractéristique est souvent mal connue, pourtant elle apporte un bénéfice certain, notamment sur le plan de l'organisation du code (conception détaillée).

XL-A - Fonctions

Par défaut, la portée d'une fonction est globale.

```
int function (int a, char *b)
{
}
```

Elle est visible d'un autre module une simple déclaration:

```
int function ();
```

ou mieux, un prototype:

```
int function (int a, char *b);
```

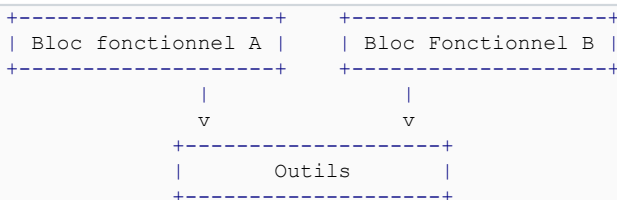
Il est possible cependant de réduire la portée de la fonction à l'unité de compilation dans laquelle elle a été définie, en ajoutant le qualificateur static.

```
static int function (int a, char *b)
{
}
```

Cette pratique, lorsqu'elle est possible, apporte différents avantages :

- Une économie d'identificateurs. La portée de celui-ci étant limitée à une unité de compilation, il est possible de le réutiliser pour une autre fonction qualifiée 'static' dans une autre unité de compilation.
- Une meilleure optimisation. Certains compilateurs sont capables d'"inliner" une telle fonction dans certaines conditions, ce qui diminue le temps d'exécution au prix d'une augmentation de la taille (le code de la fonction est recopié autant de fois que nécessaire).
- Une meilleure organisation du code. Etant donné que ces fonctions sont forcément appelées par une fonction de l'unité de compilation dans laquelle elles ont été définies, le code se trouve naturellement organisé en blocs fonctionnels cohérents. De plus, comme à priori, ces fonctions n'ont pas besoin de prototypes séparés, cela favorise une organisation du fichier source selon le principe 'Définir avant d'utiliser' (*Top-down*)

On évitera cependant de multiplier les codes identiques, et les principes de factorisation du code restent en vigueur.



XL-B - Objets

La portée d'un objet est régie selon plusieurs critères.

XL-B-1 - Définition hors d'un bloc

La portée par défaut est globale. Elle peut être réduite à l'unité de compilation en ajoutant le qualificateur `static`.

XL-B-2 - Définition dans un bloc

Si deux objets ont le même nom, l'objet de portée inférieure masque les objets de portée supérieure. Pour cette raison, qui entraîne un comportement confus, on évite de donner le même nom à des objets dont les portées sont imbriquées.

La portée est celle du bloc et des blocs inclus.

XL-B-3 - Masquage (Shadowing)

```
/* objet de portee globale */
int x;

int f (void)
{
    /* la globale x est masque'e par une locale du meme nom */
    int x = 0;

    /* la globale x n'est pas modifie'e. */
    x++;
}

int main (void)
{
    /* la globale x est modifie'e */
    x = 2;

    f();

    /* la globale x vaut toujours 2 */

    return 0;
}
```


XLI - Du bon usage de assert()

assert() est une macro qui permet de 'poser un piège'. On s'en sert en phase de mise au point pour vérifier si la conception et la réalisation sont correctes.

Le paramètre est une expression. Si elle retourne 0 (expression fausse), le programme s'arrête et un message indiquant le lieu et la cause est affiché.

Il est d'usage qu'en mode production (release), la macro globale NDEBUG soit définie, ce qui fait que les macros assert(), bien que toujours présentes dans le source, ne génèrent plus aucun code de vérification. En conséquence, cette macro ne doit évidemment pas être utilisée pour détecter des erreurs d'utilisation ou de système.

