Université Pierre et Marie Curie
DEUST Informatique
Année 2002–2003

Module de programmation
C. Gonzales 2002

Cours de langage C : Memento et exercices

Table des matières

1 Memento **2**
1.1 Les mots réservés du langage . . . 2
1.2 Les types de base . . . 2
1.3 Étendue des variables et constantes importantes . . . 4
1.4 Visibilité des variables . . . 6
1.5 Règles de conversion . . . 6
1.6 Utilisation de `printf` et de `scanf` . . . 7
1.7 Opérateurs et expressions . . . 8
1.8 Instructions . . . 10
1.9 Instructions de contrôle . . . 10
1.10 Ruptures de séquences . . . 12
1.11 Les fonctions . . . 14
1.12 Pointeurs et tableaux . . . 16
1.13 Structures, unions et énumérations . . . 19
1.14 Entrée/Sorties . . . 21
1.15 Quelques règles de bonne conduite . . . 28
1.15.1 pagination . . . 28
1.15.2 Les fonctions . . . 29
1.15.3 Écriture du code . . . 30
1.15.4 Compilation et préprocesseur . . . 33

2 Exercices **34**
2.1 Syntaxe, types, initialisations et conversions . . . 34
2.2 Éléments de base . . . 36
2.3 Opérateurs et expressions . . . 38
2.4 Instructions de contrôle . . . 40
2.5 fonctions . . . 46
2.6 Pointeurs et tableaux . . . 49
2.7 structures . . . 54
2.8 Entrées/Sorties . . . 62
2.9 Préprocesseur et lignes de compilation . . . 66

1 Memento

Comme je vous l'avais dit lors du premier cours, j'utilise comme support le cours de Christian Bac (si, si, je vous l'avais dit, de toute façon, j'ai les noms de ceux qui ne suivent pas). Le memento ci-après est donc une compilation de son cours, que je vous ai par ailleurs installé sur vos machines dans le répertoire consacré à Visual C++.

1.1 Les mots réservés du langage

type d'identificateur	mots révervés
structures de données	char const double float int long short signed unsigned void volatile
classes d'allocation	auto extern register static
constructeurs	enum struct typedef union
instructions de boucle	do for while
sélections	case default else if switch
ruptures de séquence	break continue goto return
divers	asm entry fortran sizeof

TAB. 1: Liste des mots réservés du langage

1.2 Les types de base

Type	Description
`void`	C'est le type vide. Il a été introduit par la norme ANSI. Il est surtout utilisé pour préciser les fonctions sans argument ou sans retour. Il joue un rôle particulier dans l'utilisation des pointeurs.
`int`	C'est le type entier. Ce type peut se décliner avec des qualificateurs pour préciser sa taille (long ou short), et le fait qu'il soit uniquement positif (unsigned) ou positif et négatif (signed). Le qualificateur signed est appliqué par défaut, ainsi il n'y a pas de différence entre une variable de type int et une variable de type signed int.
`char`	Ce type est très proche de l'octet. Il représente un entier sur huit bits. Sa valeur peut évoluer entre -128 et +127. Il est le support des caractères au sens commun du terme. Ces caractères sont représentés par la table ASCII. Comme le type int, le type char peut être qualifié de manière à être signé ou non signé. La norme ANSI introduit un type permettant de supporter des alphabets comprenant plus de 255 signes, ce type est appelé wchar_twchar_t. Il est défini dans le fichier `<stddef.h>`.
`float`	Ce type sert pour les calculs réels (avec des parties décimales).
`double`	C'est un type qui permet de représenter des valeurs ayant une partie décimale avec une plus grande précision que le type float. Il est le type le plus couramment utilisé pour représenter des nombres réels (cf. les fonctions mathématiques sinus, cosinus, etc).
`long double`	Ce type est récent, il permet de représenter des nombres réels qui nécessitent une très grande précision.

TAB. 2: Liste des types de base du langage

Les constantes de type caractère simple (char) sont toujours entourées d'apostrophes (simples quotes). Elles peuvent être représentées selon quatre méthodes décrites dans le tableau ci-dessous :

1.	Excepté pour l'antislash et l'apostrophe, lorsque le caractère est disponible au clavier, le caractère correspondant est donné par 'caractère', par exemple 'a'.
2.	un caractère peut aussi être représenté par sa valeur exprimée dans la table ASCII en utilisant une notation en octal (base huit). Cette valeur est précédée à l'intérieur des apostrophes par un antislash : '\0' : octet nul, il sert à délimiter les fins de chaînes de caractères. '\012' : saut de ligne (Line Feed, LF). '\015' : retour chariot (Carriage Return, CR). '\011' : tabulation horizontale (Horizontal Tabulation, HT).
3.	de même, un caractère peut être représenté par sa notation en hexadécimal : '\x0' : caractère nul. '\xA' : saut de ligne. '\xD' : retour chariot. '\x9' : tabulation horizontale.
4.	un certain nombre d'abréviations sont aussi disponibles : '\a' : alert (sonnerie, BEL). '\b' : backspace (BS). '\f' : saut de page (FF). '\n' : fin de ligne (LF) . '\r' : retour chariot (CR). '\t' : tabulation horizontale (HT). '\v' : tabulation verticale (VT).

Type	DEC PDP 11	PC 486	SUN4	Alpha
char	8 bits	8 bits	8 bits	8 bits
short	16 bits	16 bits	16 bits	16 bits
int	16 bits	16 bits	32 bits	32 bits
long	32 bits	32 bits	32 bits	64 bits
float	32 bits	32 bits	32 bits	32 bits
double	64 bits	64 bits	64 bits	64 bits
long double	64 bits	64 bits	64 bits	128 bits

TAB. 3: Longueur des types de base sur quelques machines

1.3 Étendue des variables et constantes importantes

```
/* These assume 8-bit ‘char’s, 16-bit ‘short int’s,
   and 32-bit ‘int’s and ‘long int’s.  */

/* Number of bits in a ‘char’.  */
#  define CHAR_BIT      8

/* Minimum and maximum values a ‘signed char’ can hold.  */
#  define SCHAR_MIN     (-128)
#  define SCHAR_MAX     127

/* Maximum value an ‘unsigned char’ can hold.  (Minimum is 0.)  */
#  define UCHAR_MAX     255

/* Minimum and maximum values a ‘char’ can hold.  */
#  define CHAR_MIN     SCHAR_MIN
#  define CHAR_MAX     SCHAR_MAX

/* Minimum and maximum values a ‘signed short int’ can hold.  */
#  define SHRT_MIN      (-32768)
#  define SHRT_MAX      32767

/* Maximum value an ‘unsigned short int’ can hold.  (Minimum is 0.)  */
#  define USHRT_MAX     65535

/* Minimum and maximum values a ‘signed int’ can hold.  */
#  define INT_MIN       (-INT_MAX - 1)
#  define INT_MAX       2147483647

/* Maximum value an ‘unsigned int’ can hold.  (Minimum is 0.)  */
#  define UINT_MAX     4294967295U

/* Minimum and maximum values a ‘signed long int’ can hold.  */
#  define LONG_MAX     2147483647L
#  define LONG_MIN      (-LONG_MAX - 1L)

/* Maximum value an ‘unsigned long int’ can hold.  (Minimum is 0.)  */
#  define ULONG_MAX   4294967295UL
```

TAB. 4 : exemple de fichier *limits.h*

```
/* Radix of exponent representation */
#define FLT_RADIX 2
/* Number of base-FLT_RADIX digits in the significand of a float */
#define FLT_MANT_DIG 24
/* Number of decimal digits of precision in a float */
#define FLT_DIG 6
/* Addition rounds to 0: zero, 1: nearest, 2: +inf, 3: -inf, -1: unknown */
#define FLT_ROUNDS 1
/* Difference between 1.0 and the minimum float greater than 1.0 */
#define FLT_EPSILON 1.19209290e-07F
/* Minimum int x such that FLT_RADIX**(x-1) is a normalised float */
#define FLT_MIN_EXP (-125)
/* Minimum normalised float */
#define FLT_MIN 1.17549435e-38F
/* Minimum int x such that 10**x is a normalised float */
#define FLT_MIN_10_EXP (-37)
/* Maximum int x such that FLT_RADIX**(x-1) is a representable float */
#define FLT_MAX_EXP 128
/* Maximum float */
#define FLT_MAX 3.40282347e+38F
/* Maximum int x such that 10**x is a representable float */
#define FLT_MAX_10_EXP 38

/* Number of base-FLT_RADIX digits in the significand of a double */
#define DBL_MANT_DIG 53
/* Number of decimal digits of precision in a double */
#define DBL_DIG 15
/* Difference between 1.0 and the minimum double greater than 1.0 */
#define DBL_EPSILON 2.2204460492503131e-16
/* Minimum int x such that FLT_RADIX**(x-1) is a normalised double */
#define DBL_MIN_EXP (-1021)
/* Minimum normalised double */
#define DBL_MIN 2.2250738585072014e-308
/* Minimum int x such that 10**x is a normalised double */
#define DBL_MIN_10_EXP (-307)
/* Maximum int x such that FLT_RADIX**(x-1) is a representable double */
#define DBL_MAX_EXP 1024
/* Maximum double */
#define DBL_MAX 1.7976931348623157e+308
/* Maximum int x such that 10**x is a representable double */
#define DBL_MAX_10_EXP 308

/* Number of base-FLT_RADIX digits in the significand of a long double */
#define LDBL_MANT_DIG 64
/* Number of decimal digits of precision in a long double */
#define LDBL_DIG 18
```

TAB. 5 : exemple de fichier *float.h*

1.4 Visibilité des variables

Une définition de variable est l'association d'un identificateur à un type et la spécification d'une classe mémoire. La classe mémoire sert à expliciter la visibilité de la variable et son implantation en machine. Les classes mémoire sont :

classe	Description
`global`	cette classe est celle des variables définies en dehors d'une fonction. Ces variables sont accessibles à toutes les fonctions. La durée de vie des variables de type global est la même que celle du programme en cours d'exécution.
`local` ou `auto`	cette classe comprend l'ensemble des variables déclarées dans un bloc. C'est le cas de toute variable déclarée à l'intérieur d'une fonction. L'espace mémoire réservé pour ce type de variable est alloué dans la pile d'exécution. C'est pourquoi elles sont appelées aussi automatiques car l'espace mémoire associé est créé lors de l'entrée dans la fonction et il est détruit lors de la sortie de la fonction. La durée de vie des variables de type local est celle de la fonction dans laquelle elles sont définies.
`static`	ce prédicat modifie la visibilité de la variable, ou son implantation. Dans le cas d'une variable locale, il modifie son implantation en attribuant une partie de l'espace de mémoire global pour cette variable. Une variable locale de type statique a un nom local mais a une durée de vie égale à celle du programme en cours d'exécution. Dans le cas d'une variable globale, ce prédicat restreint la visibilité du nom de la variable à l'unité de compilation. Une variable globale de type statique ne peut pas être utilisée par un autre fichier source participant au même programme grâce à une référence avec le mot réservé extern (voir point suivant).
`extern`	ce prédicat permet de spécifier que la ligne correspondante n'est pas une tentative de définition mais une déclaration. Il précise les variables globales (noms et types) qui sont définies dans un autre fichier source et qui sont utilisées dans ce fichier source.
`register`	ce prédicat permet d'informer le compilateur que les variables locales définies dans le reste de la ligne sont utilisées souvent. Le prédicat demande de les mettre si possible dans des registres disponibles du processeur de manière à optimiser le temps d'exécution. Le nombre de registres disponibles pour de telles demandes est variable selon les machines. Il est de toute façon limité (4 pour les données, 4 pour les pointeurs sur un 680X0). Seules les variables locales peuvent être déclarées register.

TAB. 6: Liste des classes mémoire

1.5 Règles de conversion

Le compilateur fait de lui-même des conversions lors de l'évaluation des expressions. Pour cela il applique des règles de conversion implicite. Ces règles ont pour but de perdre un minimum d'information dans l'évaluation de l'expression. Ces règles sont décrites dans la table ci-dessous :

Règle de Conversion Implicite :
Convertir les éléments de la partie droite d'une expression dans le type de la variable la plus riche. Faire les calculs dans ce type. Puis convertir le résultat dans le type de la variable affectée.

La notion de richesse d'un type est précisée dans la norme. Le type dans lequel un calcul d'une expression à deux opérandes doit se faire est donné par les règles suivantes :

1. si l'un des deux opérandes est du type long double alors le calcul doit être fait dans le type long double.
2. sinon, si l'un des deux opérandes est du type double alors le calcul doit être fait dans le type double.
3. sinon, si l'un des deux opérandes est du type float alors le calcul doit être fait dans le type float.
4. sinon, appliquer la règle de promotion en entier, qui précise que lorsqu'on utilise des variables ou des constantes de type caractère, champ de bits ou énumération dans une expression, alors les valeurs de ces variables ou constantes sont transformées en leur équivalent en entier avant de faire les calculs (ceci permet d'utiliser des caractères, des entiers courts, des champs de bits et des énumérations de la même façon que des entiers). Lorsque la promotion est réalisée, appliquer les règles suivantes :
   (a) si l'un des deux opérandes est de type unsigned long int alors le calcul doit être fait dans ce type.
   (b) si l'un des deux opérandes est de type long int alors le calcul doit être fait dans le type long int.
   (c) si l'un des deux opérandes est de type unsigned int alors le calcul doit être fait dans le type unsigned int.
   (d) si l'un des deux opérandes est de type int alors le calcul doit être fait dans le type int.

TAB. 7: Conversion des opérandes d'une expression

1.6 Utilisation de `printf` et de `scanf`

Pour `printf()`, un format est une chaîne de caractères dans laquelle sont insérés les caractères représentant la ou les variables à écrire.
Pour chaque variable, un type de conversion est spécifié. Ce type de conversion est donné par les caractères qui suivent un caractère "%" (voir ci-après).
Pour `scanf()`, un format est une chaîne de caractères qui décrit la ou les variables à lire.

TAB. 8: Qu'est-ce qu'un format ?

`%d`	entier décimal
`%f`	flottant
`%c`	caractère (1 seul)
`%s`	chaîne de caractères

TAB. 9: Formats usuels de `printf` et de `scanf`

déclaration	lecture	écriture	format externe
`int i ;`	`scanf("%d",&i) ;`	`printf("%d",i) ;`	décimal
`int i ;`	`scanf("%o",&i) ;`	`printf("%o",i) ;`	octal
`int i ;`	`scanf("%x",&i) ;`	`printf("%x",i) ;`	hexadécimal
`unsigned int i ;`	`scanf("%u",&i) ;`	`printf("%u",i) ;`	décimal
`short j ;`	`scanf("%hd",&j) ;`	`printf("%d",j) ;`	décimal
`short j ;`	`scanf("%ho",&j) ;`	`printf("%o",j) ;`	octal
`short j ;`	`scanf("%hx",&j) ;`	`printf("%x",j) ;`	hexadécimal
`unsigned short j ;`	`scanf("%hu",&j) ;`	`printf("%u",j) ;`	décimal
`long k ;`	`scanf("%ld",&k) ;`	`printf("%d",k) ;`	décimal
`long k ;`	`scanf("%lo",&k) ;`	`printf("%o",k) ;`	octal
`long k ;`	`scanf("%lx",&k) ;`	`printf("%x",k) ;`	hexadécimal
`unsigned long k ;`	`scanf("%lu",&k) ;`	`printf("%u",k) ;`	décimal
`float l ;`	`scanf("%f",&l) ;`	`printf("%f",l) ;`	point décimal
`float l ;`	`scanf("%e",&l) ;`	`printf("%e",l) ;`	exponentielle
`float l ;`		`printf("%g",l) ;`	la plus courte des deux
`double m ;`	`scanf("%lf",&m) ;`	`printf("%f",m) ;`	point décimal
`double m ;`	`scanf("%le",&m) ;`	`printf("%e",m) ;`	exponentielle
`double m ;`		`printf("%g",m) ;`	la plus courte
`long double n ;`	`scanf("%Lf",&n) ;`	`printf("%Lf",n) ;`	point décimal
`long double n ;`	`scanf("%Le",&n) ;`	`printf("%Le",n) ;`	exponentielle
`long double n ;`		`printf("%Lg",n) ;`	la plus courte
`char o ;`	`scanf("%c",&o) ;`	`printf("%c",o) ;`	caractère
`char p[10] ;`	`scanf("%s",p) ;`	`printf("%s",p) ;`	chaîne de caractères
	`scanf("%s",&p[0]) ;`		

TAB. 10: Exemple d'utilisation des formats

1.7 Opérateurs et expressions

Opérateur	Utilisation
`&`	opérateur d'adresse
`*`	opérateur d'indirection sur une adresse
`--`	opérateur de décrémentation
`++`	opérateur d'incrémentation
`sizeof`	opérateur donnant la taille en octet
`!`	NOT logique, il sert à inverser une condition
`-`	moins unaire : inverse le signe
`+`	plus unaire : sert à confirmer
`~`	complément à un

TAB. 11: Liste des opérateurs unaires

Type d'opérateurs	Opérateurs	Usage

Arithmétique	`+ -`
	`* /`
	`%`
		addition, soustraction,
		multiplication, division,
		reste de la division entière.

Masquage	`& | ^`	et, ou, ou exclusif
Décalage	`>> <<`	vers la droite ou vers la gauche

Relation	`< <=`
	`> >=`
	`== !=`
		inférieur, inférieur ou égal,
		supérieur, supérieur ou égal,
		égal, non égal

Logique	`&& ||`	et logique, ou logique
Affectation	`=`	affectation

TAB. 12: Liste des opérateurs binaires

arithméthique	`+= -= *= /= %=`
masquage	`&= |= ^=`
décalage	`>>= <<=`

TAB. 13: Liste des opérateurs binaires d'affectation

expression	résultat	équivalence	lecture
opérateurs arithmétiques			
`i += 10`	110	`i = i + 10`	ajoute 10 à i
`i += j`	115	`i = i + j`	ajoute j à i
`i -= 5`	110	`i = i - 5`	retranche 5 à i
`i -= j`	105	`i = i - j`	retranche j à i
`i *= 10`	1050	`i = i * 10`	multiplie i par 10
`i *= j`	5250	`i = i * j`	multiplie i par j
`i /= 10`	525	`i = i / 10`	divise i par 10
`i /= j`	105	`i = i / j`	divise i par j
`i %= 10`	5	`i = i % 10`	i reçoit le reste de la division entière de i par 10
opérateurs de masquage			
`i &= 8`	0	`i = i & 8`	ET de i avec 8
`i |= 8`	8	`i = i | 8`	OU de i avec 8
`i ^= 4`	0x0C	`i = i ^ 4`	OU exclusif de i avec 4
opérateurs de décalage			
`i <<= 4`	0xC0	`i = i << 4`	décale i à gauche de 4 positions
`i >>= 4`	0x0C	`i = i >> 4`	décale i à droite de 4 positions

TAB. 14: Exemples d'opérateurs binaires d'affectation avec `int i = 100, j = 5;`

opérateur	usage
ex1 ? ex2 : ex3	retourne ex2 si ex1 est vrai retourne ex3 si ex1 est faux

TAB. 15: opérateur ternaire

RÈGLE de précédence :
La priorité est décroissante de haut en bas selon le tableau ci-dessous. Lorsque deux opérateurs se trouvent dans la même ligne du tableau, la priorité d'évaluation d'une ligne de C dans laquelle se trouvent ces opérateurs, est donnée par la colonne de droite (associativité).

Opérateur	Associativité
`() [] -> .`	de gauche à droite
`! - + ++ -- ~ (type) * & sizeof`	de droite à gauche
`* / %`	de gauche à droite
`+ -`	de gauche à droite
`<< >>`	de gauche à droite
`< <= > >=`	de gauche à droite
`== !=`	de gauche à droite
`&`	de gauche à droite
`^`	de gauche à droite
`|`	de gauche à droite
`&&`	de gauche à droite
`||`	de gauche à droite
`? :`	de droite à gauche
`= += -= etc`	de droite à gauche
`,`	de gauche à droite

TAB. 16: Précédence des opérateurs

1.8 Instructions

Une instruction est :
- soit une instruction simple,
- soit un bloc.

Une instruction simple est :
- soit une instruction de contrôle,
- soit une expression suivie de «;».