XLI-A - Exemple d'utilisation

```
#include <stdio.h>

static void afficher (int const t[], size_t n)
{
    size_t i;
    for (i = 0; i <= n; i++)
    {
        printf ("%4d", t[i]);
    }
    printf ("\n");
}

int main (void)
{
    int tab[] = {1, 2, 3, 4};

    afficher (tab, sizeof tab / sizeof *tab);
    return 0;
}
```

Ce code parait correct, mais à l'exécution, on constate :

```
1  2  3  4  2
Press ENTER to continue.
```


Pour vérifier le comportement, je pose un piège qui vérifie la validité de l'index.

- il doit être ≥ 0 (toujours vrai, vu le type size_t)
- il doit être $< n$

Je vais donc ajouter un piège :

```
assert (i < n);
```

Avant l'accès en lecture au tableau.

 *Pour être valide, la conception du piège doit se faire sans lire le code à tester, mais en se basant uniquement sur l'interface et le comportement présumé.*

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

static void afficher (int const t[], size_t n)
{
```

```
size_t i;
for (i = 0; i <= n; i++)
{
    /* ajout du piege */
    assert (i < n);
    printf ("%4d", t[i]);
}
printf ("\n");

int main (void)
{
    int tab[] = {1, 2, 3, 4};

    afficher (tab, sizeof tab / sizeof *tab);
    return 0;
}
```

Ce qui provoque bien sûr :

```
1  2  3  4Assertion failed: i < n, file main.c, line 10

This application has requested the Runtime to terminate it in an unusual way.
Please contact the application's support team for more information.

Press ENTER to continue.
```

Ce qui signifie que `i` a dépassé la valeur maximale qu'autorise le langage C. La cause est évidemment le `<=` au lieu de `<` dans l'expression du `for()`, ce qui entraîne une action corrective immédiate :

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

static void afficher (int const t[], size_t n)
{
    size_t i;
    /* correction */
    for (i = 0; i < n; i++)
    {
        /* ajout du piege */
        assert (i < n);
        printf ("%4d", t[i]);
    }
    printf ("\n");
}

int main (void)
{
    int tab[] = {1, 2, 3, 4};

    afficher (tab, sizeof tab / sizeof *tab);
    return 0;
}
```

L'exécution et, à présent, conforme aux attentes :

```
1  2  3  4

Press ENTER to continue.
```

XLII - Comportement indéfini

Le langage C est défini par un document unique et reconnu sur le plan international (ISO) par tous les intervenants, que ce soit les développeurs de compilateurs (les 'implémenteurs') les développeurs d'applications (les 'utilisateurs') ou les différents formateurs.

Ce **document de référence** définit un certain nombre d'éléments (obligations, interdictions).

Les autres éléments sont soit laissés à l'appréciation des implémenteurs (*implementation defined* ou défini par l'implémentation) qui doivent accompagner leur production (compilateur etc.) d'un document précisant les comportement de tel ou tels éléments, soit non définis du tout. Dans ce dernier cas, le comportement est dit indéfini ou indéterminé. (*Undefined Behaviour* ou UB)

Quelques exemples :

```
#include <stdio.h>

int main (void)
{
    int i = 0;

    printf ("i = %d\n", i);
    return 0;
}
```

Ce code est conforme à la spécification du langage, aucune zone n'a été laissée dans l'ombre. Le comportement est déterminé. Il est garanti d'écrire

```
i = 0
```

Par contre, voici 2 cas de comportement indéterminé :

- Absence de prototype pour printf()

```
int main (void)
{
    int i = 0;

    printf ("i = %d\n", i);
    return 0;
}
```

- Lecture d'une valeur non initialisée

```
#include <stdio.h>

int main (void)
{
    int i;

    printf ("i = %d\n", i);
    return 0;
}
```

Les conséquences d'un UB ne sont pas prévisibles. En effet, ça va du crash au comportement d'apparence conforme. Il est donc impossible de compter sur la simple vérification du comportement pour garantir qu'un code est correct. Il faut avant tout qu'il soit exempt de tout UB.