Une instruction simple est toujours terminée par un «;».
Un bloc a la structure suivante :
- une accolade ouvrante «{»
- une liste de définitions locales au bloc (optionnelle)
- une suite d'instructions
- une accolade fermante «}».

TAB. 17: Qu'est-ce qu'une instruction en C ?

1.9 Instructions de contrôle

```
if (condition) instruction

if (condition) instruction1
else instruction2
```

TAB. 18: Syntaxe du `if`

```
switch (expression)
  {
    case value1:
      inst 10
      inst 11
      inst 12
    case value2:
      inst 20
      inst 21
      etc.
      break;
    case valueN:
      inst N0
      inst N1
      inst N2
    default:
      inst 30
      inst 31
      inst 32
  }
```

TAB. 19: Syntaxe du `switch`

```
while (condition) instruction
```

TAB. 20: Syntaxe du `while`

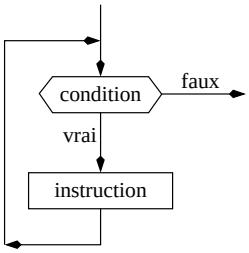


FIG. 1: organigramme de la boucle while

La syntaxe du `for` est la suivante :

`for(expression1 ; expression2 ; expression3) instruction`

Le for s'utilise avec trois expressions, séparées par des **points virgules**, qui peuvent, éventuellement, être vides :

- L'expression `expression1` est une instruction d'initialisation. Elle est exécutée avant l'entrée dans la boucle.
- L'expression `expression2` est la condition de passage. Elle est testée avant chaque passage dans la boucle, y compris le premier. Si sa valeur est vraie (`true`), l'instruction est exécutée, sinon la boucle se termine.
- L'expression `expression3` est une instruction de rebouclage. Elle fait avancer la boucle. Elle est exécutée en fin de boucle avant le nouveau test de passage.

TAB. 21: La boucle `for`

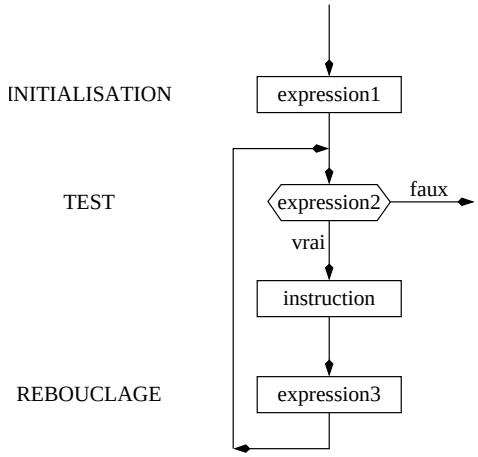


FIG. 2: organigramme de la boucle for

```
do instruction while (condition)
```

TAB. 22: Syntaxe du `do-while`

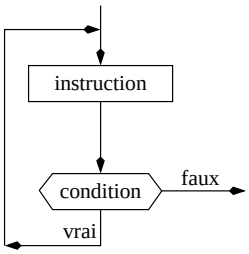


FIG. 3: organigramme de la boucle do–while

1.10 Ruptures de séquences

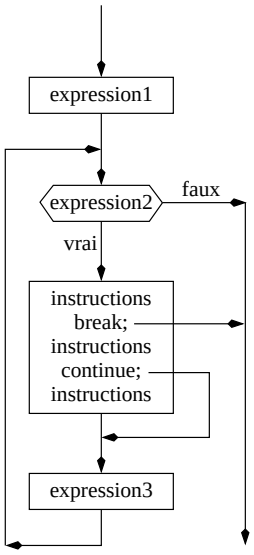


FIG. 4: organigramme des break et continue dans la boucle for

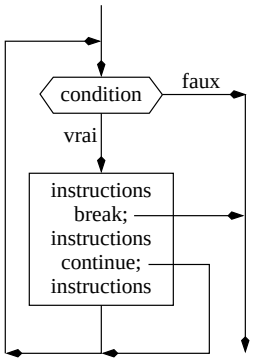


FIG. 5: organigramme des break et continue dans la boucle while

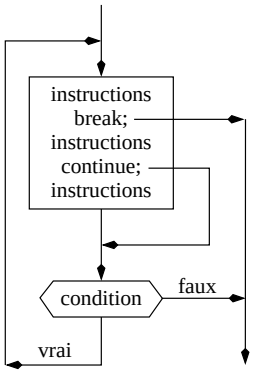


FIG. 6: organigramme des break et continue dans la boucle do–while

1.11 Les fonctions

1/ **Mise en pile des paramètres :** En langage C, les passages de paramètres se font par valeur, c'est-à-dire que la fonction appelante fait une copie de la valeur passée en paramètre et passe cette copie à la fonction appelée à l'intérieur d'une variable créée dans l'espace mémoire géré par la pile d'exécution. La fonction appelante empile les copies des paramètres. Si le paramètre est une constante (entière, flottante ou pointeur), le programme empile une copie de cette constante et l'on obtient une variable de type compatible. Ceci explique pourquoi vous pouvez passer une constante et manipuler celle-ci comme une variable dans la fonction appelée. En C ANSI vous avez cependant la possibilité de dire que la fonction considère ses arguments comme constants. Si les paramètres sont des variables, la fonction appelante empile une copie constituée à partir de la valeur courante de la variable. Ceci explique pourquoi la valeur d'un argument formel peut être différente de celle de son argument réel.
2/ **Saut à la fonction appelée :** La fonction appelante provoque un saut à l'adresse de début de la fonction appelée en empilant l'adresse de retour.
3/ **Prologue dans la fonction appelée :** La fonction appelée prépare son environnement. En particulier, elle positionne son pointeur de contexte dans la pile en sauvegardant l'ancien pointeur de contexte. Ce pointeur de contexte servira pendant toute l'exécution de la fonction à retrouver d'un côté les arguments de la fonction, de l'autre les variables locales à la fonction. Elle fait ensuite grandir la pile de manière à pouvoir ranger une copie des registres qu'elle va utiliser et les variables locales qui ne sont pas en registre. Cette étape est appelée le prologue de la fonction. Ce prologue dépend du type du processeur et des choix de réalisation faits par les concepteurs du compilateur.
4/ **La fonction appelée s'exécute :** jusqu'à rencontrer un return en langage C, ce return provoque le passage à l'épilogue dans la fonction appelée. Lorsque le return est associé à une valeur, celle-ci est conservée dans un registre de calcul (qui sert aussi pour évaluer les expressions).
5/ **Épilogue dans la fonction appelée :** L'épilogue fait le travail inverse du prologue, à savoir il restitue le contexte de la fonction appelante au niveau des registres du processeur (sauf les registres qui contiennent la valeur de retour de la fonction). Pour cela, l'épilogue restaure les registres qu'il avait sauvegardés (correspondant aux registres demandés par les définitions de variables de type register), puis il restaure le contexte de pile en reprenant son ancienne valeur dans la pile. Enfin, il fait diminuer la pile conformément à la réservation qu'il avait fait dans le prologue. Finalement, il retourne à la fonction appelante.
6/ **Récupération du résultat et effacement des paramètres :** la fonction appelante dépile les paramètres qu'elle avait empilés au début de l'appel et utilise la (ou les) valeur(s) de retour de la fonction pour calculer l'expression courante dans laquelle la fonction a été appelée.

TAB. 23: Étapes d'un appel de fonction

Lors de l'appel d'une fonction, les paramètres subissent des conversions de type. Les plus importantes sont les suivantes : • les paramètres du type char et short sont transformés en int ; • les paramètres du type float sont transformés en double.

TAB. 24: Conversion des paramètres

La structure des arguments de la fonction `main()` reflète la liaison entre le langage C et le système d'exploitation, et en particulier le système UNIX. Dans un système de type UNIX, les paramètres de la fonction `main()` sont passés par le shell dans la majorité des cas. Ces paramètres ont une structure prédéfinie. Ils sont décrits de la manière suivante :

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

Les noms `argc`, `argv` et `envp` sont des noms mnémoniques. Ils signifient argument count, argument values et environment pointer.
La fonction `main()` dispose donc toujours de trois paramètres passés par le shell. Ces paramètres sont un entier et deux tableaux de pointeurs sur des caractères. La signification de ces arguments est la suivante :

- L'entier contient le nombre d'arguments qui ont été passés lors de l'appel du programme (nombre de mots dans la ligne de commande).
- Le premier tableau contient les arguments de la ligne de commande au niveau du shell. Ces arguments sont découpés en mots par le shell et chaque mot est référencé par un pointeur dans le tableau. Il y a toujours au moins un argument qui correspond au nom du programme (`argv[0]`).
- Le nombre de pointeurs valides dans le premier tableau est donné par le contenu de la variable entière `argc`.
- Le deuxième tableau contient les variables d'environnement du shell au moment de l'appel du programme. Contrairement au premier tableau, on ne dispose pas pour ce deuxième tableau d'un nombre de mots valides. La fin de ce deuxième tableau est donnée par un marqueur. Ce marqueur est un pointeur NULL, c'est-à-dire, un pointeur qui contient l'adresse 0. Dans ce cas, cette adresse est du type `(char *)` ou bien encore l'adresse d'un caractère.

La figure 8.5 donne une vision interne des paramètres issus de la commande :
`echo essai de passage de parametres`

TAB. 25: Les paramètres de la fonction `main()`

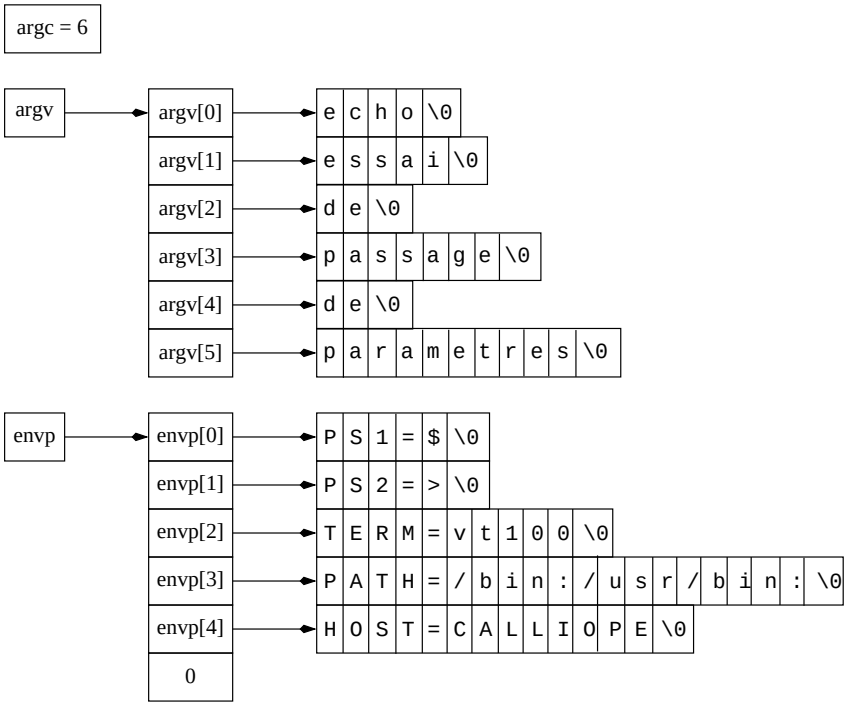


FIG. 7: Illustration des paramètres du `main()`

1.12 Pointeurs et tableaux

Un pointeur est une variable destinée à contenir une adresse mémoire. Il est reconnu syntaxiquement par l'`*` lors de sa déclaration. Comme les adresses, le pointeur est associé à un type d'objet. Ce type est celui des objets qui sont manipulés grâce au pointeur. L'objet peut être une variable ou une fonction. Voici deux exemples de déclaration de pointeurs : `int *ptint;` pointeur sur un entier, `char *ptchar;` pointeur sur un caractère.
La déclaration d'un pointeur n'implique pas la création de l'objet associé et l'affectation de l'adresse de l'objet au pointeur. Il faut donc déclarer un objet du type correspondant et initialiser le pointeur avec l'adresse de cet objet. Par convention, l'adresse 0 (NULL) est invalide et si l'on cherche à l'accéder, on obtient une erreur d'exécution du type bus-error sur UNIX. Les pointeurs déclarés en variables globales sont initialisés à 0. Les pointeurs déclarés en local ont des valeurs initiales dépendantes du contenu de la pile à cet instant.
Le compilateur C vérifie le type des adresses qui sont affectées à un pointeur. Le type du pointeur conditionne les opérations arithmétiques sur ce pointeur. Les opérations possibles sur un pointeur sont les suivantes : • affectation d'une adresse au pointeur ; par exemple, si `x` est une variable de type `int`, et si `ptr` est un `int*`, alors on peut affecter à `ptr` l'adresse de `x` de la manière suivante : `ptr = &x;` • utilisation du pointeur pour accéder à l'objet dont il contient l'adresse ; par exemple, `*ptr` accède à la valeur de la variable `x`. • addition d'un entier (n) à un pointeur ; la nouvelle adresse est celle du $n^{ème}$ objet à partir de l'adresse initiale ; par exemple, si `ptr1` est un `float*` et `ptr2` un `int*`, `ptr1 + 1` est l'adresse du flottant juste après `ptr1` en mémoire, et `ptr2 + 2` est l'adresse de l'entier juste après l'entier qui est après `ptr2`. • soustraction de deux pointeurs du même type ; cela calcule le nombre d'objets entre les adresses contenues dans les pointeurs (cette valeur est exprimée par un nombre entier).

TAB. 26: Les pointeurs

Instruction	Interprétation
`px = &x[0];`	`px` reçoit l'adresse du premier élément du tableau.
`y = *px;`	`y` reçoit la valeur de la variable pointée par `px`
`px++;`	`px` est incrémenté de la taille de l'objet pointé (ici 4 octets). Il contient donc `&x[1]`.
`px = px + i;`	`px` reçoit l'adresse du $i^{ème}$ objet à partir de l'objet courant.
`pt1 = &tab[0];`	`pt1` reçoit l'adresse du premier élément du tableau `tab`.
`pt2 = &tab[10];`	`pt2` reçoit l'adresse du onzième élément du tableau `tab`.
`i = pt2 - pt1;`	`i` reçoit la différence des deux pointeurs `pt1` et `pt2`, c'est-à-dire le nombre d'objets entre `pt2` et `pt1`. `i` contiendra 10 à la fin de cette instruction.

TAB. 27: Addition et soustraction avec des pointeurs.

La déclaration d'un tableau à une dimension réserve un espace de mémoire contigüe dans lequel les éléments du tableau peuvent être rangés. Comme le montre la figure 8, le nom du tableau seul est une constante dont la valeur est l'adresse du début du tableau. Les éléments sont accessibles par : le nom du tableau, un crochet ouvrant, l'indice de l'élément et un crochet fermant.
L'initialisation d'un tableau se fait en mettant une accolade ouvrante, la liste des valeurs servant à initialiser le tableau, et un accolade fermante. La figure 8 montre l'espace mémoire correspondant à la définition d'un tableau de dix entiers avec une initialisation selon la ligne : `int tab[10] = {9,8,7,6,5,4,3,2,1,0};`.
Il est aussi possible de ne pas spécifier la taille du tableau ou de ne pas initialiser tous les élements du tableau. Par exemple, `int tb1[] = {12,13,4,15,16,32000};` définit `tb1` comme un tableau de 6 entiers initialisés, et `int tb2[10] = {112,413,49,5,16,3200};` définit `tb2` comme un tableau de 10 entiers dont les 6 premiers sont initialisés. Depuis la normalisation du langage, l'initialisation des premiers éléments d'un tableau provoque l'initialisation de l'ensemble des éléments du tableau. Les derniers éléments sont initialisés avec la valeur 0 (ensemble des octets à zéro quelle que soit la taille des éléments).
Les noms de tableaux étant des constantes, il n'est pas possible de les affecter.

TAB. 28: Déclaration et initialisation

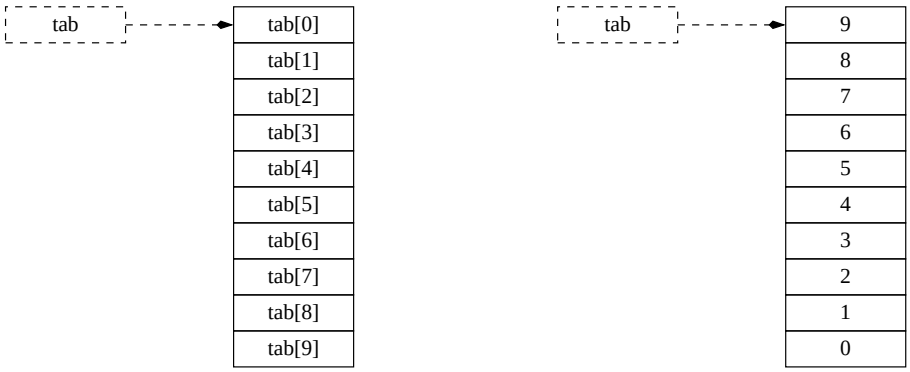


FIG. 8: représentation d'un tableau en mémoire.

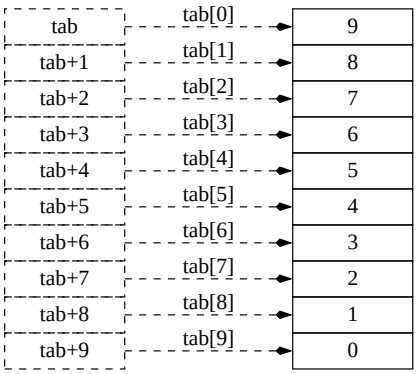


FIG. 9: équivalence entre adressage et indexation.

Il est possible d'additionner ou de soustraire un entier n à une adresse. Cette opération calcule une nouvelle adresse de la manière suivante :

- L'opération suppose que l'adresse de départ et l'adresse résultante sont les adresses de deux variables contenues dans le même tableau.
- L'opération suppose aussi que le tableau est d'une taille suffisamment grande, c'est-à-dire qu'elle suppose que le programmeur doit être conscient des risques de dépassement des bornes du tableau. Dans le cas du tableau tab ci-dessus, les adresses doivent être comprises entre `&tab[0]` et `&tab[9]`. Pour une question de test de borne, l'adresse immédiatement supérieure est calculable. Il est donc autorisé d'écrire `&tab[10]`, mais la zone mémoire correspondante ne doit pas être accédée.
- Selon ces conditions, l'addition d'un entier à une adresse retourne une adresse qui est celle du $n^{\text{ème}}$ objet contenu dans le tableau à partir de l'adresse initiale. Dans ces conditions, `tab + n` est l'adresse du $n^{\text{ème}}$ entier à partir du début du tableau. Dans notre exemple de tableau de dix éléments, n doit être compris entre 0 et 9. L'opérateur d'accès à la variable à partir de l'adresse (*) ne peut s'appliquer que pour n valant entre 0 et 9. Ainsi, `*(tab+n)` n'est valide que pour n compris entre 0 et 9.
- L'addition ou la soustraction d'un entier est possible pour toute adresse dans un tableau. Ainsi, `&tab[3] + 2` donne la même adresse que `&tab[5]`. De même, `&tab[3] - 2` donne la même adresse que `&tab[1]`.

Il est aussi possible de réaliser une soustraction entre les adresses de deux variables appartenant à un même tableau. Cette opération retourne une valeur du type `ptrdiff_t` qui correspond au nombre d'objets entre les deux adresses. Ainsi, `&tab[5] - &tab[3]` doit donner la valeur 2 exprimée dans le type `ptrdiff_t`. De même, `&tab[3] - &tab[5]` retourne la valeur -2 exprimée dans le type `ptrdiff_t`.

TAB. 29: Opérations possibles sur les adresses.

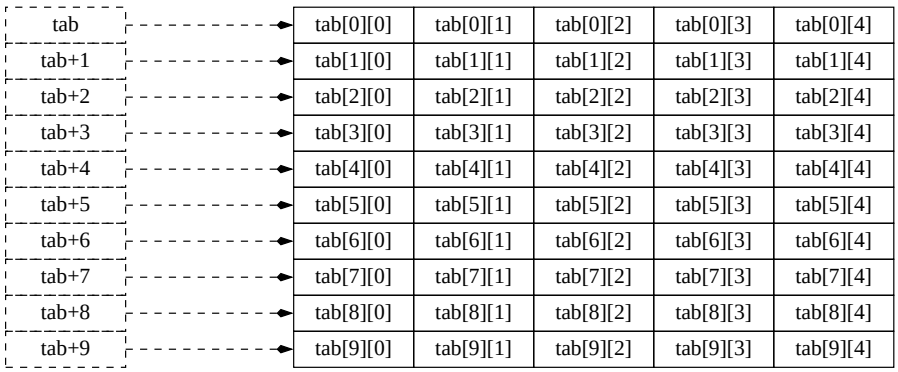


FIG. 10: Structure des tableaux à deux dimensions.

1.13 Structures, unions et énumérations

```
struct nom_de_structure {
   type1 nom_champ1;
   type2 nom_champ2;
   type3 nom_champ3;
   type4 nom_champ4;
   type5 nom_champ5;
   type6 nom_champ6;
   etc.
   typeN nom_champ_N;
};
```

```
struct nom_de_structure {
   type1 nom_champ1;
   type2 nom_champ2;
   type3 nom_champ3;
   type4 nom_champ4;
   type5 nom_champ5;
   type6 nom_champ6;
   etc.
   typeN nom_champ_N;
} variables;
```

```
struct {
   type1 nom_champ1;
   type2 nom_champ2;
   type3 nom_champ3;
   type4 nom_champ4;
   type5 nom_champ5;
   type6 nom_champ6;
   etc.
   typeN nom_champ_N;
} variables;
```

TAB. 30: Déclarations d'une structure.

L'accès aux éléments d'une structure, que nous appelons aussi champs, se fait selon la syntaxe : `nom_de_variable.nom_du_champ`
Lorsqu'on dispose d'un pointeur sur une structure, l'écriture diffère un peu en s'écrivant : `nom_de_variable->nom_du_champ`, où la flèche est construite avec le signe moins (−) et le signe supérieur (>).
par exemple :

```
struct date obdate, *ptdate;
ptdate = &obdate;
obdate.jour = 1; ptdate->jour = 1;
```

Pour le compilateur, l'accès à un champ d'une structure revient à faire un calcul de déplacement par rapport au début de la structure.

TAB. 31: Opérations sur les champs d'une structure.

La normalisation du langage C a autorisé une pratique qui était déjà courante dans de nombreux compilateurs, qui consiste à considérer une variable construite à partir d'un type structuré comme une variable simple. Ainsi, comme le montrent les définitions suivantes, il est possible de construire une fonction qui accepte en argument une variable structurée ou qui retourne une variable de type structuré.

```
struct date obdate, obdate2;
struct date newdate();
int chect date newdate();
int checkdate(struct date);
int result;
```

On autorise les opérations suivantes sur les variables de type structuré :

- l'affectation d'une variable structurée par une autre variable du même type : `obdate = obdate2`
- le test d'égalité ou d'inégalité entre deux variables structurées du même type : `obdate == obdate2` ou `obdate != obdate2`
- le retour d'une variable structurée par une fonction : `obdate = newdate()`
- le passage en argument d'une variable structurée à une fonction : `result = checkdate(obdate2)`

TAB. 32: Opérations sur une structure dans son ensemble.

Les structures donnent accès aux bits. Il est possible de découper logiquement un ensemble d'octets en des ensembles de bits. La précision de la longueur de chaque champ est faite par l'ajout de « : longueur» à chaque élément de la structure. On obtient ainsi une déclaration similaire à :

```
struct nom_de_structure {
  unsigned nom_champ1 : longueur1;
  unsigned nom_champ2 : longueur2;
  unsigned nom_champ3 : longueur3;
  unsigned nom_champ4 : longueur4;
                      : longueur5;
  unsigned nom_champ6 : longueur6;
  etc.
  unsigned nom_champ_N : longueurN;
} objets;
```

Il est recommandé de n'utiliser que des éléments de type unsigned. Un champ sans nom, avec simplement la taille, est un champ de remplissage pour cadrer sur des frontières de mot machine.

TAB. 33: Champs de bits.

La déclaration d'une union respecte une syntaxe proche de celle d'une structure :

```
union nom_de_union {
   type1 nom_champ1;
   type2 nom_champ2;
   type3 nom_champ3;
   type4 nom_champ4;
   type5 nom_champ5;
   type6 nom_champ6;
   etc.
   typeN nom_champ_N;
} variables;
```

Lorsque l'on définit une variable correspondant à un type union, le compilateur réserve l'espace mémoire nécessaire pour stocker le plus grand des champs appartenant à l'union.

TAB. 34: Déclaration d'une union.

La syntaxe d'accès aux champs d'une union est identique à celle pour accéder aux champs d'une structure. Une union ne contient cependant qu'une donnée à la fois et l'accès à un champ de l'union pour obtenir une valeur doit être fait dans le même type que celui qui a été utilisé pour stocker la valeur. Si on ne respecte pas cette uniformité dans l'accès, l'opération devient dépendante de la machine.

Les unions ne sont pas destinées à faire des conversions. Elles ont étés inventées pour utiliser un même espace mémoire avec des types de données différents dans des étapes différentes d'un même programme. Elles sont très utilisées dans les compilateurs. Les différents «champs» d'une union sont à la même adresse physique.

TAB. 35: Accès aux champs d'une union.

La définition d'une énumération respecte la syntaxe suivante :

```
enum nom_de_enumeration {
   enumerateur1,
   enumerateur2,
   enumerateur3,
   enumerateur4,
   enumerateur5,
   etc,
   enumerateurN
} variables;
```

Les valeurs associées aux différentes constantes symboliques sont, par défaut, définies de la manière suivante :

- la première constante est associée à la valeur 0 ;
- les constantes suivantes suivent une progression de 1. Il est possible de fixer une valeur à chaque énumérateur en faisant suivre l'énumérateur du signe égal et de la valeur entière exprimée par une constante. La progression reprend 1 pour l'énumérateur suivant. Par exemple, si on a la déclaration :
  `enum couleurs {rouge, vert, bleu} rvb;`
  alors `rouge` correspond à la valeur 0, `vert` à la valeur 1 et `bleu` à la valeur 2. Par contre, si on a la déclaration :
  `enum autrescouleurs {violet=4, orange, jaune=8};`
  alors `violet` est associé à la valeur 4, `orange` à la valeur 5 (violet + 1), et `jaune` à la valeur 8.

TAB. 36: Déclaration d'une énumération.

1.14 Entrée/Sorties

Vous trouverez ci-dessous les principales fonctions permettant de réaliser des opérations d'entrées/sorties soit clavier/écran, soit sur fichier. Rappelons que les entrées/sorties standards du C sont `stdin` (par défaut, le clavier), `stdout` (par défaut l'écran), et `stderr` (la sortie d'erreur, par défaut, l'écran).

`int getchar(void);`	
synopsis	cette fonction permet de lire un caractère sur `stdin` s'il y en a un. Ce caractère est considéré comme étant du type `unsigned char`.
argument	aucun.
retour	la fonction retourne un entier pour permettre la reconnaissance de la valeur fin de fichier (EOF). L'entier contient soit la valeur du caractère lu soit EOF.
conditions d'erreur	en fin de fichier la fonction retourne la valeur EOF.

`int putchar(int);`	
synopsis	cette fonction permet d'écrire un caractère sur `stdout`.
argument	Elle est définie comme recevant un entier pour être conforme à `getchar()`. Ceci permet d'écrire `putchar(getchar())`.
retour	Elle retourne la valeur du caractère écrit toujours considéré comme un entier.
conditions d'erreur	en cas d'erreur la fonction retourne EOF.

`char *gets(char *);`	
synopsis	lire une ligne sur `stdin` ; les caractères de la ligne sont rangés (un caractère par octet) dans la mémoire à partir de l'adresse donnée en argument à la fonction. Le retour chariot est lu mais n'est pas rangé en mémoire. Il est remplacé par un caractère nul `'\0'` de manière à ce que la ligne, une fois placée en mémoire, puisse être utilisée comme une chaîne de caractères.
argument	l'adresse d'une zone mémoire dans laquelle la fonction doit ranger les caractères lus.
retour	Si des caractères ont été lus, la fonction retourne l'adresse de son argument pour permettre une imbrication dans les appels de fonction.
conditions d'erreur	Si la fin de fichier est atteinte, lors de l'appel (aucun caractère n'est lu), la fonction retourne un pointeur de caractère de valeur 0 (NULL défini dans `<stdio.h>`).
remarque	Cette fonction ne peut pas vérifier que la taille de la ligne lue est inférieure à la taille de la zone mémoire dans laquelle on lui demande de placer les caractères. On lui préférera la fonction `fgets()` pour tout logiciel de qualité.

`int puts(char *);`	
synopsis	cette fonction permet d'écrire une chaîne de caractères, suivie d'un retour chariot sur `stdout`.
argument	l'adresse d'une chaîne de caractères.
retour	une valeur entière non négative en cas de succès.
conditions d'erreur	Elle retourne la valeur EOF en cas de problème.

`int scanf(const char *, ...);`	
synopsis	lecture formatée sur `stdin`.
arguments	comme l'indique la spécification «...», cette fonction accepte une liste d'arguments variable à partir du second argument. Le premier argument est une chaîne de caractères qui doit contenir la description des variables à saisir. Les autres arguments sont les adresses des variables (conformément à la description donnée dans le premier argument) qui sont affectées par la lecture.
retour	nombre de variables saisies.
conditions d'erreur	la valeur EOF est retournée en cas d'appel sur un fichier standard d'entrée fermé.

`int printf(const char *, ...);`	
synopsis	écriture formatée sur `stdout`.
arguments	chaîne de caractères contenant des descriptions d'arguments à écrire, suivie des valeurs des variables.
retour	nombre de caractères écrits.
conditions d'erreur	la valeur EOF est retournée en cas d'appel sur un fichier standard de sortie fermé.

`FILE *fopen(const char *, const char* );`	
synopsis	ouverture d'un fichier référencé par le premier argument (nom du fichier dans le système de fichiers sous forme d'une chaîne de caractères) selon le mode d'ouverture décrit par le second argument (chaîne de caractères).
arguments	la première chaîne de caractères contient le nom du fichier de manière à référencer le fichier dans l'arborescence. Ce nom est dépendant du système d'exploitation dans lequel le programme s'exécute. Le deuxième argument est lui aussi une chaîne de caractères. Il spécifie le type d'ouverture.
retour	un pointeur sur un objet de type FILE (type défini dans `<stdio.h>`) qui sera utilisé par les opérations de manipulation du fichier ouvert (lecture, écriture ou déplacement).
conditions d'erreur	le pointeur NULL (`(void *)0`) est retourné si le fichier n'a pas pu être ouvert (problèmes d'existence du fichier ou de droits d'accès).
mode d'accès	Le type d'ouverture est spécifié à partir d'un mode de base et de compléments. A priori, le fichier est considéré comme une fichier de type texte (c'est-à-dire qu'il ne contient que des caractères ASCII). Le type d'ouverture de base peut être : `"r"` le fichier est ouvert en lecture. Si le fichier n'existe pas, la fonction ne le crée pas. `"w"` le fichier est ouvert en écriture. Si le fichier n'existe pas, la fonction le crée. Si le fichier existe, la fonction le vide. `"a"` le fichier est ouvert en ajout. Si le fichier n'existe pas, la fonction le crée. Les écritures auront lieu à la fin du fichier. Le type d'ouverture peut être agrémenté de deux caractères qui sont : `"b"` le fichier est considéré en mode binaire. Il peut donc contenir des données qui sont transférées sans interprétation par les fonctions de la bibliothèque. `"+"` le fichier est ouvert dans le mode complémentaire du mode de base. Par exemple s'il est ouvert dans le mode `"r+"`, cela signifie qu'il est ouvert en mode lecture et plus, soit lecture et écriture. La combinaison des modes de base et des compléments donne les possibilités suivantes : `"r+"` le fichier est ouvert en lecture plus écriture. Si le fichier n'existe pas, la fonction ne le crée pas. Le fichier peut être lu, modifié et agrandi. `"w+"` le fichier est ouvert en écriture plus lecture. Si le fichier n'existe pas, la fonction le crée. Si le fichier existe, la fonction le vide. Le fichier peut être manipulé en écriture et relecture. `"a+"` le fichier est ouvert en ajout plus lecture. Si le fichier n'existe pas, la fonction le crée. Les écritures auront lieu à la fin du fichier. Le fichier peut être lu. `"rb"` le fichier est ouvert en lecture et en mode binaire. Si le fichier n'existe pas, la fonction ne le crée pas. `"wb"` le fichier est ouvert en écriture et en mode binaire. Si le fichier n'existe pas, la fonction le crée. Si le fichier existe, la fonction le vide. `"ab"` le fichier est ouvert en ajout et en mode binaire. Si le fichier n'existe pas, la fonction le crée. Les écritures auront lieu à la fin du fichier. `"r+b"` le fichier est ouvert en lecture plus écriture et en mode binaire. Si le fichier n'existe pas, la fonction ne le crée pas. Le fichier peut être lu, modifié et agrandi. `"rb+"` identique au précédent. `"w+b"` le fichier est ouvert en écriture plus lecture et en mode binaire. Si le fichier n'existe pas, la fonction le crée. Si le fichier existe, la fonction le vide. Le fichier peut être écrit puis lu et écrit. `"wb+"` identique au précédent. `"a+b"` le fichier est ouvert en ajout plus lecture et en mode binaire. Si le fichier n'existe pas, la fonction le crée. Les écritures auront lieu à la fin du fichier. `"ab+"` identique au précédent.

`int fclose(FILE *);`	
synopsis	c'est la fonction inverse de `fopen` ; elle détruit le lien entre le descripteur de fichier ouvert et le fichier physique. Si le fichier ouvert est en mode écriture, la fermeture provoque l'écriture physique des données du tampon.
arguments	un descripteur de fichier ouvert valide.
retour	0 dans le cas normal.
conditions d'erreur	la fonction retourne EOF en cas d'erreur.

`int fgetc(FILE *);`	
synopsis	lit un caractère (`unsigned char`) dans le fichier associé.
argument	descripteur de fichier ouvert.
retour	la valeur du caractère lu promue dans un entier.
conditions d'erreur	à la rencontre de la fin de fichier, la fonction retourne EOF et positionne les indicateurs associés.

`int getc(FILE *);`	
synopsis	cette fonction est identique à fgetc() mais peut être réalisée par une macro définie dans `<stdio.h>`.

`int ungetc(int, FILE *);`	
synopsis	cette fonction permet de remettre un caractère dans le buffer de lecture associé à un flux d'entrée.

`int fputc(int , FILE *);`	
synopsis	écrit dans le fichier associé décrit par le second argument un caractère spécifié dans le premier argument. Ce caractère est converti en un `unsigned char`.
argument	le premier argument contient le caractère à écrire et le second contient le descripteur de fichier ouvert.
retour	la valeur du caractère écrit promue dans un entier sauf en cas d'erreur.
conditions d'erreur	en cas d'erreur d'écriture, la fonction retourne EOF et positionne les indicateurs associés.

`int putc(int , FILE *);`	
synopsis	cette fonction est identique à `fputc()` mais elle est réalisée par une macro définie dans `<stdio.h>`.

`char *fgets(char *, int , FILE *);`	
synopsis	lit une ligne de caractères dans le fichier associé, les caractères de la ligne sont rangés dans la mémoire à partir de l'adresse donnée en argument à la fonction. Le retour chariot est lu et rangé en mémoire. Il est suivi par un caractère nul `'\0'` de manière à ce que la ligne une fois placée en mémoire, puisse être utilisée comme une chaîne de caractères.
arguments	le premier argument est l'adresse de la zone de stockage des caractères en mémoire. Le deuxième représente le nombre maximum de caractères (taille de la zone de stockage). Le troisième est le descripteur de fichier ouvert.
retour	adresse reçue en entrée sauf en cas d'erreur.
conditions d'erreur	à la rencontre de la fin de fichier, la fonction retourne NULL et positionne les indicateurs associés.

`int fputs(const char *, FILE *);`	
synopsis	cette fonction permet d'écrire une chaîne de caractères référencée par le premier argument dans le fichier décrit par le second argument.
argument	le premier argument contient l'adresse de la zone mémoire qui contient les caractères à écrire. Cette zone doit être une chaîne de caractères (terminée par un caractère nul). Elle doit contenir un retour chariot si on veut un passage à la ligne suivante. Le second argument contient le descripteur de fichier ouvert dans lequel les caractères seront écrits.
retour	une valeur positive si l'écriture s'est correctement déroulée.
conditions d'erreur	en cas d'erreur d'écriture, la fonction retourne EOF et positionne les indicateurs associés.

`size_t  fread(void *Zone,size_t Taille,size_t Nbr,FILE *fp);` `size_t fwrite(void *Zone,size_t Taille,size_t Nbr,FILE *fp);`	
synopsis	`fread` lit des enregistrements à partir d'un fichier, et `fwrite` écrit des enregistrements dans un fichier.
argument	le premier argument, que nous avons appelé Zone, est l'adresse de l'espace mémoire à partir duquel l'échange avec le fichier est fait. L'espace mémoire correspondant reçoit les enregistrements lus, ou fournit les données à écrire dans les enregistrements. Il faut, bien entendu, que l'espace mémoire correspondant à l'adresse soit de taille suffisante pour supporter le transfert des données, c'est-à-dire d'une taille au moins égale à (Taille * Nbr). Le deuxième argument, que nous avons appelé Taille, est la taille d'un enregistrement en nombre d'octets. Le troisième argument, que nous avons appelé Nbr, est le nombre d'enregistrements que l'on désire échanger. Le dernier argument, que nous avons appelé fp, est un descripteur de fichier ouvert dans un mode de transfert binaire.
retour	Ces deux fonctions retournent le nombre d'enregistrements échangés.

`int fprintf(FILE *,const char *,...);`	
synopsis	écriture formatée sur un fichier ouvert en mode texte.
arguments	le premier argument est un descripteur de fichier ouvert valide dans lequel les caractères sont rangés. Le deuxième est le format d'écriture des données. Enfin, le troisième correspond aux valeurs des données.
retour	nombre de caractères écrits.
conditions d'erreur	une valeur négative est retournée en cas d'erreur d'écriture.

`int fscanf(FILE *,const char *,...);`	
synopsis	lecture formatée dans un fichier ouvert en mode texte.
arguments	le premier argument est un descripteur de fichier ouvert dans lequel les caractères sont lus. Le deuxième est le format de lecture des données . Le troisième correspond à l'adresse des variables à affecter à partir des données.
retour	nombre de variables saisies.
conditions d'erreur	la valeur EOF est retournée en cas d'appel sur un fichier standard d'entrée fermé.

`int fseek(FILE *, long, int);`	
synopsis	change la position courante dans le fichier.
arguments	le premier argument est un descripteur de fichier ouvert. Le deuxième est le déplacement à l'intérieur du fichier en nombre d'octets. Le troisième correspond au point de départ du déplacement. Cet argument peut prendre les valeurs suivantes qui selon la norme doivent être définies dans le fichier `<stdio.h>` mais sont souvent dans le fichier `<unistd.h>` sur les machines de type "UNIX systeme V" : `SEEK_SET` : le déplacement est alors relatif au début du fichier ; `SEEK_CUR` : le déplacement est alors relatif à la position courante ; `SEEK_END` : le déplacement est alors relatif à la fin du fichier.
retour	0 en cas de succès.
conditions d'erreur	une valeur différente de zéro est retournée si le déplacement ne peut pas être réalisé.

`long ftell(FILE *);`	
synopsis	retourne la valeur de la position courante dans le fichier.
argument	descripteur de fichier ouvert.
retour	sur les fichiers binaires : nombre d'octets entre la position courante et le début du fichier. Sur les fichiers texte : une valeur permettant à fseek de repositionner le pointeur courant à l'endroit actuel.
conditions d'erreur	la valeur -1L est retournée, et la variable errno est modifiée.

`int fgetpos(FILE *, fpos_t *);`	
synopsis	acquiert la position courante dans le fichier.
arguments	le premier argument est le descripteur de fichier ouvert. Le deuxième contient la référence d'une zone permettant de conserver la position courante du fichier (le type `fpos_t` est souvent un type équivalent du type entier `long`).
retour	0 en cas de succès.
conditions d'erreur	une valeur différente de 0 est retournée, et la variable errno est modifiée.