Le compilateur et ses warnings (ou un outil d'analyse spécialisé comme Lint) peut nous aider à débusquer certains UB. Ici, il est probable qu'une 'utilisation de variable non initialisée' ou qu'un 'appel de fonction sans prototypes' soient détectés (mais ça dépend du compilateur et de ses réglages). Mais il est des cas où le compilateur ne voit rien. Le seul recours est alors l'oeil exercé du programmeur expérimenté.

La chasse aux UB est donc ouverte en permanence. C'est la principale source de bugs dans un programme C. Il convient donc, d'une part, de bien connaître le langage et ses limites de définition et, d'autre part, d'être extrêmement vigilant lors de l'écriture et de la relecture du code. Lorsqu'elle est possible, la relecture croisée est une bonne méthode de détection des UB.

Exercice : trouver le UB :

```
#include <stdio.h>

int main (void)
{
    int num = 12;
    char num_text[] = "";

    sprintf (num_text, "%d", num);
    printf ("Voici num_text : %s\n", num_text);
    return 0;
}
```

XLIII - Les item-lists

XLIII-A - Introduction

Qui n'a jamais été confronté à ce problème :

Comment faire le lien entre des constantes symboliques et leur représentation textuelle ?

Le but des item-lists est de résoudre de façon la plus automatique possible ce genre de problème.

XLIII-B - Mise en oeuvre

Le principe est de séparer les informations de bases constantes (par exemple, la correspondance entre caractère et chaîne morse) dans un fichier indépendant (ni.c, ni.h, on y reviendra), mais que l'on peut inclure (`#include`) de façon à réaliser une génération automatique de code en fonction de la demande.

Je prends un exemple plus parlant.

Je veux créer une série de constantes qui représentent des fruits

```
enum fruits
{ BANANE, ORANGE, POMME, FRAISE, KIWI };
```

Ca peut servir à définir une masse de fruit :

```
struct dosage_fruit
{
    enum fruits fruit;
    int masse;
};
```

puis des recettes de fruits mixés :

```
struct dosage_fruit mix_energie[] =
{
    {BANANE, 100},
    {ORANGE, 150},
    {FRAISE, 80},
};

struct dosage_fruit mix_forme[] =
{
    {BANANE, 50},
    {ORANGE, 50},
    {POMME, 100},
    {KIWI, 50},
    {FRAISE, 50},
};

// etc.
```

Si on veut afficher la recette, il faut un moyen simple pour convertir la constante fruit en une chaîne imprimable.

On va donc utiliser un tableau de chaînes construit sur le même modèle que le enum (même ordre, c'est primordial, car l'enum sert d'indice au tableau):

```
static char const *chaines_fruits[] =
{
    "banane",
```

```
"orange",  
"pomme",  
"fraise",  
"kiwi",  
};
```

ce qui permet maintenant d'afficher la composition :

```
void afficher_composition (char const *nom, struct dosage_fruit const *a,  
                           size_t n)  
{  
    size_t i;  
    printf ("%s\n", nom);  
    for (i = 0; i < n; i++)  
    {  
        struct dosage_fruit const *p = a + i;  
        printf ("%d g de %s\n", p->masse, chaines_fruits[p->fruit]);  
    }  
    printf ("\n");  
}
```

que l'on appelle comme ceci :

```
#define N(a) (sizeof(a) / sizeof *(a))  
  
{  
    afficher_composition ("Mix energie", mix_energie, N(mix_energie));  
    afficher_composition ("Mix forme", mix_forme, N(mix_forme));  
  
    etc.
```

Ce qui donne

```
Mix energie  
100 g de banane  
150 g de orange  
80 g de fraise  
  
Mix forme  
50 g de banane  
50 g de orange  
100 g de pomme  
50 g de kiwi  
50 g de fraise  
  
Press ENTER to continue.
```

(je laisse au programmeur malin le soin d'écrire "d'orange" au lieu de "de orange"...)