`int fsetpos(FILE *, const fpos_t *);`	
synopsis	change la position courante dans le fichier.
arguments	le premier argument est un descripteur de fichier ouvert. Le deuxième contient la référence d'une zone ayant servi à conserver la position courante du fichier par un appel précédent à `fgetpos()`.
retour	0 en cas de succès.
conditions d'erreur	une valeur différente de 0 est retournée, et la variable errno est modifiée.

`void rewind(FILE *);`	
synopsis	si le descripteur de fichier ouvert `fp` est valide `rewind(fp)` est équivalent à `(void)fseek(fp,0L,0)`.

`void setbuf(FILE *,char *);`	
synopsis	associe un buffer à un fichier ouvert, dans le cas où le $2^{\text{ème}}$ pointeur est NULL, les entrée-sorties du fichier sont non bufferisées (chaque échange donne lieu à un appel système).
arguments	le premier argument est un descripteur de fichier ouvert. Le deuxième est l'adresse d'une zone mémoire destinée à devenir le buffer d'entrée-sortie associé au fichier ouvert, cette zone doit avoir une taille prédéfinie dont la valeur est BUFSIZE. Elle peut être égale au pointeur NULL, ce qui rend les entrée-sorties du fichier non bufferisées.

`int setvbuf(FILE * , char * , int , size_t );`	
synopsis	contrôle la gestion de la bufferisation d'un fichier ouvert avant son utilisation.
arguments	le premier argument est un descripteur de fichier ouvert. Le deuxième est l'adresse d'une zone mémoire destinée à devenir le buffer d'entrée-sortie associé au fichier ouvert, cette zone doit avoir la taille donnée en quatrième argument. Si l'adresse est égale à NULL, la fonction alloue de manière automatique un buffer de la taille correspondante. Le troisième argument représente le type de bufferisation, ce paramètre peut prendre les valeurs suivantes définies dans `<stdio.h>` : `_IOFBF` signifie que les entrée-sorties de ce fichier seront totalement bufferisées (par exemple les écritures n'auront lieu que lorsque le tampon sera plein). `_IOLBF` signifie que les entrée-sorties seront bufferisées ligne par ligne (i.e. dans le cas de l'écriture un retour chariot provoque l'appel système). `_IONBF` les entrée-sorties sur le fichier sont non bufferisées. Le quatrième argument contient la taille de la zone mémoire (buffer).

`int fflush(FILE *);`	
synopsis	vide le buffer associé au fichier.
argument	descripteur de fichier ouvert en mode écriture ou en mode mise-à-jour. Ce descripteur peut être égal à NULL auquel cas l'opération porte sur l'ensemble des fichiers ouverts en écriture ou en mise-à-jour.
retour	0 dans le cas normal et EOF en cas d'erreur.
conditions d'erreur	la fonction retourne EOF si l'écriture physique s'est mal passée.

`int ferror(FILE *);`	
synopsis	Cette fonction retourne une valeur différente de zéro si la variable qui sert à mémoriser les erreurs sur le fichier ouvert correspondant a été affectée lors d'une opération précédente.
argument	le descripteur de fichier ouvert pour lequel la recherche d'erreur est faite.
retour	une valeur différente de zéro si une erreur s'est produite.

`int feof(FILE *);`	
synopsis	Cette fonction teste si l'indicateur de fin de fichier a été affecté sur le fichier ouvert correspondant au descripteur passé en argument.
argument	le descripteur de fichier ouvert sur lequel le test de fin de fichier est désiré.
retour	retourne vrai si la fin de fichier est atteinte.

`void clearerr(FILE *);`	
synopsis	Cette fonction efface les indicateurs de fin de fichier et d'erreur du fichier ouvert correspondant au descripteur passé en argument.
argument	le descripteur de fichier ouvert pour lequel on désire effacer les valeurs de la variable mémorisant les erreurs et de la variable servant à mémoriser la rencontre de la fin de fichier.

`void perror(const char *);`	
synopsis	Cette fonction fait la correspondance entre la valeur contenue dans la variable errno et une chaîne de caractères qui explique de manière succincte l'erreur correspondante.
argument	Cette fonction accepte un argument du type chaîne de caractères qui permet de personnaliser le message.

1.15 Quelques règles de bonne conduite

Avec le langage C, il est très facile de programmer comme un sagouin et d'obtenir des programmes illisibles, inmaintenables et bugués jusqu'à la (substantifique) moelle. Aussi, afin d'éviter au mieux ces malheureux désagréments, vous demanderais-je de vous conformer aux règles de bon sens et d'esthétique ci-dessous. Celles-ci sont une compilation de la charte C de Christian Queinnec, de la description du style de programmation à employer pour contribuer à l'écriture du noyeau Linux ainsi que de mes goûts personnels. Bref, que des sources recommandables `:-)`.

1.15.1 pagination

Le source d'un programme est destiné à être lu : i) sur votre console d'ordinateur ; ii) sur un listing. Il faut donc qu'il puisse apparaître de manière lisible sur ces deux média. Aussi, vous n'écrirez pas de ligne plus longue que 75 caractères.

Pensez que, quand vous serez en entreprise, vos programmes pourront être lus par d'autres personnes, voire même vous les écrirez à plusieurs. Il convient dans ce cas d'avoir une indentation claire et précise. Il convient d'éviter les indentations fantaisistes telles que le tchoo-tchoo d'Eric Marshall :

```
                                                    extern int
                                                        errno
                                                          ;char
                                                             grrr
                              ;main(                           r,
  argv, argc )                  int    argc                       ,
   r        ;                 char *argv[];{int                    P( );
#define x  int i,             j,cc[4];printf("      choo choo\n"     ) ;
x  ;if    (P(  !              i              )        |  cc[  !      j ]
&  P(j    )>2  ?              j              :        i  ){*  argv[i++ +!-i]
;              for    (i=              0;;    i++                    );
_exit(argv[argc- 2    / cc[1*argc]|-1<<4 ]    ) ;printf("%d",P(""));}}
  P  (    a  )   char a   ;  {    a  ;   while(    a  >      "  B   "
  /* -    by E            ricM    arsh             all-      */);    }
```

C'est fun, mais quand vous aurez bossé pendant 24 heures d'affilée sur un programme pour pouvoir livrer à temps un logiciel, vous trouverez ce genre de programme beaucoup moins fun. Aussi, nous allons fixer les règles suivantes :

1/ les indentations se feront toutes par tranche de 2 caractères.

2/ une ligne qui contient une accolade ne contient pas d'autre texte. De plus, toute accolade ouvrante et son accolade fermante correspondante se trouvent sur les mêmes colonnes. Entre les deux, les instructions sont indentées de 2 caractères. Exemple :

```
{
  x = 3;
}
```

Ainsi, on peut voir rapidement quelle(s) instruction(s) correspondent à quel bloc. Ça peut vous paraître superflu, mais quand on écrit des dizaines de milliers de lignes de code, cela devient rapidement indispensable si on veut éviter les nervous breakdowns comme on dit de nos jours.

3/ si, après indentation, la fin d'une instruction dépasse les 75 caractères autorisés, vous écirez cette instruction sur plusieurs lignes et, à partir de la deuxième ligne, vous indenterez de 4 caractères. Par exemple :

```
printf("la somme de %d et de %d est egale a : %d\n",
    x, y, x+y);
```

4/ les instructions de contôle seront écrites sur une ligne, suivie d'une ligne d'instruction identée de 2 caractères, ou d'accolades indentées de 2 caractères contenant des instructions. Par exemple :

```
if (x == 3)           if (x == 3)           for (i = 0; i < 4; i++)
  y = 2;                {                     {
                          y = 2;                y += 2;
                        }                     }
```

5/ vous n'insérerez jamais d'espace avant les `';'`.

6/ vous n'écrirez jamais plus d'une instruction ou plus d'une déclaration par ligne. Par exemple, je ne veux pas voir :
```
int x,y; float z; char t;
x = 2; printf("%d", x); x++;
```
C'est en effet horrible quand on relit un programme d'avoir plusieurs instructions par ligne, surtout quand on recherche des bugs, car si on lit un peu trop rapidement, on risque de ne pas voir certaines instructions.

7/ toujours pour augmenter la lisibilité des programmes, et surtout faciliter les relectures rapides, vous insérerez une ligne blanche entre les déclarations de variables locales et la première instruction. Par exemple :
```
{
  int x;

  x = 2;
}
```

8/ quand votre programme fait plusieurs dizaines de milliers de lignes, il faut pouvoir retrouver rapidement sur un listing une fonction donnée. Il faut donc que les débuts des fonctions soient facilement reconnaissables. Pour cela, vous insérerez plusieurs lignes blanches entre la fin d'une fonction et le début de la suivante (3 lignes par exemple).

9/ Afin d'uniformiser les en-têtes de fonctions, lorsque vous déclarerez une fonction, vous écrirez sur une ligne son type de retour de ainsi que son nom et une parenthèse ouvrante, puis sur les lignes suivantes les paramètres (un par ligne, indenté de 2 caractères), le dernier caractère étant suivi sur la même ligne par la parenthèse fermante. Pour la fonction `main`, vous pourrez toutefois placer tous les paramètres sur la même ligne que `main` car c'est une fonction standard. Par exemple :
```
int ma_fonction(
  int   x,  /* coordonnnees d'un point dans l'espace */
  int   y,
  color c)  /* couleur du point */
{
```

1.15.2 Les fonctions

Comme indiqué précédemment, afin d'augmenter la lisibilité, vous séparerez vos fonctions de 3 lignes blanches. Cela dit, ça ne suffit pas pour pouvoir repérer rapidement une fonction sur un listing, vous allez donc rajouter avant la déclaration de la fonction une ligne de commentaires de 69 `'='`, suivie par des lignes de commentaires indiquant ce que fait la fonction, et ce qu'elle retourne (si elle retourne quelque chose évidemment). De plus, vous écrirez des WARNING indiquant les prérequis de votre fonction (elle ne renvoie le bon résultat que si certaines conditions sont vérifiées) et les dépendances éventuelles avec d'autres fonctions (par exemple, votre fonction utilise la manière dont une autre fonction a été écrite et, si l'on modifie celle-ci, la votre risque de ne plus marcher). Exemple :

```
/* ======================================================================== */
/* === la fonction suivante teste si elle peut rajouter un arc a un  === */
/* === graphe sans creer de cycle. Si l'ajout induit un cycle, elle  === */
/* === renvoie le statut CircuitDetecte. Sinon elle teste si elle    === */
/* === peut mettre a jour l'hypermatrice du noeud a l'origine de     === */
/* === l'arc. Le cas echeant, elle effectue le rajout et la mise a   === */
/* === jour et renvoie le statut AjoutOK, sinon elle renvoie un      === */
/* === statut ErreurHypermatrice.                                    === */
/* === WARNING: le test de la presence de cycle depend fortement de  === */
/* === l'implementation de la fonction DetecteCycle. Si celle-ci est === */
/* === modifiee, il faut reverifier la validite de la fonction       === */
/* === ci-dessous.                                                   === */
/* ======================================================================== */
```

```
int ajoute_arc(
    Graphe graphe,
    Arc    arc)
{
  ...
}
```

La fonction `main()` renvoie toujours un entier indiquant son statut d'exécution. Cela provient en fait de la gestion des processus unix qui peuvent ainsi indiquer au processus qui les a lancés si tout a bien fonctionné. Imaginez que vous lanciez un script qui lance un programme formattant un disque dur, puis un programme qui copie une archive d'un autre disque dur sur celui-ci, puis qui détruit l'archive sur l'autre disque dur. Si le premier programme a planté, vous risquez d'effacer votre archive alors même qu'elle n'a pas été sauvegardée sur votre disque dur. Si les programmes renvoient un code indiquant son statut d'exécution, i) vous n'effectuez les opérations suivantes que si tout a bien fonctionné; ii) si un problème est survenu, le code de retour vous indique de quel problème il s'agissait. Bref, la fonction `main` doit donc renvoyer un code, en fait un entier. Vous utiliserez les codes prévus à cet effet dans `<stdlib.h>`, répondant aux noms évocateurs de `EXIT_SUCCESS` et `EXIT_FAILURE`.

1.15.3 Écriture du code

Là encore, vous devez être particulièrement vigilants à obtenir un code TRÈS LISIBLE. Cela suggère que vous réfléchissiez un peu à ce que vous comptez écrire avant de programmer. Toute bonne programmation commence avec une feuille et un crayon. Lorsque vous pensez avoir une bonne vision de ce que vous devez écrire, alors seulement vous passez sur machine. Vous devez toujours obtenir des programmes simplement compréhensibles par un autre programmeur. Par exemple, évitez des programmes tels que :

```
#define _ -F<00||--F-00--;
int F=00,00=00;main(){F_00();printf("%1.3f\n",4.*-F/00/00);}F_00()
{
            _-_-_-_
       _-_-_-_-_-_-_-_-_
    _-_-_-_-_-_-_-_-_-_-_
  _-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
  _-_-_-_-_-_-_-_-_-_-_-_-_
    _-_-_-_-_-_-_-_-_-_-_
        _-_-_-_-_-_-_
            _-_-_-_
}
```

Ils sont très jolis, mais peu maintenables et totalement incompréhensibles. Voici quelques règles simples à appliquer pour ne pas «galérer» lorsque vous écrirez de grosses (et même de petites) applications :

1/ A l'exception des index de boucles, vous devez avoir des noms de variables significatifs et relativement petits. Des variables telles que `toto`, `xxx`, `X23`, sont à proscrire car on ne voit pas du tout ce qu'elles représentent. L'illustration la plus significative est lorsqu'on lit un programme écrit par une personne étrangère. Ci-dessous un programme hollandais, pas facile à lire tout de même :

```
typedef int Knoop;
typedef Knoop Knooplijst[N];
typedef int Matrix [N][N];

void KortstePAD(
  Matrix a,
  Knoop s,
  Knoop t,
  Knooplijst pad)
{
  enum Lab {perm, voorl};
  typedef struct {
    Knoop voorganger;
    int lengte;
    enum Lab labl;
    } Knooplabel;
  typedef Knooplabel ToestandGraaf[N];
  ToestandGraaf toestand;
  Knoop i, k, j;
  int  min;

  for (i = 1; i < N; i++)
    {
      toestand[i].voorganger = 0;
      toestand[i].lengte = ONEINDIG;
      toestand[i].labl = voorl;
    }
  toestand[t].lengte = 0;
  toestand[t].labl = perm;
  k = t;                /* aanvankelijk werkknoop */
  do                    /* is er een beter pad vanuit k? */
    {
      for (i = 1; i < N; i++)
        if ((a[k][i]!=0) && (toestand[i].labl == voorl))
          {
            if (toestand[k].lengte + a[k][i] < toestand[i].lengte)
              {
                toestand[i].voorganger = k;
                toestand[i].lengte = toestand[k].lengte + a[k][i];
              }
          }
      /* bepaal de voorlopig gelabelde knoop met de kleinste label */
      min = ONEINDIG;
      k = 0;
      for (i = 1; i < N; i++)
        if ((toestand[i].labl == voorl) && (toestand[i].lengte < min))
          {
            min = toestand[i].lengte;
            k = i;
          }
```

```
        toestand[k].labl = perm;
        printf ("%d %d %d\n", k, t, s);
        for (j = 1; j < N; j++)
          printf ("%6d\n", toestand[j].voorganger);
        for (j = 1; j < N; j++)
          printf ("%6d\n", toestand[j].lengte); printf ("\n");
        for (j = 1; j < N; j++)
          printf ("%6d\n", toestand[j].labl);
      } while(k != s);
}
```

2/ Vous n'emploierez jamais de variable nommée `'O'` ou `'l'` (o majuscule ou l minuscule) car on peut trop facilement les confondre avec les constantes entières 0 et 1.

3/ Vous commenterez vos programmes. Si vous éprouvez des problèmes à écrire un morceau de code, c'est que le code a besoin d'être commenté : vous ne pouvez pas espérer le relire plus tard et n'avoir aucune difficulté à comprendre ce que vous aviez écrit. C'est pourquoi les commentaires sont à écrire en même temps que le code, pas après quand vous devez me rendre un listing : à ce moment là, vous avez déjà largement oublié tout ce qui méritait d'être expliqué. Attention : il ne faut pas «trop» commenter les programmes, ils risquent de devenir illisibles. Il faut simplement décrire brièvement ce que vous faites et surtout expliquer toute subtilité.

4/ toutes les conditions simples à l'intérieur des instructions de test seront parenthésées. Par exemple, vous n'écrirez pas : `if (x == 3 && y == 2)` mais `if ((x == 3) && (y == 2))`. Ainsi, vous serez sûrs que vos tests sont corrects et vous ne passerez pas des journées entières à traquer des bugs dûs à une interprétation par C de vos conditions différente de la votre.

5/ Afin de limiter les risques de bugs, vos fonctions doivent être courtes. En principe, on considère qu'une fonction qui a plus de 50 lignes (2 écrans de console) a toutes les chances d'être buguée. Donc vous essayerez d'écrire des fonctions de moins de 50 lignes. Au besoin, vous pouvez scinder une grosse fonction en un ensemble de petites.

6/ Il peut arriver que, sous certaines conditions, une fonction puisse être amenée à effectuer une opération illégale (par exemple une affectation d'une variable dont l'adresse est NULL). Même si le reste de votre programme est conçu pour que ce cas n'arrive jamais, votre fonction doit tester si le cas arrive et, le cas échéant, afficher un message d'erreur puis terminer le programme. Par exemple :

```
void affecte_pointeur(
  int *ptr,
  int nombre)
{
  if (ptr == NULL)
    {
      fprintf(stderr, "affecte_pointeur : ne peut affecter un pointeur NULL\n");
      exit(EXIT_FAILURE);
    }
  *ptr = nombre;
}
```

Quel intérêt ? Tout d'abord si vous avez un bug dans votre programme qui amène à appeler `affecte_pointeur` avec un pointeur NULL, eh bien votre programme vous indiquera lors de son exécution ce qui n'a pas marché, et dans quelle fonction il a planté. Ensuite, si vous modifiez votre programme plusieurs mois après l'avoir écrit, vous risquez d'éliminer par inadvertance une instruction qui empêchait que `affecte_pointeur` ne reçoive un pointeur NULL. Dans ce cas, il est bien pratique de savoir où se situe l'erreur.

7/ Lorsque vous créez un `switch`, vous devez toujours avoir une ligne `default :`. Dans le cas où celle-ci n'est pas employée, vous indiquez en commentaire que l'on ne doit jamais passer sur cette ligne, vous affichez un message d'erreur et vous sortez du programme :

```
switch(variable)
{
  ...
  default:
    /* on ne doit jamais arriver sur cette ligne. Si c'est le cas, */
    /* cela signifie que l'on a rajoute des valeurs a 'variable' */
    fprintf("fonction bidule : valeur indefinie de 'variable'\n");
    exit(EXIT_FAILURE);
}
```

Ainsi, si une jour vous modifiez votre programme et rajoutez de nouvelles valeurs possibles pour `'variable'`, le `default` vous indiquera ce qui n'a pas marché et dans quelle fonction cela s'est produit.

8/ Si vous voulez limiter les bugs, il faut impérativement limiter le nombre de variables globales. Une variable n'est globale que si elle doit être partagée par un grand nombre de fonction et si l'on ne peut vraiment pas la passer en paramètre.

9/ Voyons maintenant une règle permettant une bonne évolutivité de vos programmes : on n'écrit jamais de constantes directement dans le code, mais on utilise des définitions en début de programme : `#define RAYON 1.32`. Ainsi, si vous voulez modifier la valeur de la constante, il n'y a qu'une ligne du programme à modifier.

10/ Dans la mesure du possible, les `#define` se trouveront dans des fichiers `.h`. Cependant, si on les place dans des fichiers `.c`, ils se trouveront OBLIGATOIREMENT en début de fichier.

11/ Autant que possible, vous factoriserez dans des boucles les répétitions de séquences d'instructions de tailles significatives.

12/ Quand vous écrivez un programme, après l'écriture de chaque fonction, vous testez celle-ci de manière exhaustive (envisager tous les cas possibles) et vous ne passez pas à la suivante tant que celle-ci ne marche pas correctement. Le non respect de cette règle entraîne des temps de débugage monstrueux : si votre programme plante sur une fonction donnée, est-ce que le bug se trouve dans cette fonction ou bien dans celles que vous aviez écrites avant ?