Maintenant, patatras, nouvelle recette à la carte :

```
struct dosage_fruit mix_tropical[] =  
{  
    {BANANE, 50},  
    {ORANGE, 80},  
    {MANGUE, 100},  
    {ANANAS, 50},  
};
```

Quelles sont les conséquences sur le programme :

2 modifications :

- l'enum :

```
enum fruits
{ BANANE, ORANGE, POMME, FRAISE, KIWI, MANGUE, ANANAS };
```

- la liste des chaines 'associées' (manuellement, pour le moment)

```
static char const *chaines_fruits[] =
{
    "banane",
    "orange",
    "pomme",
    "fraise",
    "kiwi",
    "mangue",
    "ananas",
};
```

Si j'inverse ou que j'en oublie une, c'est la catastrophe. Idem, et c'est beaucoup plus sournois, si j'oublie une ','.

Donc, après quelques sueurs froides, ça marche, et on obtient bien :

```
Mix energie
100 g de banane
150 g de orange
80 g de fraise

Mix forme
50 g de banane
50 g de orange
100 g de pomme
50 g de kiwi
50 g de fraise

Mix tropical
50 g de banane
80 g de orange
100 g de mangue
50 g de ananas

Press ENTER to continue.
```

Mais c'est déjà beaucoup de stress dans un petit programme comme ça. Dans l'industrie, la moyenne, c'est 1 000 000 de lignes... Pas question de se stresser comme ça si, pour ajouter une valeur dans la liste, il faut modifier dans 10 fichiers différents... (Lesquels ? On n'est pas des robots...)

Par contre, on peut utiliser des techniques de programmations qui font que la maintenance est centralisée en un seul fichier. On le modifie, on recompile tout le projet, et les modifications sont automatiquement reportées dans tout le code.

Pour ça, on va faire travailler la machine à partir d'un fichier unique, pour qu'elle produise ce qu'on veut. C'est toute la puissance qu'offre le préprocesseur bien maîtrisé.

Je vais montrer les différentes étapes pour expliquer le principe, mais dans la pratique, on ne fait que la dernière évidemment.

Dans notre exemple, Les deux éléments à "synchroniser" sont :

```
enum fruits
{ BANANE, ORANGE, POMME, FRAISE, KIWI, MANGUE, ANANAS };
```

et la liste des chaines 'associées' (manuellement, pour le moment)

```
static char const *chaines_fruits[] =
```

```
{  
    "banane",  
    "orange",  
    "pomme",  
    "fraise",  
    "kiwi",  
    "mangue",  
    "ananas",  
};
```

Si on observe bien ces deux éléments, on constate qu'ils se ressemblent. Pour être plus parlant, on va les réorganiser en colonnes :

```
enum fruits  
{  
    BANANE,  
    ORANGE,  
    POMME,  
    FRAISE,  
    KIWI,  
    MANGUE,  
    ANANAS  
};
```

et la liste des chaînes 'associées' (manuellement, pour le moment)

```
static char const *chaines_fruits[] =  
{  
    "banane",  
    "orange",  
    "pomme",  
    "fraise",  
    "kiwi",  
    "mangue",  
    "ananas",  
};
```

On voit que le schéma est similaire :

- une entête
- une liste (chaque élément est terminé par une ,)
- une fin particulière.

On voit que la liste des enum présente une petite irrégularité : le dernier élément n'est pas terminé par une ','. C'est normal (et obligatoire) en C90 (en C99, la virgule est acceptée).

Petite astuce : on ajoute un élément à la liste, qui sert à terminer la liste sans virgule. Il ne fait pas partie de la liste. C'est soit un 'dummy' (inutile), soit, comme ici où les valeurs sont automatiques, une constante qui exprime le nombre d'éléments de la liste, ce qui, à priori, n'est pas complètement inutile....

Modification :

```
enum fruits  
{  
    BANANE,  
    ORANGE,  
    POMME,  
    FRAISE,  
    KIWI,  
    MANGUE,  
    ANANAS,  
    NB_FRUITS  
};
```


Afin de faciliter la maintenance, il faudrait disposer d'une liste 'double' comme ceci :

```
BANANE "banane"
ORANGE "orange"
POMME  "pomme"
FRAISE "fraise"
KIWI   "kiwi"
MANGUE "mangue"
ANANAS "ananas"
```

Comment faire comprendre ça à un programme C ?

C'est là qu'intervient la puissance du préprocesseur.

Il suffit d'écrire une liste de macros avec deux paramètres. Chaque macro représentant un groupe cohérent d'informations ou 'item' :

```
ITEM (BANANE, "banane")
ITEM (ORANGE, "orange")
ITEM (POMME , "pomme" )
ITEM (FRAISE, "fraise")
ITEM (KIWI  , "kiwi" )
ITEM (MANGUE, "mangue")
ITEM (ANANAS, "ananas")
```

Il suffit ensuite d'écrire la définition de ITEM qui correspond à l'usage qu'on en fait :

```
enum fruits
{
#define ITEM(id, chaine)\
    id,

ITEM (BANANE, "banane")
ITEM (ORANGE, "orange")
ITEM (POMME , "pomme" )
ITEM (FRAISE, "fraise")
ITEM (KIWI  , "kiwi" )
ITEM (MANGUE, "mangue")
ITEM (ANANAS, "ananas")

#undef ITEM

    NB_FRUITS
};
```

ce qui va produire automatiquement la bonne liste.

On fait pareil avec l'autre liste :

```
static char const *chaines_fruits[] =
{
#define ITEM(id, chaine)\
    chaine,

ITEM (BANANE, "banane")
ITEM (ORANGE, "orange")
ITEM (POMME , "pomme" )
ITEM (FRAISE, "fraise")
ITEM (KIWI  , "kiwi" )
ITEM (MANGUE, "mangue")
ITEM (ANANAS, "ananas")

#undef ITEM
};
```

Le `#undef` permet le 'recyclage' du nom de la macro qui est invariablement `ITEM`.

L'étape ultime consiste à inclure la liste à partir d'un fichier extérieur auquel je donne l'extension `.itm` (par exemple : `fruits.itm` semble approprié).

Je conseille de placer un commentaire dans ce fichier qui rappelle son nom et le début de la définition de la macro avec la signification des champs.

```
/* fruits.itm

#define ITEM(id, chaine)\

*/
ITEM (BANANE, "banane")
ITEM (ORANGE, "orange")
ITEM (POMME , "pomme" )
ITEM (FRAISE, "fraise")
ITEM (KIWI , "kiwi" )
ITEM (MANGUE, "mangue")
ITEM (ANANAS, "ananas")
```

La maintenance de ce fichier est extrêmement simple. Elle est "visuelle".

Les deux définitions deviennent alors :

```
enum fruits
{
#define ITEM(id, chaine)\
    id,
#include "fruits.itm"
#undef ITEM

    NB_FRUITS
};
```

et

```
static char const *chaines_fruits[] =
{
#define ITEM(id, chaine)\
    chaine,
#include "fruits.itm"
#undef ITEM
};
```

Ce qui allège considérablement le code source et rend la maintenance automatique. (Le C, c'est Bien)

Il peut y avoir des centaines de lignes dans un `.itm`. Idem pour le nombre de champs. Ici, il y en a 2 champs, mais on aurait pu en avoir qu'un seul. En effet, une macro sait transformer un paramètre symbolique (`ORANGE`) en une chaîne ("`ORANGE`") avec `#`. Mais elle ne sait pas modifier la casse des caractères, d'où mon choix de mettre 2 champs.

Exemples de fichiers itm que j'utilise

(Simple) : **gestion des erreurs**

(Complexe) : **tables de caractères**

Je laisse au lecteur le soin d'écrire l'ensemble de l'exemple et de faire les tests nécessaires. Tous les éléments sont là.