1.15.4 Compilation et préprocesseur

Vous vous arrangerez pour qu'il n'y ait bien évidemment aucune erreur de compilation dans vos programmes, mais aussi aucun warning. Au besoin, vous casterez explicitement vos variables, mais je ne veux voir aucun warning. Pourquoi? Simplement parce qu'un warning vous avertit que le compilateur n'interprète peut-être pas bien ce que vous avez écrit. En rajoutant ce qu'il faut pour les éliminer, i) vous vérifiez que vous n'avez pas fait d'erreur dans ce que vous avez écrit ; ii) vous faites ce qu'il faut pour que le compilateur comprenne correctement ce que vous voulez faire.

Dans vos fichiers `.h`, vous vous assurerez qu'il n'y a pas d'inclusion multiple : les fichiers `.h` sont insérés dans vos programmes grâce à la primitive `#include`. Rien n'empêche un programme C d'inclure un fichier `toto.h` qui, lui-même, incluera `titi.h`, qui lui aussi incluera `toto.h` et ainsi de suite. Pour éviter les cycles, dans un fichier `toto.h` vous écrirez la chose suivante :

```
#ifndef TOTO_H
#define TOTO_H

tout ce que vous voulez ecrire dans le fichier

#endif /* TOTO_H */
```

Maintenant, vous voilà prêt à programmer. Je sais bien que ces règles ont l'air un peu embêttantes à première vue, mais elles sont très simples à respecter et elles vous éviteront bien de déboires. En particulier, elles devraient vous permettre de limiter au maximum les phases de débugage.

2 Exercices

2.1 Syntaxe, types, initialisations et conversions

Exercice 1 Dans le programme suivant, quels sont les variables dont le nom est correct et celles dont le nom est incorrect ?

```
01 int xxx;
02 int yy-xx, zzzz**, **wwww;
03
04 void main(int argc, char **argv)
05 {
06   car    X;
07   double xxx, __standard__;
08   void   $abcde_01, 123abcd, aaa$bbb;
09   bool   XXX, xxx;
10   real   r@lm, rz08zr;
11   int    abcdefghijklmnopqrstuvwxyzabcdefghijklmnopq = 1;
12   int    abcdefghijklmnopqrstuvwxyzabcdefghijklmnopr = 2;
13 }
```

Exercice 2 D'après vous, quelles sont les valeurs des variables sur chaque ligne du programme suivant :

```
01 main()
02 {
03   int x = 2;
04   int y = 3, z = 5, t = 8;
05   int a = 5, b = 2;
06
07   x = t - 5 + z;
08   y = z / x;
09
10   a = a + b;
11   b = a - b;
12   a = a - b;
13 }
```

Exercice 3 Soient les déclarations :

```
long  A = 15;
char  B = 'A';        /* code ASCII : 65 */
short C = 10;
```

Quels sont le type et la valeur de chacune des expressions :

```
(1)   C + 3
(2)   B + 1
(3)   C + B
(4)   3 * C + 2 * B
(5)   2 * B + (A + 10) / C
(6)   2 * B + (A + 10.0) / C
```

Exercice 4 D'après vous, quelles sont les valeurs des variables à la fin du programme suivant :

```
01 #include <limits.h>
02
03 unsigned char x = '\010';
04
05 void main()
06 {
07   unsigned char x, u, v, w;
08   char a;
09   short int y, t;
10   unsigned short int z;
11
12   x = 255; x = x+1;
13   y = SHRT_MIN; y++;
14   z = SHRT_MAX; z++;
15   t = USHRT_MAX;
16   u = x + v;
17   v = '\010' + 3;
18   w = '\0' - 1;
19   a = '\0' - 1;
20 }
```

Exercice 5 D'après vous, quelles sont les valeurs des variables à chaque ligne du programme suivant :

```
01 void main()
02 {
03   int x = 1, y = 2;
04   float t, z;
05
06   z = (x * y) / (x / y);
07   t = ((float) x) - x;
08   x = 5; z = 2; y = x / z;
09   z = t + x / y;
10   t = x / y; t = (float) x / y; t = (float) (x / y);
11 }
```

Exercice 6 Quelles sont les valeurs des variables du programme suivant :

```
01 void main()
02 {
03   int   x = 10, y = 010, z = 0x10, u = 0x10 - 010;
04   short v = 32768U, w = 32768;
05   char  a = 256, b = x - y, c = 65536U, d = 65535UL;
06 }
```

Exercice 7 Quelles sont les valeurs des variables du programme suivant :

```
01 int x, y=10, z=20;
02
03 void main()
04 {
05   int y = y+2, t = x + y;
06   int u = 2 * y, v;
```

```
07
08   u = u + z; y = y + v; x = 3 + Z;
09   { int x = 20; y = 30; }
10   u = x + y;
11 }
```

Exercice 8 Quelles sont les chaînes contenues dans les variables du programme suivant :

```
01 void main()
02 {
03   char x[7] = "abcdef", y[8] = "abcdef\", z[10] = "abcd\0777", u[10] = "abcd\0aaa";
04   char a[5] = 'abcd', b[1] = 'a', c[10] = "a\0120\n\n";
05   char z[10] = "abcd\0777";
06 }
```

Exercice 9 Le but de cet exercice est d'écrire un petit convertisseur degrés celsius $\leftrightarrow$ degrés fahrenheit. La conversion se fait grâce à l'expression suivante :

$$\text{degré celsius} = \frac{5}{9}(\text{degré fahrenheit} - 32).$$

1/ Écrivez un code qui affecte à une variable `y` l'équivalent en celsius de la valeur d'une variable `x` exprimée en fahrenheit.
2/ Écrivez le code inverse, qui calcule `x` en fonction de `y`.

2.2 Éléments de base

Exercice 10 Écrivez un programme qui demande à l'utilisateur de rentrer un nombre entier et qui affiche la phrase «le nombre que vous avez rentré est : XXX en décimal, 0xYYY en hexadécimal ou encore 0oZZZ en octal», où XXX correspond bien évidemment au chiffre rentré par l'utilisateur et où YYY et ZZZ sont les équivalents de XXX en base 16 et 8.

Exercice 11 D'après vous, que fait le programme suivant (le genre de programme à ne pas me rendre en interro si vous comptez avoir la moyenne) :

```
01 #include <stdio.h>
02
03 main()
04 {
05   float x,y;
06
07   printf("entrez X:");
08   scanf("%d",&x);
09   printf("le nombre rentre est: %s\n", x);
10 }
```

Exercice 12 D'après vous, que fait le programme suivant :

```
01 #include <stdio.h>
02
03 main()
04 {
05   float x,y;
```

```
06
07   printf("entrez X et Y:");
08   scanf("%f %f",&x, &y);
09   printf("  %09.4f\n+ %09.4f\n----------\n  %9.4f\n", x,y,x+y);
10 }
```

Exercice 13 Trouvez les erreurs dans ce programme :

```
01 #inlcude <stdio.h>
02 #include [stdlib.h]
03
04 main()
05 {
06   float x; y; z;
07   int   a; b;
08
09   printf(entrez X Y et Z:);
10   {
11     int x;
12     scanf("%f", x);
13   }
14   scanf("%f" "%f",&y, &z)
15   printf("la somme de %d et de %f est %d\n", &x, y, x+y);
16   printf("le rapport est de %f/%f%\n"; x/y);
17   printf("la division entiere de %d par %d est : %f\n", a,b, (float) (a/b));
18 }
```

Exercice 14 Quel affichage produit le programme suivant :

```
01 #include <stdio.h>
02
03 main()
04 {
05   float x, y, z;
06   int   a, b;
07
08   printf("entrez x,y,z,a,b :");
09   scanf("%f %f %f %d %d", &x, &y, &z, &a, &b);
10   printf("%f %f %d %f %d", x+a, a+b, x+y, a/b, x/y);
11 }
```

2.3 Opérateurs et expressions

Exercice 15 Évaluez les expressions suivantes en supposant que :

$$A = 20 \qquad B = 5 \qquad C = -10 \qquad D = 2 \qquad X = 12 \qquad Y = 15$$

Notez chaque fois la valeur rendue comme résultat de l'expression et les valeurs des variables dont le contenu a changé.

```
(1)   (5*X)+2*((3*B)+4)
(2)   (5*(X+2)*3)*(B+4)
(3)   A == (B=5)
(4)   A += (X+5)
(5)   A != (C *= (-D))
(6)   A *= C+(X-D)
(7)   A %= D++
(8)   A %= ++D
(9)   (X++)*(A+C)
(10)  A = X*(B<C)+Y*!(B<C)
(11)  !(X-D+C)||D
(12)  A&&B||!O&&C&&!D
(13)  ((A&&B)||(!O&&C))&&!D
(14)  ((A&&B)||!O)&&(C&&(!D))
```

Exercice 16 Écrivez un programme qui prend un nombre en entrée entre 0 et 255 (8 bits) et qui affiche l'index du bit le plus significatif : 1 correspond au bit le plus à droite et 8 celui le plus à gauche. Par exemple, avec le nombre binaire 00101100, le programme affichera 6, avec 00101101 il faffichera toujours 6, avec 00000001 il affichera 1, et avec 00000000 il affichera 0.

Exercice 17

```
01 #include <stdio.h>
02 main()
03 {
04   int N=10, P=5, Q=10, R;
05   char C='S';
06
07   N = 5; P = 2;
08   Q = N++ > P || P++ != 3;
09   printf ("C : N=%d P=%d Q=%d\n", N, P, Q);
10
11   N = 5; P = 2;
12   Q = N++ < P || P++ != 3;
13   printf ("D : N=%d P=%d Q=%d\n", N, P, Q);
14
15   N = 5; P = 2;
16   Q = ++N == 3 && ++P == 3;
17   printf ("E : N=%d P=%d Q=%d\n", N, P, Q);
18
19   N=5; P=2;
20   Q = ++N == 6 && ++P == 3;
21   printf ("F : N=%d P=%d Q=%d\n", N, P, Q);
22
23   N=C;
24   printf ("G : %c %c\n", C, N);
```

```
25   printf ("H : %d %d\n", C, N);
26   printf ("I : %x %x\n", C, N);
27   return 0;
28 }
```

a) Sans utiliser l'ordinateur, trouvez et notez les résultats du programme ci-dessus.
b) Vérifiez vos résultats à l'aide de l'ordinateur.

Exercice 18 Sur internet, pour se connecter sur une machine, il suffit souvent de lui donner un nom, par exemple `www.google.com`. En fait, le protocole TCP/IP, celui qui vous sert dans vos connexions, ne comprend pas ces noms et il faut lui traduire en une adresse sur 4 octets (grâce à un serveur DNS par exemple). Par exmple, `www.google.com` correspond à 216.239.37.101. Lorsque vous configurez votre réseau, on vous demande souvent de rentrer un masque de sous-réseau. Celui-ci se présente aussi sous la forme d'un ensemble de 4 octets. Le masque sert à déterminer si une machine donnée appartient au même sous-réseau que la votre. S'il en est ainsi, elles pourront converser entre elles sans avoir à passer par une passerelle. «Comment ça marche?» me direz-vous. Eh bien on prend l'adresse de la machine externe, on en fait un ET logique avec le masque. On fait de même avec votre adresse. Si on retrouve les mêmes résultats, cela veut dire que la machine externe fait partie du même réseau. Par exemple, si votre machine est 216.239.37.101 (11011000.11101111.00100101.01100101 en binaire), si votre masque de sous-réseau est 255.255.165.0 (11111111.11111111.10100101.00000000) et si la machine `www.glurp.fr` a pour adresse 216.239.111.203 (11011000.11101111.01101111.11001011) alors :

```
      11011000.11101111.01101111.11001011          216.239.111.203
ET    11111111.11111111.10100101.00000000    ET    255.255.165.0
      11011000.11101111.00100101.00000000          216.239.37.0
```

De même,

```
      11011000.11101111.00100101.01100101          216.239.37.101
ET    11111111.11111111.10100101.00000000    ET    255.255.165.0
      11011000.11101111.00100101.00000000          216.239.37.0
```

Donc les deux machines sont sur le même sous-réseau et peuvent converser sans passerelle.

Écrivez un programme qui demande à l'utilisateur de rentrer l'adresse de sa machine, un masque de sous-réseau ainsi que l'adresse d'une machine distante et qui affiche à l'écran un message indiquant si les deux machines sont sur le même sous-réseau.

Exercice 19 Qu'affiche le programme suivant :

```
01 #include <stdio.h>
02
03 main()
04 {
05   int x=3, y=2;
06   float z=1.4, t=2.8;
07
08   printf("%f, %f, %f\n", z*x+x*y/x, x%2*5, 25>>4/2+(8!=3)*9<=4);
09   printf("%f, %f, %f\n", z&&x&y>4+2*5&3, 5>>20, x==y!=z==3);
10 }
```

Exercice 20 Écrivez un programme qui demande à l'utilisateur de rentrer un nombre entier X, et qui affiche la valeur de X^{13}. Réécrivez ce programme de manière à faire moins de 8 multiplications.

Exercice 21 Qu'affiche le programme suivant :

```
01 #include <stdio.h>
02
03 int x=1;
04
05 main()
06 {
07   int x=3, y=2;
08   float z=1.4, t=2.8;
09   {int x=2; {int y=3; printf("%d ",x/y);} printf("%d ",x/y);x++;}
10   printf("%d ",x/y); printf("%d ",x++/y); printf("%d ",++x/y);
11   {int y=3; {int y; printf("%d\n", y+2);}}
12   printf("%d\n",y);
13 }
```

Exercice 22 Écrivez un programme lisant deux flottants au clavier, disons x et y, et retournant leur moyenne géométrique, c'est-à-dire $\sqrt{xy}$.
Indication : utiliser le header «math.h» qui contient la fonction `sqrt`. Lisez-en le manuel en ligne.

Exercice 23 Déterminez si vous avez un pentium buggué jusqu'à la mœlle : Un tel test est possible en vérifiant si, en flottant, on a bien $x = (x/y)y$ avec $x = 4195835$ et $y = 3145727$. (J'ai «emprunté» cet exercice dans le recueil d'exercices en C (licence d'informatique de Paris 6) de Christian Queinnec).

2.4 Instructions de contrôle

Exercice 24 Écrivez un programme qui demande à l'utilisateur s'il veut rentrer au clavier un nombre ou une chaîne de caractères, qui lit ce que l'utilisateur a entré et, s'il avait choisi un nombre, renvoie un message indiquant si le nombre est pair ou non, et si c'était une phrase le message «le début de la phrase est :» suivi des 4 premiers caractères de la string.

Exercice 25 Écrivez un programme qui demande à l'utilisateur de rentrer une chaîne d'au plus 5 caractères et qui affiche son CRC sur 8 bits. Le CRC est obtenu en faisant la somme de tous les caractères de la chaîne.

Exercice 26 Écrivez un programme qui lit une phrase au clavier au format suivant : `x=`$nb1$ `y=`$nb2$ `z=`$nb3$ où $nb1$, $nb2$ et $nb3$ sont des nombres entiers rentrés par l'utilisateur. Par exemple `x=3 y=4 z=5` et `x=-5 y=33 z=0` sont des phrases valides, mais `3 5 6` n'en est pas une. Après avoir lu la phrase entrée par l'utilisateur, votre programme affichera la phrase :«la somme de $nb1$ et de $nb2$ est egale a $nb1 + nb2$», puis votre programme indiquera si $nb3$ est égal à $nb1 + nb2$.

Exercice 27 Écrivez un programme qui lit N nombres entiers au clavier et qui affiche leur somme, leur produit et leur moyenne. Choisissez un type approprié pour les valeurs à afficher. Le nombre N est à entrer au clavier. Résolvez ce problème,
a) en utilisant une boucle `while`,
b) en utilisant une boucle `do-while`,
c) en utilisant une boucle `for`.
d) Laquelle des trois variantes est la plus naturelle pour ce problème ?

Exercice 28 Écrivez un programme qui demande à l'utilisateur de rentrer deux nombres entiers m et n, et qui renvoie le PGCD de ces deux nombres. Pour effectuer ce calcul, remarquons que lorsque $m \geq n$, si q est le quotient de m par n et r le reste de la division de m par n, et si le PGCD que l'on cherche s'appelle S, alors il existe $k, k' > 0$ tels que $m = kS = qn + r$ et $n = k'S$. Bien entendu, si $r = 0$, $n = S$. Sinon $kS = qk'S + r$, et donc r est forcément un multiple de S. Donc $\mathrm{PGCD}(m, n) = \mathrm{PGCD}(n, r)$.

Exercice 29 Vous en avez assez de programmer en C et vous voulez monter un élevage de lapins dans le larzac comme un célèbre italien du douzième siècle. Bon, pourquoi pas. Au début (mois de janvier), vous ne possédez qu'un couple de bébés lapins, un mâle et une femelle. Chaque mois, chacun des couples enfante un nouveau couple de bébés. Ce dernier ne commence à se reproduire qu'au bout de deux mois. Écrivez un programme qui affiche, pour chaque mois des deux premières années, le nombre de lapins que vous possédez.

Exercice 30 Reprenons l'exercice précédent et essayons d'améliorer un peu l'algorithme. Clairement, si on appelle u_n le nombre de lapins au bout de n mois, alors on essaye de calculer $u_{n+1} = u_n + u_{n-1}$, avec $u_0 = 1$ et $u_1 = 1$, bref, c'est une suite de Fibonacci. Sans trop s'engager mathématiquement, on peut affirmer que $u_n = u_n$. Par conséquent, en utilisant une notation vectorielle :

$$\begin{pmatrix} u_{n+1} \\ u_n \end{pmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{pmatrix} u_n \\ u_{n-1} \end{pmatrix}.$$

Donc, par récurrence,

$$\begin{pmatrix} u_{n+1} \\ u_n \end{pmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n \begin{pmatrix} u_1 \\ u_0 \end{pmatrix}.$$

Écrivez donc un programme qui utilise cette formule pour calculer les nombres de Fibonacci.

Après avoir utilisé une bête boucle `for`, calculez la puissance n$^{\text{ème}}$ plus efficacement en utilisant la représentation binaire de n : il suffit de remarquer que $x^{n+p} = x^n \times x^p$, $x^{2n} = (x^n)^2$ et que la représentation binaire de $2n$ correspond à celle de n décalée d'un cran vers la gauche. Par conséquent, si l'on veut calculer x^{13}, puisque la représentation binaire de 13 est égale à 1101 ($= 8 + 4 + 0 + 1$ en décimal), il suffit de calculer $x^8 \times x^4 \times x^1$. Votre programme de calcul doit donc parcourir la représentation binaire de l'exposant n et calculer les x^i correspondant aux bits à 1 dans n. De plus, pour limiter les calculs, chaque fois que vous vous déplacez d'un cran vers la gauche dans n, si vous avez conservé la valeur de x^{2^i}, vous pouvez calculer $x^{2^{i+1}}$ en faisant le produit de x^{2^i} par x^{2^i}. Le calcul de x^{13} se résume donc à :

initialisation :	$y = 1$		$x2i = x$	
1er bit (1) :	$y = y \times x2i$	$(= x)$	$x2i = x2i \times x2i$	$(= x^2)$
2eme bit (0) :	$y = y$	$(= x)$	$x2i = x2i \times x2i$	$(= x^4)$
3eme bit (1) :	$y = y \times x2i$	$(= x^5)$	$x2i = x2i \times x2i$	$(= x^8)$
4eme bit (1) :	$y = y \times x2i$	$(= x^{13})$	$x2i = x2i \times x2i$	$(= x^{16})$

Exercice 31 L'exercice précédent améliorait les calculs de Fibonacci. On peut démontrer simplement en utilisant des séries génératrices (qui ne sont pas à votre programme, mais que je peux vous expliquer si vous insistez) que la suite converge vers :

$$u_n = \frac{\big((1+\sqrt{5})/2\big)^n - \big((1-\sqrt{5})/2\big)^n}{\sqrt{5}}.$$

Reprogrammez Fibonacci en utilisant cette formule. Vous ferez le calcul de la puissance n$^{\text{ème}}$ d'abord par dichotomie puis en utilisant la formule $x^n = e^{n \log x}$. Rappelons que l'exponentielle et le log nécessitent l'inclusion de «math.h».