XLIV - Borland C : "floating point formats not linked"

Un programme généré avec l'IDE Borland C++ 3.1 signale parfois ce message à l'exécution:

```
scanf : floating point formats not linked
Abnormal program termination
```

Il s'agit en fait d'un bug connu de certains compilateurs Borland. Il se produit lorsqu'on utilise un format 'flottant' avec `*printf()` ou `*scanf()`, et qu'on n'utilise pas de fonction de la bibliothèque mathématique.

La parade est simple. Il suffit d'ajouter ces quelques lignes dans le code source contenant le `main()`, par exemple.

```
#ifdef __BORLANDC__
/* The pesky "floating point formats not linked" killer hack : */
extern unsigned _floatconvert;
#pragma extref _floatconvert
#endif
```

XLV - Code standard ? Code portable ? Je suis perdu !

On entend parler de code standard, de code portable ? Qu'est-ce que ça signifie ? Ca sert à quoi ?

XLV-A - "standard" ?

Le terme 'standard' est erroné. C'est un anglicisme de 'standard' qui signifie 'norme' (subst.) ou 'normalisé' (adj.).

Le langage C est normalisé. Cela signifie en clair qu'il est défini par un document spécifié et publié sous la responsabilité de l'ISO (*International Standard Organisation* ou Organisme de normalisation international). Ce document décrit la syntaxe et la sémantique du langage ainsi que l'interface et le comportement des fonctions de la bibliothèque d'exécution (RTL ou *Run-Time Library*).

Cette spécification s'applique aux compilateurs réputés 'conformes à la norme' et par conséquent aux programmeurs qui les utilise. La spécification définit en gros trois domaines :

- Ce qui est défini par la norme
- Ce qui est défini par la cible^[1]
- Ce qui n'est pas défini

Ce qui n'est pas défini par la norme peut l'être par une implémentation du C qui comporte des extensions spécifiques à une cible ou à un système. (Mots clés, fonctions, bibliothèques)

Par exemple `system()` est une fonction normalisée, dont le paramètre est une chaîne de caractères. Cependant, la sémantique du texte porté par cette chaîne de caractères peut varier d'un système à l'autre, voire ne pas être reconnue du tout par le système.

[1] (on dit aussi implémentation (anglicisme), implantation ou plateforme)

XLV-B - "portable" ?

C'est la capacité qu'a un code source à produire un comportement identique sur différentes plateformes. On distingue la portabilité absolue (pour n'importe quelle plateforme) de la portabilité relative (limitée à un certain nombre de plateformes bien définies).

XLV-B-1 - portabilité absolue

C'est lorsqu'un code source ne contient que des éléments normalisés du langage dont la définition et l'utilisation ne dépendent pas de la plateforme. Certaines pratiques additionnelles peuvent cependant rendre portable du code standard, comme ajouter `fflush(stdout)` après un `printf()` qui ne se termine pas par un `'\n'`.

XLV-B-2 - portabilité relative

C'est un code portable 'absolu' auquel s'ajoutent des extensions (généralement, des bibliothèques) tierces conçues pour fonctionner sur un certain nombre de plateformes bien définies.

La norme POSIX.1 en définit un certain nombre, notamment en matière de gestion des répertoires, processus légers (threads) et réseau (sockets). Mais il existe des initiatives indépendantes comme GTK+ qui définit une interface de programmation graphique pour utilisateur (GUI) commune à Windows, X (Unix/Linux), Apple/Mac et même BeOS.

XLV-C - Bon usage

Le fait de bien connaître les domaines couverts par le langage C constitue une aide considérable pour l'écriture du code. En effet, la portabilité n'est possible que si le code est portable et donc, dans sa grande majorité normalisé, ou tout au moins conforme aux définitions de telle ou telle bibliothèque d'abstraction.