Exercice 32 Écrivez un programme qui demande à l'utilisateur de rentrer une chaîne de caractères, et qui affiche le palindrome constitué par la chaîne suivie des mêmes caractères mais dans l'ordre inverse. Par exemple, si l'utilisateur rentre la chaîne "abcd", le programme écrira à l'écran la chaîne "abcddcba". On supposera que la chaîne entrée par l'utilisateur a une longueur inférieure à 100.
Indication : on se rappellera que les chaînes de caractères sont des tableaux de caractères terminés par le caractère '\0'.

Exercice 33 Le jeu des huit erreurs : bon, y en a 8, faut les trouver.

```
01 #include <stdio.h>
02
03 void main()
04 {
05   int x;
06
07   /* si x est egal a 5, on redemande une nouvelle valeur a l'utilisateur */
08   if (x = 5)
09     {
10       printf('entrez une nouvelle valeur pour x : ');
11       scanf ('%d', x);
12     }
13
14   /* pour i variant de 0 a 3, on ajoute x a lui-meme et on affiche le nouvel x */
15   for (i=0, i<=3, i++)
16     x *= 2;
17     printf("la nouvelle valeur de x est : ");
18     printf ("%d\n", x);
19 }
```

Exercice 34 Réalisez un programme en langage C permettant de rechercher dans une chaîne de caractères donnée, le nombre de fois qu'apparaît une séquence particulière. La suite de caractères à tester sera entrée la première dans un tableau de taille égale à 100. Le texte de l'utilisateur sera entré en second dans un tableau de caractères de taille 1000. On suppose que ni le texte, ni la séquence ne sont vides.

Exercice 35 Déterminez si une phrase entrée au clavier, d'au plus 100 caractères, terminée par un «retour-chariot», est un palindrome. On suppose que le texte n'est composé que de minuscules non-accentuées, que les mots ne sont séparés que par des espaces, et qu'il n'y a aucune ponctuation. Rappelons qu'un palindrome est un mot ou une phrase qui n'est pas modifié lorsqu'on inverse l'ordre des lettres qui le compose.

Exercice 36 Écrivez un programme qui, après avoir lu au clavier un texte contenant moins de 1000 caractères, inverse l'ordre des caractères dans ce tableau, puis affiche le texte inversé.

Exercice 37 Écrivez un programme C qui, à partir d'un tableau de n entiers trié par ordre croissant, renvoie un booléen indiquant si un nombre entré au clavier par l'utilisateur appartient au tableau ou non. La recherche du nombre dans le tableau se fera par dichotomie.

Exercice 38 Écrivez un programme qui lit une chaîne de caractères en utilisant la fonction `getchar()` (cf. la sous-section 1.14), et qui affiche le nombre d'espaces, de tabs et de newlines.

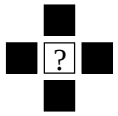
Exercice 39 Écrivez un programme qui lit une chaîne de caractères en utilisant la fonction `getchar()` et qui la recopie sur la sortie standard en remplaçant les chaînes de plusieurs espaces par un seul.

Exercice 40 Écrivez un programme qui lit une chaîne de caractères (`getchar()`) et qui l'affiche sur la sortie standard avec un seul mot par ligne.

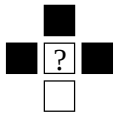
Exercice 41 Écrivez un programme qui inverse l'ordre des caractères dans une chaîne. En déduire un programme qui lit une chaîne de caractères (`getchar()`) et inverse l'ordre des caractères de chaque ligne.

Exercice 42 (le jeu de la vie)
Sur une matrice $n \times n$, une situation est composée de cases noires et de cases blanches. Les cases noires indiquent une situation de vie, et les cases blanches, une situation de mort.
Pour une case `m[i][j]`, on définit quatre voisins : `m[i-1][j]`, `m[i+1][j]`, `m[i][j-1]` et `m[i][j+1]`.

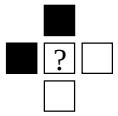
À chaque tour de jeu, on réexamine la situation des cases de la grille selon les règles suivantes :



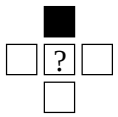
4 voisins noirs : “mort par étouffement”.
Si la case est blanche, elle reste blanche ;
Si la case est noire, elle devient blanche.



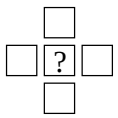
3 voisins noirs : aucun changement.



2 voisins noirs : la case devient noire.
Si la case est noire, elle reste noire ;
Si la case est blanche, elle devient noire.



1 voisin noir : aucun changement.



0 voisin noir : “mort par isolement”.
Si la case est blanche, elle reste blanche ;
Si la case est noire, elle devient blanche.

Écrivez un programme qui, après avoir lu une configuration initiale au clavier, demande à l'utilisateur le nombre d'étapes qu'il veut visualiser, et les affiche une à une à l'écran.

Le problème peut être simplifié en prenant une matrice $(n+2) \times (n+2)$ dont les contours sont toujours blancs. En effet, comme les cases blanches n'interviennent pas dans la décision de modification de l'état d'une case, cette bordure rajoutée artificiellement permettra de ne pas faire de traitement particulier pour les cases de bordure ou pour les cases d'angle de la matrice $n \times n$, qui sans cela ne posséderaient que 3 ou 2 voisins.

Exercice 43 Écrivez un programme qui lit une chaîne de caractères en utilisant la fonction `getchar()` et qui affiche un histogramme horizontal des longueurs des mots de la chaîne. Faites de même avec un histogramme vertical.

Exercice 44 Écrivez un programme qui range une suite de mots par ordre alphabétique. Les informations sont lues au clavier. L'arrêt de l'entrée d'un mot se traduit par la lecture d'un caractère \n. L'arrêt de l'entrée de la série de mots se traduit par la lecture d'un mot vide (c'est-à-dire composé uniquement du caractère \n). On suppose que chaque mot comprend au plus 19 caractères, et qu'il y a au plus 100 mots à classer.

Exercice 45 Écrivez un programme qui enlève tous les commentaires d'un programme C (on suppose que celui-ci est entré au clavier pendant l'exécution de votre programme). Ne pas oublier de tenir compte des chaînes de caractères : en effet, la chaîne `"/*"` n'est pas un commentaire.

Exercice 46 Écrivez un programme rudimentaire pour vérifier la validité de vos programmes C : vous testerez que les parenthèses, les crochets, les accolades, les commentaires, les quotes et les guillemets sont bien balancés.

Exercice 47 Écrivez une boucle équivalente à :

```
for (i=0; i<lim-1 && (c=getchar()) != '\n' && c!= EOF; ++i)
```

sans utiliser `&&` ni `||`.

Exercice 48 Écrivez un programme qui convertit une chaîne de caractères correspondant à un nombre en hexadécimal (incluant 0x ou 0X) en son équivalent entier. Par exemple, avec la chaîne `"0xFF"`, votre programme vous affichera 255.

Exercice 49 Écrivez un programme `lower` qui prend une chaîne de caractères en entrée, la convertit en minuscules et l'affiche à l'écran.

Exercice 50 Écrivez un programme qui choisit un nombre au hasard entre 0 et 100 (cf. les fonctions `rand()` et `srand(unsigned int seed)` déclarées dans `stdlib.h`) et qui vous demande de le retrouver. Si vous tapez un nombre plus élevé, il vous affiche «trop élevé»; si le nombre est plus petit, il vous affiche «pas assez grand, mon fils»; dans les deux cas, il vous redemande de taper un nombre. Si le nombre que vous avez rentré est celui qui avait été tiré au hasard, le programme vous dit «bravo, vous avez trouvé» et se termine.

Exercice 51 Vous allez réaliser sur machine un programme qui affiche un triangle de Sierpinsky. C'est un fractal qui est très simple à dessiner : Au départ, vous vous fixez un triangle (ABC). Les coordonnées de ces points n'ont aucune importance, mais si vous voulez obtenir une belle figure, il vaut mieux prendre un triangle équilatéral. Prenez le point M milieu du segment $[BC]$. L'algorithme consiste à itérer la séquence suivante un grand nombre de fois (10000 par exemple) :

1. Choisissez un point au hasard parmi A, B et C. Appelons-le D.
2. Calculez les coordonnées du point au milieu du segment $[MD]$. M prend les coordonnées de ce point.
3. affichez le pixel du point M à l'écran.
4. recommencez en 1.

Exercice 52 Reprenez l'exercice précédent de manière à dessiner un cône :

1. Choisissez un cercle (un centre X et un rayon R) ainsi qu'un point $A \neq X$ à l'intérieur du cercle. Choisissez un nombre entier n entre 6 et 20.
2. Parcourez votre cercle dans le sens trigonométrique et placez n points X_i à égale distance les uns des autres.
3. Dessinez un triangle de Sierpinsky pour chaque triplet (A, X_i, X_{i+1}).

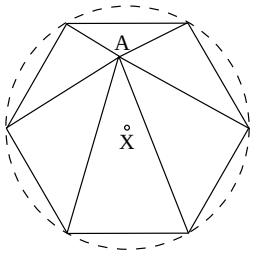


FIG. 11: Exemple de cône (à vous de le remplir)

Exercice 53 Écrivez un programme qui dessine un colimaçon sur l'écran.

Exercice 54 Écrivez un programme qui dessine l'ensemble de Mandelbrot à l'écran, ou encore mieux une partie de l'ensemble contenue dans un rectangle de l'espace. Bon, mais qu'est-ce que Mandelbrot ? C'est relativement simple : Considérons l'ensemble des nombres complexes : $Z = x + iy$. Ceux-ci peuvent être représentés par des pixels de coodonnées (x, y) sur l'écran. On peut définir la distance entre deux points $A = (x, y)$ et $B = (z, t)$ comme étant égale à $d(A, B) = (x - z)^2 + (y - t)^2$ — en fait, c'est le carré de la distance euclidienne, mais ce n'est pas bien grave pour ce qui nous concerne. Considérons maintenant la suite u_n de nombres complexes définie par $u_{n+1} = u_n^2 + u_0$. L'ensemble de Mandelbrot est tout simplement l'ensemble des points de départ u_0 tels que tous les u_n de la suite sont situés à une distance inférieure à un B donné du point de coordonnées $(0, 0)$ — $d(u_n, 0) \leq B$.

Le programme que vous allez écrire consistera donc à effectuer les opérations suivantes :

1. demander à l'utilisateur de rentrer les coordonnées du coin supérieur gauche et du coin inférieur droit d'une région rectangulaire de l'ensemble des complexes. Cette région sera représentée sur la totalité de la fenêtre de votre application.

2. demander à l'utilisateur de rentrer une valeur pour B (par défaut, choisissez la valeur 4 car on peut montrer que si, pour un i donné, $d(u_i, 0) > 4$ alors la suite contient des points aussi éloignés que l'on veut de l'origine).
3. demander à l'utilisateur de fixer un nombre d'itérations N maximum (en principe, si la région n'est pas trop petite, une valeur de 100 convient tout à fait) au dela duquel, si tous les u_i, $i < N$, ont été à une distance inférieure à B, alors on juge que tous les u_j suivants le seront aussi.
4. Pour chaque pixel de votre fenêtre, calculez les coordonnées du point u_0 correspondant dans l'espace des complexes.
5. calculez tous les u_i jusqu'à ce que soit $d(u_i, 0) > B$, soit $i = N$. Dans le premier cas, affichez le pixel avec une couleur dépendant du nombre d'itérations réalisés. Dans le deuxième cas, afficher un pixel noir.

Vous devriez obtenir des images similaires à celles ci-après :

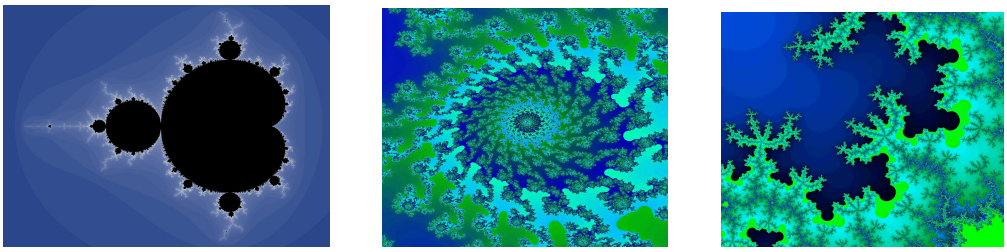


FIG. 12: Représentations de l'ensemble de Mandelbrot

Rappel sur les opérations sur les nombres complexes :

$$(x + iy) + (z + it) = (x + y) + i(y + t)$$
$$(x + iy) \times (z + it) = (xz - yt) + i(xt + yz)$$

Exercice 55 Reprenons l'exercice précédent, mais en généralisant un peu : Mandelbrot revient à calculer $u_{n+1} = g(u_n)$, où $g(x) = x^2 + u_0$. Pourquoi ne pas envisager d'autres fonctions que ce $g()$ là ? Pour un $g()$ arbitraire, l'ensemble ne s'appelle plus ensemble de Mandelbrot, mais ensemble de Julia. Modifiez donc le programme précédent en prenant les fonctions $g()$ suivantes :

1. $g(x) = x^2 + 0,3 - 0,4i$;
2. $g(x) = e^z$. Le calcul des exponentielles de nombres complexes est triviallement simple : $e^{x+iy} = e^x(\cos y + i \sin y)$.
3. $g(x) = \lambda x(1 - x)$.

Exercice 56 Vous aimez les fractals ? OK, on va changer de stratégie pour les générer. Jusqu'à maintenant, nous dessinions tous les points de l'écran, et c'est simplement l'utilisation de couleurs différentes qui nous permettaient de voir une jolie image. Maintenant, nous allons restreindre le nombre de points que nous allons dessiner et nous allons toujours utiliser la même couleur : nous allons utiliser des IFS (acronyme de «iterated function systems»). Le principe est le suivant : on part d'un point (x_0, y_0) que l'on affiche à l'écran. Nous allons calculer des points $(x_{n+1}, y_{n+1}) = f(x_n, y_n)$, où $f()$ est une fonction particulière. Chaque point (x_n, y_n) ainsi calculé sera affiché à l'écran. Avant de décrire les fonctions $f()$, commençons par décrire ce qu'est une fonction $f_1()$ linéaire à deux dimensions : c'est une fonction qui à tout couple (x, y) de $\mathbb{R}^2$ associe un autre couple (z, t), lui aussi de $\mathbb{R}^2$ tel que :

$$\begin{pmatrix} z \\ t \end{pmatrix} = f_1 \begin{pmatrix} x \\ y \end{pmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} e \\ f \end{pmatrix}.$$

Les fonctions $f()$ que nous allons utiliser sont linéaires mais aussi stochastiques, c'est-à-dire qu'elles commencent par choisir selon une certaine probabilité une fonction parmi un ensemble de fonctions linéaires, et ensuite elles appliquent cette fonction. Le choix doit être réalisé pour chaque point (x_n, y_n). Ainsi, si la fonction $f()$ peut choisir parmi les fonctions $f_1()$, $f_2()$, et $f_3()$ avec, respectivement, les probabilités 0,5, 0,25 et 0,25, on peut très bien obtenir : $(x_1, y_1) = f_2(x_0, y_0)$, $(x_2, y_2) = f_1(x_1, y_1)$, $(x_3, y_3) = f_1(x_2, y_2)$, $(x_4, y_4) = f_3(x_3, y_3)$, etc.

Réalisez donc un programme qui calcule, disons, 10000 points avec les fonctions suivantes :

- $f() = \begin{cases} f_1() = & a & b & c & d & e & f & \text{avec une probabilité de } 0.333 \\ f_2() = & 0.5 & 0 & 0 & 0.5 & 1 & 0 & \text{avec une probabilité de } 0.333 \\ f_3() = & 0.5 & 0 & 0 & 0.5 & 0.5 & 0.8660254 & \text{avec une probabilité de } 0.334 \end{cases}$

- $f() = \begin{cases} f_1() = & a & b & c & d & e & f & \text{avec une probabilité de } 0.01 \\ f_2() = & 0 & 0 & 0 & 0.16 & 0 & 0 & \text{avec une probabilité de } 0.85 \\ f_3() = & 0.85 & 0.04 & -0.04 & 0.85 & 0 & 1.6 & \text{avec une probabilité de } 0.07 \\ f_4() = & 0.2 & -0.26 & 0.23 & 0.22 & 0 & 1.6 & \text{avec une probabilité de } 0.07 \\ & -0.15 & 0.28 & 0.26 & 0.24 & 0 & 0.44 & \text{avec une probabilité de } 0.07 \end{cases}$

2.5 fonctions

Exercice 57 Écrivez une fonction qui prend en paramètres une chaîne de caractères ainsi qu'un tableau à deux dimensions de chaînes et un entier. La premier argument représente une expression mathématique (par exemple «$x + 27 * y/(18 + 24)$»), le deuxième argument permet de stocker la valeur des variables de l'expression (ici x et y), la première dimension du tableau correspondant aux noms des variables et la deuxième à leurs valeurs (par exemple on peut avoir `tab = {{"x","3.23"},{"y","68"}}`), enfin le dernier argument correspond au nombre d'éléments du tableau. Votre fonction remplacera la chaîne passée en premier argument par une chaîne contenant l'expression dans laquelle toutes les variables auront été remplacées par leurs valeurs. Ainsi la chaîne ci-dessus deviendrait «$3.23 + 27 * 68/(18 + 24)$».
Note : on supposera que la chaîne passée en argument est assez longue pour effectuer cette opération.

Exercice 58 Écrivez une fonction `strrindex(s,t)` qui retourne la position de l'occurence la plus à droite du caractère `t` dans la chaîne `s`. S'il n'y a pas d'occurence, votre fonction retournera `-1`. Rappelons que les chaînes de caractères sont des tableaux dont le premier élément se trouve à l'index 0.

Exercice 59 Écrivez une fonction `float mysqrt(float nb)` qui calcule la racine carrée X d'un nombre réel positif nb par approximations successives en utilisant la relation de récurrence suivante :

$$X_{n+1} = (X_n + nb/X_n)/2,$$

en partant de $X_0 = nb$. La précision du calcul, c'est-à-dire le nombre d'itérations, est à entrer par l'utilisateur. Vous vous assurerez lors de l'introduction des données par l'utilisateur que la valeur pour nb est un réel positif et que le nombre d'itérations est un entier naturel positif, plus petit que 50. Enfin, vous afficherez pendant le calcul toutes les approximations calculées :

```
La 1ere approximation de la racine carr\'ee de ... est ...
La 2e   approximation de la racine carr\'ee de ... est ...
La 3e   approximation de la racine carr\'ee de ... est ...
. . .
```

Exercice 60 Écrivez une fonction qui affiche un triangle isocèle formé d'étoiles de N lignes (N est fourni au clavier) :

```
Nombre de lignes : 8
       *
      ***
     *****
    *******
   *********
  ***********
 *************
***************
```

Exercice 61 Écrivez une fonction qui affiche la table de multiplication pour N variant de 1 à 10 :

```
 X*Y I   0   1   2   3   4   5   6   7   8   9  10
-----------------------------------------------------
  0  I   0   0   0   0   0   0   0   0   0   0   0
  1  I   0   1   2   3   4   5   6   7   8   9  10
  2  I   0   2   4   6   8  10  12  14  16  18  20
  3  I   0   3   6   9  12  15  18  21  24  27  30
  4  I   0   4   8  12  16  20  24  28  32  36  40
  5  I   0   5  10  15  20  25  30  35  40  45  50
  6  I   0   6  12  18  24  30  36  42  48  54  60
  7  I   0   7  14  21  28  35  42  49  56  63  70
  8  I   0   8  16  24  32  40  48  56  64  72  80
  9  I   0   9  18  27  36  45  54  63  72  81  90
 10  I   0  10  20  30  40  50  60  70  80  90 100
```

Exercice 62 Écrivez une fonction `atof` qui transforme une chaîne de caractères en un flottant (par exemple `12.45e-4`). Vous n'avez pas le droit d'utiliser les fonctions de conversion usuelles, ni les sscanf, etc.

Exercice 63 Problème de cet exercice : Rechercher dans un tableau d'entiers A une valeur VAL entrée au clavier. La valeur retournée par votre fonction sera soit la position de VAL dans le tableau, soit -1 si VAL n'appartient pas au tableau. Prototype de la fonction : `int recherche(int A[], int VAL)`. Vous implémenterez deux versions de votre fonction :

a) La recherche séquentielle, c'est-à-dire la comparaison successive des valeurs du tableau avec la valeur donnée.

b) La recherche dichotomique (ou encore «recherche binaire» ou «binary search»). Celle-ci suppose que le tableau A est trié par ordre croissant. Elle consiste à comparer le nombre recherché à la valeur au milieu du tableau.
   - s'il y a égalité ou si le tableau est épuisé, arrêter le traitement avec un message correspondant.
   - si la valeur recherchée est plus petite que la valeur actuelle du tableau, continuer la recherche dans le demi-tableau à gauche de la position actuelle.
   - si la valeur recherchée est plus grande que la valeur actuelle du tableau, continuer la recherche dans le demi-tableau à droite de la position actuelle.

Quel est l'avantage de la recherche dichotomique ? Expliquer brièvement.

Exercice 64 Écrivez une fonction qui détermine si une année est bissextile.
Rappel : Une année est bissextile si son expression numérale est divisible par 4, sauf si elle est divisible par 100, à moins qu'elle ne le soit par 400.
Exemples :
1996 est bissextile (multiple de 4, mais pas de 100) ;
2000 est bissextile (multiple de 4, de 100, mais aussi de 400) ;
2100 n'est pas bissextile (multiple de 4, de 100, mais pas de 400).

Exercice 65 Reprenez l'exercice 28 en utilisant une fonction récursive. Reprenez l'exercice 29 en utilisant une fonction récursive.

Exercice 66 On va voir dans cet exercice une nouvelle manière de générer le triangle de Sierpinsky : en utilisant un algorithme récrusif (c'est la dernière version, promis, juré). Vous partez donc d'un triangle (ABC). Tracez ce triangle. Appelons D, E et F respectivement les milieux des segments $[BC]$, $[AC]$ et $[AB]$. Tracez les segments $[DE]$, $[DF]$ et $[EF]$. Recommencez l'opération récursivement avec, à la place du triangle (ABC), les triangles (AEF), (BDF) et (CDE). (Tracez ces triangles sur une feuille de papier pour bien voir ce que doit faire l'algorithme).

Exercice 67 Écrivez une version récursive de `itoa`, qui transforme un entier en une chaîne de caractères.

Exercice 68 Écrivez une version récursive de la fonction qui inverse l'ordre des caractères dans une chaîne (cf. l'exercice 32).

Exercice 69 Écrivez un programme qui fait une conversion francs–euros et euros–francs : sur la ligne de commande, vous attendrez un nombre réel ainsi qu'une chaîne représentant l'unité du nombre. Par exemple, `prog 350.28 francs` vous affichera `53.4 euros`. Rappelons qu'un euro vaut 6,55957 francs. Vous porterez une attention particulière à la gestion des paramètres : vous indiquerez un message d'erreur si ceux-ci ne sont pas au bon format (`prog xxx yyy`).

Exercice 70 Écrivez une fonction récursive permettant de traduire un nombre romain en un nombre sous la forme décimale. La fonction ne verifiera pas la validité du nombre romain. Rappels :
M = 1000 D = 500 C = 100 L = 50 X = 10 V = 5 I = 1
Si un chiffre romain est suivi d'un chiffre romain de valeur supérieure, la valeur de ce chiffre se soustrait sinon la valeur de ce chiffre s'additionne.

Exercice 71 Modifiez le programme de l'exercice 28 de manière à ce que les deux nombres dont on cherche le PGCD soient entrés sur la ligne de commande.

Exercice 72 Écrivez un programme qui vous affiche une liste triée des arguments de votre ligne commande.

Exercice 73 Écrivez un programme qui effectue des conversions entre la base 10 et les bases 2 et 16.

Exercice 74 Écrivez un programme qui dessine un paysage fractal à l'écran. La procédure à suivre est récursive (et franchement simple à programmer), comme le montre la figure 13 : Vous partez d'un rectangle $ABCD$ horizontal sur l'axe X–Y (disons à l'altitude 100).

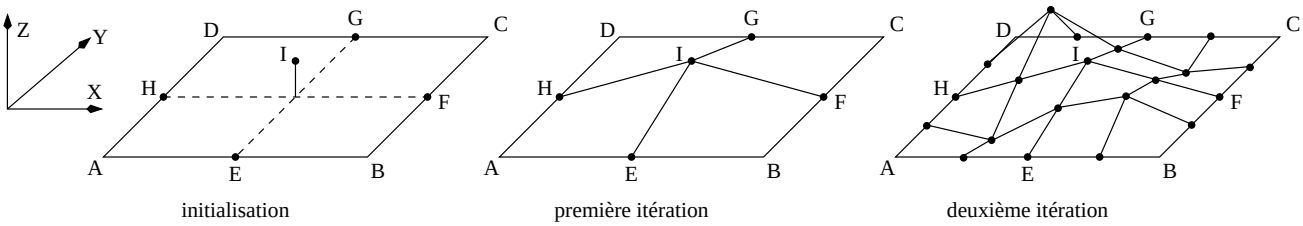


FIG. 13: Étapes de construction du paysage.

On commence par calculer les milieux de chacun des côtés, soient E, F G et H. Puis on calcule un point I selon un algorithme expliqué ci-après. Lorsque ceci est réalisé, on recommence avec les parallélépipèdes (dans l'ordre) $HIGD$, $IFCG$, $AEIH$, et $EBFI$, et ainsi de suite jusqu'à une certaine profondeur d'appel. Arrivé à cette profondeur (10 par exemple), vous dessinez le parallélépipède avec une couleur dépendant de l'altitude des points (au dessus d'une certaine hauteur, du blanc pour la neige, en dessous du marron pour faire des rochers, en dessous du vert pour les plaines, etc. L'ordre des quadruplets sert simplement à cacher les parties cachées du paysage.

Voyons maintenant la manière de calculer le point I : sur les axes X et Y, I est le barycentre de E, F, G et H, autrement dit,

$$x_I = \frac{x_E + x_F + x_G + x_H}{4} \qquad y_I = \frac{y_E + y_F + y_G + y_H}{4},$$

mais sur l'axe Z, c'est le barycentre + un bruit aléatoire. Le bruit en question sera un nombre aléatoire ϵ_n situé entre $[-100 \times 0,6^n; 100 \times 0,6^n]$, où n correspond à la profondeur de la récurrence actuelle. Autrement dit,

$$z_I = \frac{z_E + z_F + z_G + z_H}{4} + \epsilon_n,$$

où ϵ_n appartient à $[-60; 60]$ pour la première profondeur, à $[-36; 36]$ pour la deuxième, et ainsi de suite. L'idée de cette formule est de faire en sorte que plus les parallélépipèdes sont petits, plus les variations de terrain sont, elles-aussi, petites.

Bien entendu, pour afficher les parallélépipèdes à l'écran, vous aurez besoin de convertir des coordonnées en 3D en coordonnées 2D.

2.6 Pointeurs et tableaux

Exercice 75 Écrivez une fonction qui prenne en entrée deux vecteurs d'entiers U et V, ainsi que le nombre d'éléments que contiennent ces vecteurs, et qui renvoie leur produit scalaire. Rappelons que, si les vecteurs on n éléments, et si les index commencent à 1, $U \cdot V = \sum_{k=1}^{n} U_i \times V_i$.

Exercice 76 Rendez le programme suivant syntaxiquement correct tout en respectant les indications données en commentaire (sans modifier les lignes où figure le commentaire /* Ne pas modifier */ par exemple) :

```
01 #include "stdio.h"
02 int a=0; /* Ne pas modifier */
03
04 void ss_prog1(char c,*d); /* "void" obligatoire */
05 {
06   char *g; /* Ne pas modifier */
07   b = 3;
08   g = malloc(20);
09   scanf("%s",&g);
10   return(5);
11   *d = 'a'; /* Ne pas modifier */
12 }
13
14 int ss_prog2() /* Ne pas modifier */
15 {
16   char *f; /* Ne pas modifier */
17   f = (char *)malloc(10); /* Ne pas modifier */
18   f = "aaaaaaaaaa";
19   printf("%s",f); /* Ne pas modifier */
20   a++; /* Ne pas modifier */
21 }
22
23 main()
24 {
25   char b; /* Ne pas modifier */
26   int d;
27   int j = a+++1;
28   b = ssprog1(32,&b);
29   printf("%c",b); /* Ne pas modifier */
30   d = ss_prog2;
31   b = ss_prog3(6);
32   printf("%d",j); /* Ne pas modifier */
33   printf("%d",a); /* Ne pas modifier */
34   printf("%d",(int)b); /* Ne pas modifier */
35 }
36
37 char *ss_prog3(int h) /* Ne pas modifier */
38 {
39   int *e;
40   e = malloc(5*sizeof(int));
41   e = {'1','2','3','4','5'}
```

```
42   return(e); /* Ne pas modifier */
43 }
```

Indiquez, après modification du programme, les valeurs affichées à l'écran.

Exercice 77 Écrivez ce que contient chacune des variables après l'exécution du programme ci-dessous. Pour les pointeurs, vous ferez un petit dessin avec des flèches pour indiquer vers quelles adresses ils pointent.

```
01 int tab[5] = {1; 2; 3; 4; 5};
02 int x = 10, y = 30, z = 80;
03 int *a = &x, *b = &tab[3], *c;
04 int **d, **e;
05
06 d = &b; e = &a; c = &y;
07 *d++; tab += 2;
08 z = *a; **e = *b; **d = z;
09 y = tab[1];
10 a = b;
11 x += 50;
```

Exercice 78 Soit le programme suivant :

```
01 main()
02 {
03   int A = 1;
04   int B = 2;
05   int C = 3;
06   int *P1, *P2;
07   P1=&A;
08   P2=&C;
09   *P1=(*P2)++;
10   P1=P2;
11   P2=&B;
12   *P1-=*P2;
13   ++*P2;
14   *P1*=*P2;
15   A=+++*P2**P1;
16   P1=&A;
17   *P2=*P1/=*P2;
18   return 0;
19 }
```

Pour chaque ligne du programme ci-dessus, donnez les valeurs des variables `A`, `B`, `C`, `P1`, `P2`. Vous utiliserez des flèches pour représenter les pointeurs.

Exercice 79 Soit `P` un pointeur qui «pointe» sur un tableau A :

```
int A[] = {12, 23, 34, 45, 56, 67, 78, 89, 90};
int *P;
P = A;
```

Quelles valeurs ou adresses fournissent ces expressions :

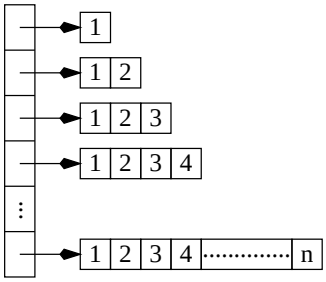
a) `*P+2`
b) `*(P+2)`
c) `&P+1`
d) `&A[4]-3`
e) `A+3`
f) `&A[7]-P`
g) `P+(*P-10)`
h) `*(P+*(P+8)-A[7])`

Exercice 80 Soit la séquence suivante :

```
01 int *tab[5],a,b,c,i;
02 a=0;
03 b=1;
04 c=2;
05 tab[0] = &a;
06 (*(*(&tab[0])))++;
07 tab[*(tab[0])] = &c;
08 *(tab+3) = tab[b];
09 tab[2]=tab[4]=&c;
10 (*(tab[2]))++;
11 for(i=0;i<5;i++) (*(tab[i]))++;
```

Donnez les valeurs de `a`, `b` et `c` à l'issue de cette séquence.
Remarque : On pourra remarquer avantageusement que, pour une variable `v` d'un type quelconque (y compris pointeur), on peut toujours utiliser la simplification suivante : `*&v == v`. En revanche, si l'on inverse les opérateurs `*` et `&`, la simplification équivalente n'a de sens que pour une variable `v` de type pointeur.

Exercice 81 Écrivez une fonction qui prend en argument un entier n et qui renvoie un élément de type int** correspondant à :



Exercice 82 Écrivez les fonctions strncpy, strncat et strncmp.

Exercice 83 Problème : On dispose de deux tableaux A et B (de dimensions respectives N et M), triés par ordre croissant. Écrivez une fonction qui retourne un tableau trié par ordre croissant contenant la fusion des éléments de A et B.
Méthode : Appelons FUS le tableau retourné par la fonction. Vous utiliserez trois indices IA, IB et IFUS. Vous comparerez A[IA] et B[IB] et vous affecterez à FUS[IFUS] le plus petit des deux éléments, puis vous avancerez dans le tableau FUS et dans le tableau qui a contribué son élément. Lorsque l'un des deux tableaux A ou B est épuisé, il suffit de recopier les éléments restants de l'autre tableau dans le tableau FUS.

Exercice 84 Écrivez un programme C permettant de saisir les notes (entières) obtenues par une classe de 40 élèves à 3 examens successifs. Les notes sont entrées dans l'ordre suivant :

[note de l'élève 1 à l'examen 1]
[note de l'élève 2 à l'examen 1]
[note de l'élève 3 à l'examen 1]
...
[note de l'élève 40 à l'examen 1]
[note de l'élève 1 à l'examen 2]
[note de l'élève 2 à l'examen 2]
...
[note de l'élève 40 à l'examen 2]
[note de l'élève 1 à l'examen 3]
[note de l'élève 2 à l'examen 3]
...
[note de l'élève 40 à l'examen 3]
A l'issue de la saisie des notes, le programme devra produire la moyenne obtenue par chaque élève à l'ensemble des examens.

Exercice 85 On veut créer une fonction générant, pour un entier $k > 0$, la suite de k lignes suivantes :
ligne 1 : 1
ligne 2 : 1 1
ligne 3 : 2 1
ligne 4 : 1 2 1 1
ligne 5 : 1 1 1 2 2 1
ligne 6 : 3 1 2 2 1 1
En fait, on obtient la ligne k en lisant ce qui est écrit sur la ligne $k-1$. Par exemple, la ligne 5 s'obtient en lisant la ligne 4 : sur cette dernière on voit 1 «1», 1 «2» et 2 «1».

1/ Écrivez une fonction `calc_prefixe` qui prend en argument une chaîne de caractères `str` ayant la forme des lignes mentionnées ci-dessus, et qui renvoie 0 si `str` est vide, ou le nombre de chiffres identiques au début de la chaîne `str`. Par exemple `calc_prefixe("1 1 1 2 2 1 ")` renvoie la valeur 3 car la chaîne de caractères commence par trois "1".

2/ Écrivez une fonction `genere_ligne` qui prend en argument une chaîne de caractères correspondant à l'une des lignes décrites ci-dessus, et qui renvoie la ligne suivante. Pour simplifier, on supposera qu'après chaque nombre, il y a un espace. On admettra en outre que tous les nombres affichés sont compris entre 1 et 9.

3/ Écrivez une fonction `genere` qui prend en argument un entier $k > 0$, et qui renvoie une chaîne de caractères contenant les k premières lignes de la suite.

4/ Écrivez une fonction `ecrit_suite` qui prend en argument un entier $k > 0$, et qui affiche la chaîne de caractères contenant les k premières lignes de la suite.

Exercice 86 Écrivez un programme qui construit le triangle de PASCAL de degré N, le mémorise dans une structure identique à celle de l'exercice 81, puis l'affiche à l'écran. Exemple : Triangle de Pascal de degré 6 :

```
n=0  1
n=1  1  1
n=2  1  2  1
n=3  1  3  3  1
n=4  1  4  6  4  1
n=5  1  5 10 10  5  1
n=6  1  6 15 20 15  5  1
```

Méthode de construction :
- Initialiser le premier élément et l'élément de la diagonale à 1.
- Calculer les valeurs entre les éléments initialisés de gauche à droite en utilisant la relation : $P_{i,j} = P_{i-1,j} + P_{i-1,j-1}$.

Exercice 87 (Spline cubique naturelle uniforme) Dans la plupart des éditeurs graphiques, il existe une fonction permettant de tracer une courbe à partir de la donnée d'un ensemble de $m+1$ points P_i de coordonnées

(x_i, y_i), $i = 0, \ldots, m$. L'idée des splines naturelles uniformes est de traiter séparément les abscisses et les ordonnées, et de créer ainsi une courbe pour les abscisses et une pour les ordonnées. L'agrégation de ces deux courbes nous fournit la courbe recherchée. Pour obtenir des calculs "simples", ces courbes sont elles-même découpées en ensembles de morceaux de courbes paramétrés $X_i(u)$ et $Y_i(u)$, $i = 0, \ldots, m-1$, telles que lorsque u varie de 0 à 1, $X_i(u)$ varie de x_i à x_{i+1}. Ainsi, dans l'exemple de la figure 14, la courbe recherchée se trouve en haut à gauche.

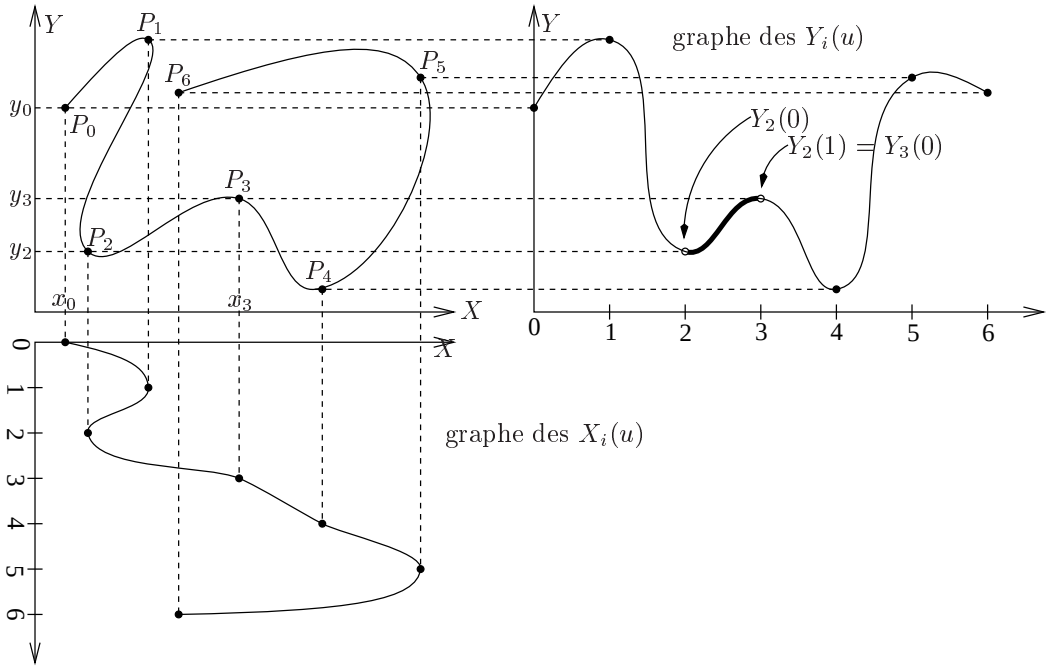


FIG. 14 – Exemple de Spline.

Elle est définie de la manière suivante :

courbe entre P_0 et P_1	$(X_0(u), Y_0(u)),\ u \in [0,1]$
courbe entre P_1 et P_2	$(X_1(u), Y_1(u)),\ u \in [0,1]$
courbe entre P_2 et P_3	$(X_2(u), Y_2(u)),\ u \in [0,1]$
courbe entre P_3 et P_4	$(X_3(u), Y_3(u)),\ u \in [0,1]$
courbe entre P_4 et P_5	$(X_4(u), Y_4(u)),\ u \in [0,1]$
courbe entre P_5 et P_6	$(X_5(u), Y_5(u)),\ u \in [0,1]$

Pour que les calculs ne soient pas trop coûteux (en temps), on se limite en principe à des courbes $X_i(u)$ et $Y_i(u)$ polynômiales de degré 3 en u, c'est-à-dire :

$$X_i(u) = a_i + b_i u + c_i u^2 + d_i u^3 \qquad Y_i(u) = e_i + f_i u + g_i u^2 + h_i u^3.$$

Le problème consiste à déterminer les bonnes valeurs de a_i, b_i, c_i, d_i, e_i, f_i, g_i, h_i, pour obtenir une courbe interpolant les points P_i.