Il est donc impératif de séparer le code normalisé (qui doit représenter la majorité de celui-ci) du code spécifique à telle ou telle plateforme, et qui n'aurait pas trouvé sa place dans les bibliothèques d'abstraction.

XLVI - Enregistrer une structure

L'enregistrement d'une structure dans un fichier est une opération plus complexe qu'il n'y paraît. En effet, le code naïf suivant :

```
#include <stdio.h>

struct data
{
    char nom[32];
    int age;
};

int main (void)
{
#define FNAME "data.txt"

    struct data data = { "Emmanuel", 50 };

    FILE *fp = fopen (FNAME, "wb");
    if (fp != NULL)
    {
        fwrite (&data, sizeof data, 1, fp);

        fclose (fp), fp = NULL;
    }
    else
    {
        perror (FNAME);
    }
    return 0;
}
```

est certes simple et efficace, mais est malheureusement non portable, et ce pour plusieurs raisons :

- La représentation interne des données en C peut changer d'une implémentation à l'autre, en terme de largeur (nombre de bits), de 'boutisme' (position du byte de poids fort) et de codage (entiers négatifs, nombres réel, jeu de caractères).
- L'alignement requis. En effet, certaines architectures imposent que les éléments de la structure soient alignés sur une adresse multiple de 2, 4 ou autre, ce qui rend impossible de prévoir de façon portable, la signification des bytes dans le fichier.

Pour résoudre ce problème, il y a 2 grandes familles de solutions :

- Le format binaire
- Le format texte

XLVI-A - Le format binaire

C'est le plus portable, mais aussi le plus complexe. Il consiste à définir un format de données indépendant de toute implémentation. Il nécessite une conversion à l'écriture (host->file) et une conversion à la lecture (file->host), et ce dans le strict respect du format 'fichier' spécifié. Une méthode simple est TLV (Type, Longueur, Valeur ou Type, Length, Value).

(à venir : exemple de spécification TLV)

Il existe des solutions normalisées comme **BER** (*Basic Encoding Rules*) spécifié par les recommandations ITU-T X.209 et X.690 ou **XDR** (*eXternal Data Representation*) spécifié par la **RFC 1832** plus ou moins basées sur TLV. Ces solutions sont complexes et sont plutôt utilisées avec l'aide d'une bibliothèque tierce comme **BER sous Linux**.

Dans tous les cas, le fichier est traité en mode binaire ("wb", "rb", "ab"), et les fonctions les plus utilisées sont fgetc(), fputc(), fread() et fwrite().

XLVI-B - Le format texte

Il est un peu moins portable, mais moins complexe. Il consiste à définir un format de données indépendant sous forme de texte. Il nécessite une conversion à l'écriture (host->file) et une conversion à la lecture (file->host), et ce dans le strict respect du format 'fichier' spécifié. Une méthode simple est CSV (*Comma Separated Values*).

On peut trouver les spécifications sur [Wotsit](#), le site de référence des formats de fichiers.

Le fichier est traité le plus souvent en mode texte ("w", "r", "a"), mais aussi parfois en mode binaire pour régler les problèmes de fins de ligne hétérogènes. (voir plus loin). Les fonctions les plus utilisées sont fgetc(), fputc(), fgets(), fputs() et fprintf().

Les principaux problèmes de portabilité proviennent :

- Du jeu de caractères utilisé. En dehors d'EBCDIC utilisé sur les gros systèmes IBM (*Mainframes*), c'est généralement ASCII (0-127), mais ça peut ne pas suffire à encoder tous les caractères, notamment les caractères accentués et autres signes mathématiques ou pseudo-graphiques. Or il existe plusieurs façons de coder ces caractères (ASCII étendu, IBM-PC8, UTF-8, Unicode etc.). Là encore, une définition indépendante peut aider...
- De la façon de coder les fins de ligne (CR, LF, CRLF etc.)

Certaines corrections doivent donc parfois être effectuées à l'aide de tables de codages et autres astuces algorithmiques.