1/ On veut que la courbe finale soit continue. On impose donc que les $X_i(1)$ (resp. $Y_i(1)$) se raccordent exactement aux $X_{i+1}(0)$ (resp. $Y_{i+1}(0)$). De plus on veut que la courbe finale ait un aspect lissé. Pour cela, on impose que la dérivée en u de $X_i(u)$ (resp. $Y_i(u)$) en $u = 1$ soit égale à celle de $X_{i+1}(u)$ (resp. $Y_{i+1}(u)$) en $u = 0$. Quelles équations obtient-on ?
Posons $D_i = \frac{dX_i(u)}{du}|_{u=0}$ pour tout $i \in \{0, \ldots, m-1\}$. Écrivez les valeurs de a_i, b_i, c_i et d_i en fonction des x_j et des D_j, $j = 0, \ldots, m$.

2/ Si l'on suppose de plus que la dérivée seconde de $X_i(\cdot)$ en 1 coincide avec celle de $X_{i+1}(\cdot)$ en 0, et que les dérivées secondes de $X_0(\cdot)$ en 0 et de $X_{m-1}(\cdot)$ en 1 sont égales à 0, alors on montre que les D_i sont solution du système suivant :

$$\begin{pmatrix} 2 & 1 & & & & & & \\ 1 & 4 & 1 & & & & & \\ & 1 & 4 & 1 & & & & \\ & & 1 & 4 & 1 & & & \\ & & & \cdot & \cdot & \cdot & \cdot & \\ & & & & & 1 & 4 & 1 \\ & & & & & & 1 & 2 \end{pmatrix} \begin{pmatrix} D_0 \\ D_1 \\ \cdot \\ \cdot \\ \cdot \\ D_{m-1} \\ D_m \end{pmatrix} = \begin{pmatrix} 3(x_1 - x_0) \\ 3(x_2 - x_0) \\ \cdot \\ \cdot \\ \cdot \\ 3(x_m - x_{m-2}) \\ 3(x_m - x_{m-1}) \end{pmatrix}.$$

En déduire un programme graphique (win32 application) qui a, dans un menu, 2 entrées : lorsque la première est sélectionnée, le programme enregistre en mémoire les coordonnées des points cliqués par l'utilisateur ; lorsque la deuxième est sélectionnée, le programme trace la spline passant par les points enregistrés.

Exercice 88 Déterminez dans quel sens le compilateur que vous utilisez fait croître la pile de C.

2.7 structures

Exercice 89 Soit le programme suivant :

```
01 #include "stdio.h"
02
03 struct s1 {
04   char c[4];
04   char *s;
05 };
06
07 struct s1 st1 = {"abc","def"};
08
09 struct s2 {
10   char *cp;
11   struct s1 ss1;
12 } st2 = {"ghi",{"jkl","mno"}};
13
14 int main()
15 {
16   printf("%c %c\n",st1.c[0],*st1.s);
17   printf("%s %s\n",st1.c, st1.s);
18   printf("%s %s\n",st2.cp,st2.ss1.s);
19   st2.cp=st2.cp+1;
20   st2.ss1.s=st2.ss1.s+1;
21   printf("%s %s\n",st2.cp,st2.ss1.s);
22 }
```

Donnez la représentation des données dans la mémoire centrale et les résultats affichés sur l'écran.

Exercice 90 Soit le programme suivant :

```
01 #include "stdio.h"
02
03 struct s1 {
04    char *s;
05    int i;
06    struct s1 *s1p;
07  };
08
09 struct s1 a[3] = {{"abcd",1,a+1},{"efgh",2,a+2},{"ijkl",3,a}};
10
11 int main()
12 {
13   struct s1 *p = a;
14   printf("%s %s %s\n", a[0].s, p->s, a[2].s1p->s);
15 }
```

Donnez les résultats affichés à l'écran ainsi que la représentation des données dans la mémoire centrale.

Exercice 91 Écrivez une petite bibliothèque de manipulation de nombres complexes. vous supposerez que ceux-ci sont représentés par la structure suivante :

```
typedef struct {
  float re;  /* partie reelle */
  float im;  /* partie imaginaire */
} complex;
```

Vous programmerez les fonctions suivantes :

1/ `complex *somme(complex nb1, complex nb2)`, qui renvoie le nombre complexe somme de `nb1` et de `nb2`. On rappelle que $(x+iy)+(z+it)=(x+z)+i(y+t)$.

2/ `complex *produit(complex nb1, complex nb2)`, qui renvoie le nombre complexe produit de `nb1` et de `nb2`. On rappelle que $(x+iy)+(z+it)=(xz-yt)+i(xt+yz)$.

3/ `complex *division(complex nb1, complex nb2)`, qui renvoie le nombre complexe résultat de la division de `nb1` par `nb2`. On rappelle que :

$$\frac{x+iy}{z+it}=\frac{(x+iy)(z-it)}{(z+it)(z-it)}=\frac{xz+yt}{z^2+t^2}+i\frac{yz-xt}{z^2+t^2}.$$

4/ `bool est_egal(complex nb1, complex nb2)`, qui renvoie un booléen indiquant si `nb1` est égal à `nb2`.

Exercice 92 On veut gérer soit-même ses allocations dynamiques parce qu'on pense qu'elles seront plus efficaces ainsi. Pour cela, vous allez écrire deux fonctions, `mymalloc` et `myfree`, qui vont remplacer les fonctions bien connues `malloc` et `free`. Voici comment vous allez vous y prendre :
Vous allez commencer par déclarer une variable globale `char tab_alloc[100000];`. Ce tableau contiendra l'espace mémoire dans lequel iront "piocher" `mymalloc` et `myfree`.
Au début du programme, on considérera que les 100000 éléments du tableau `tab_alloc` sont libres, c'est-à-dire qu'ils peuvent être alloués par `mymalloc`. Chaque fois que vous utiliserez `mymalloc`, celui-ci vous renverra un pointeur vers le début d'une zone du tableau non encore utilisée. Afin de différencier les zones du tableau utilisées de celles qui ne le sont pas, vous utiliserez une liste simplement chaînée de type :

```
struct zone_allouee {
   char *debut_zone;
   int taille_zone;
   struct zone_allouee *next;
};
```

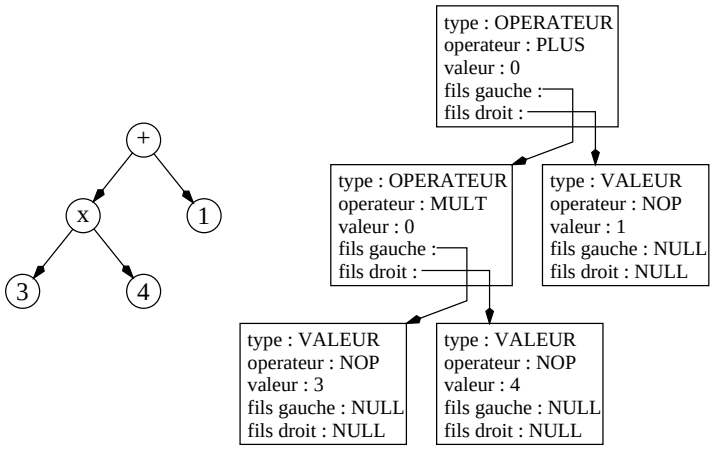
qui contiendra l'ensemble des zones allouées dans le tableau. Vous ferez en sorte que cette liste soit triée par ordre d'adresse de début de zone croissante. Pour allouer (resp. désallouer) les éléments de la liste chaînée, vous utiliserez le `malloc` (resp. `free`) classique.

Dans le cas où `mymalloc` ne peut plus faire d'allocation à l'intérieur de `tab_alloc`, vous renverrez le pointeur `NULL`. La fonction `myfree` aura pour but, tout comme `free`, de désallouer ce qui a été alloué par `mymalloc`, c'est-à-dire de rendre libre une zone occupée du tableau `tab_alloc`.

Tout comme `malloc` et `free`, les fonctions que vous écrirez auront pour type :

```
char *mymalloc (int);
void  free (char *);
```

Exercice 93 On désire écrire un petit module pour manipuler des expressions arithmétiques. Pour cela, on va utiliser des arbres d'évaluation tels que celui ci-dessous :



Afin de représenter de tels arbres, on définit le type `arbre` de la manière suivante :

```
typedef struct t_arbre {
  int   type;
  int   operateur;
  float valeur;
  struct t_arbre *fils_gauche;
  struct t_arbre *fils_droit;
} arbre;
```

où `type` est un entier représentant le type de noeud que possède l'arbre. Deux types sont possibles : `OPERATEUR` et `VALEUR`. Les noeuds de type `VALEUR` sont les feuilles de l'arbre et correspondent à des nombres. Les noeuds de type `OPERATEUR` sont ceux de l'arbre ayant des enfants. Les opérateurs arithmétiques possibles sont `PLUS`, `MOINS`, `MULT` et `DIV`. Pour les noeuds de type `VALEUR`, le champ `operateur` de l'enregistrement sera mis à `NOP`. Pour les noeuds de type `OPERATEUR`, le champ `valeur` sera mis à 0 (cf. la figure ci-dessus). On pourra utiliser les constantes suivantes :

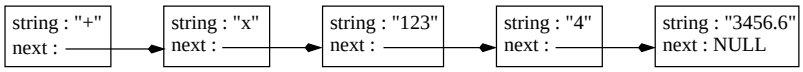
```
/* constantes de type */
#define OPERATEUR 0
#define VALEUR    1

/* constantes d'operateurs */
#define NOP   0
#define PLUS  1
#define MOINS 2
#define MULT  3
#define DIV   4
```

Question 1 Soit la structure de liste chaînée suivante :

```
typedef struct t_liste {
  char *string;
  struct t_liste *next;
} liste;
```

Écrivez une fonction `liste* decoupe_string(char *str)` qui prend en argument une chaîne de caractères `str`, qui sépare `str` en sous-chaînes en utilisant comme séparateur le caractère «espace», et qui renvoie la liste chaînée de ces sous-chaînes. Par exemple, `decoupe_string("+ x 123 4 3456.6 ")` renverra la liste :



Pour simplifier, on pourra supposer que la chaîne `str` se termine toujours par un espace.

Question 2 Afin de vérifier que votre fonction précédente est correcte, écrivez une fonction `void affiche_liste (liste *ll)`

Question 3 Écrivez une fonction `liste *enleve_premier_liste(liste **ll)` qui prend en argument un pointeur sur une liste chaînée, qui enlève de la liste le premier élément (s'il existe), et qui renvoie celui-ci s'il existe ou NULL sinon.

Question 4 Écrivez une fonction `arbre *cree_arbre(liste *ll)` qui prend en argument une liste chaînée créée par `liste* decoupe_string(char *str)`, et qui renvoie l'arbre correspondant. On supposera que la liste représente une expression sous forme préfixe (comme par exemple «+ x 123 4 3456.6 ») et qu'elle représente une expression valide. On supposera en outre qu'à part les nombres, on ne peut trouver que les opérateurs-chaînes de caractères «+», «-», «x» et «/».

Lorsque l'on rencontre un opérateur, l'idée est d'éliminer cet opérateur de la liste chaînée et d'allouer le noeud correspondant. On peut alors remplir tous les champs du noeud. Pour remplir le `fils_gauche`, il suffit d'appeler la fonction `cree_arbre` sur la liste `ll`. On peut alors remplir le `fils_droit` en réappelant la fonction `cree_arbre` sur la liste `ll`.

Question 5 Modifiez la fonction ci-dessus de manière à ce qu'elle puisse repérer les expressions incorrectement formées (par exemple : "+ x 123 4 ").

Question 6 Écrivez une fonction `float eval_arbre(arbre *arb)` qui prend en argument un arbre contenant une expression arithmétique bien formée et qui renvoie la valeur de cette expression.

Question 7 Écrivez une fonction `int main()` qui demande à l'utilisateur de rentrer une expression arithmétique sous forme préfixe et qui lui affiche la valeur de cette expression.

Exercice 94 On désire écrire un petit module pour manipuler des matrices. Pour cela, on définit le type `matrice` de la manière suivante :

```
typedef struct {
  float **tab;
  int     dim1; /* nombre de lignes de la matrice */
  int     dim2; /* nombre de colonnes de la matrice */
} matrice;
```

Question 1 Écrivez une fonction `matrice *cree_matrice(int dim1, int dim2, float val)` qui crée une matrice de dimension `dim1`×`dim2`, dont tous les éléments sont initialisés à `val`.

Question 2 Écrivez une fonction `void detruit_matrice(matrice *mat)` qui désalloue la matrice `mat`.

Question 3 Écrivez une fonction `void affecte_matrice(matrice *mat, int index1, int index2, float val)` qui affecte à l'élément de la matrice `mat` situé à l'indice `index1` sur la dimension 1 et à l'indice `index2` sur la dimension 2, la valeur `val` .

Question 4 Écrivez une fonction `void affiche_matrice(matrice *mat)` qui affiche le contenu de la matrice `mat`.

Question 5 Écrivez une fonction `matrice *copie_matrice(matrice *mat)` qui renvoie une copie de la matrice `mat`.

Question 6 Écrivez une fonction `matrice *produit_matrice(matrice *mat1, matrice *mat2)` qui crée une nouvelle matrice contenant le produit de `mat1` et de `mat2`.

Question 7 En utilisant la formule, $\forall n \geq 1$:

$$\begin{pmatrix} x_{n+1} \\ x_n \end{pmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n \begin{pmatrix} 1 \\ 1 \end{pmatrix}.$$

et en posant qu'un vecteur est une matrice dont la deuxième dimension n'a qu'un seul élément, écrivez une fonction qui calcule le $n^{\text{ème}}$ élément de la suite de Fibonacci. Dans un deuxième temps, faites le calcul de la puissance $n^{\text{ème}}$ de la matrice par dichotomie.

Exercice 95 La représentation d'un jeu de cartes :
Cet exercice est furieusement inspiré d'un exercice en CAML de Bruno Pagano.
Dans le présent exercice, vous allez progressivement programmer les premisses d'un arbitre pour jouer à la belote. Pour l'instant, votre but est de modéliser en langage C le jeu de 32 cartes servant à la belote.

Question 1 Une carte est définie par sa couleur (carreau, cœur, pique, trèfle) et sa hauteur (as, roi, dame, valet, dix, neuf, huit, sept). Définissez les types énumérés `couleur` et `hauteur` à cet effet.

Question 2 Définissez une structure `carte` comme un enregistrement contenant les informations de couleur et de hauteur. Puis définissez les valeurs représentant l'as de trèfle, la dame de cœur et le 8 de carreau.

Question 3 En utilisant des listes chaînées, définissez la structure `main` représentant l'*ensemble* (une liste) des cartes d'un joueur. En quoi le type `main` ne répond t-il pas entièrement à la question ? Donnez un contre-exemple.

Le comptage des points :
En fin de partie, les cartes sont évaluées de la façon suivante : quelle que soit la couleur de l'atout, l'as vaut 11 points, le roi vaut 4 points, la dame vaut 3 points, le dix vaut 10 points, le huit et le sept ne valent aucun point. Dans la couleur de l'atout, le valet vaut 20 points et le 9 vaut 14 points, mais dans les autres couleurs, le valet ne vaut que 2 points et le 9 ne vaut rien du tout.

Question 4 Écrivez une fonction `valeur` donnant la valeur d'une carte.

Question 5 Utilisez la fonction précédente pour définir une fonction `valeur_main` donnant la valeur d'une liste de cartes.

Le rangement des cartes :
Dans cet exercice, on souhaite ranger une `main` par couleur et par valeur (on rappelle que la valeur d'une carte dépend de sa couleur). L'ordre souhaité pour les couleurs est celui du bridge (Trèfle, Carreau, Cœur, Pique).

Question 6 Écrivez un prédicat `inf` qui retourne un booléen `true` si son premier argument est une carte devant être placée avant la carte donnée en second argument. (vous définirez bien entendu le type `booleen` ainsi que les valeurs `true` et `false`).

Question 7 Écrivez une fonction `insere` qui introduit une carte à l'endroit souhaité dans une main déjà convenablement rangée. Déduisez-en une fonction `range` qui classe une main.

Le tirage des cartes :
Par définition, l'ensemble des cartes d'un jeu est en bijection avec l'intervalle $[0, 31]$ des entiers.

Question 8 Définissez formellement une telle bijection (il y a bien entendu plus d'une réponse possible). Définissez alors au moyen de deux fonctions `cartes_of_int` et `int_of_cartes` cette bijection.

Question 9 Écrivez la fonction `tirage` qui génère une distribution du jeu de 32 cartes sous la forme d'une liste de cartes.

La représentation d'un jeu :
La belote est un jeu qui se joue à quatre joueurs usuellement désignés sous les noms de Nord, Sud, Est et Ouest. Les joueurs opposés géographiquement sont dans la même équipe. En début de partie, chaque joueur reçoit 8 cartes et une couleur est désignée comme celle de l'atout. Dans un tour, chacun joue une carte et l'équipe qui remporte le pli, le place dans son talon afin que les points soient comptés en fin de partie.

Question 10 En tenant compte des informations précédentes, définissez la structure de données `jeu` représentant l'état d'une partie à la fin d'un pli.

Question 11 Écrivez la fonction `initialise` créant une situation de début de partie.

Exercice 96 Dans cet exercice, vous allez progressivement programmer une version relativement simple (pas de graphisme ni de gestion du temps) du jeu de démineur (si ça vous amuse, vous pourrez facilement l'adapter graphiquement). Celui-ci se présente sous la forme d'un plateau rectangulaire de `nb_lignes` × `nb_colonnes` cellules. Au début du jeu, des mines (au nombre de `nb_mines`) sont déposées dans des cellules au hasard, mais le contenu des cellules est caché au joueur. À chaque tour de jeu, ce dernier sélectionne une nouvelle cellule et effectue l'une des opérations suivantes :

- il marque la cellule comme ayant, d'après lui, une mine. Il peut réaliser cette opération `nb_mines` fois ;
- il découvre le contenu de la cellule.

Le but est de découvrir l'ensemble des cellules ne contenant pas de mine. Si le joueur découvre une case minée, il saute avec et le jeu est perdu, sinon, on lui indique le nombre de mines autour de cette cellule, c'est-à-dire sur les 8 cellules adjacentes si la cellule n'est pas au bord du plateau, les 5 cellules adjacentes si elle est sur un bord mais pas dans un coin, et les 3 cellules adjacentes si elle est dans un coin. Pour réaliser le jeu, vous utiliserez les constantes et structures de données suivantes :

- `enum Statut_Cellule { Minee, Non_Minee };`
  C'est le statut d'une cellule : soit elle est minée, soit elle ne l'est pas.
- `enum Marquage_Cellule { Decouverte, Marquee_Minee, Non_decouverte };`
  C'est le marquage d'une cellule par le joueur : soit elle a été decouverte par le joueur, soit elle a été marquée comme contenant une mine, soit elle n'a pas encore été sélectionnée par le joueur.
- Le champ de mines est une matrice de `nb_lignes` × `nb_colonnes` cellules. Pour chaque cellule, on stocke les informations sur son statut, sur le nombre de mines sur les cases adjacentes, et sur son marquage :

```
struct champ_mine {
  int nb_lignes;
  int nb_colonnes;
  enum Statut_Cellule **statut;
  int  **mines_adjacentes;
  enum Marquage_Cellule **marquage;
};
```

- vous définirez deux variables globales `nb_marquages` et `nb_decouvertes`, qui contiendront respectivement le nombre de cases marquées comme minées par le joueur, et le nombre de cases découvertes par ce dernier.

Question 1 En se fondant sur les informations ci-dessus, écrivez une fonction `ajoute_mine` qui prend en argument un champ de mines et deux coordonnées `x` et `y`. Cette fonction rajoute une mine dans le champ sur la `y`$^{\text{ème}}$ ligne, `x`$^{\text{ème}}$ colonne du champ. Elle met bien évidemment à jour, pour chaque cellule du champ de mines, le nombre de cellules minées qui lui sont adjacentes. Bien entendu, si les coordonnées `x,y` sont en dehors

des coordonnées valides du champ de mines, la fonction n'effectuera aucune opération. Le type de retour de la fonction sera un void.

Question 2 En utilisant la fonction de la question précédente, écrivez une fonction `initialise_mines` qui prend en argument `nb_mines` ainsi qu'une matrice de type `champ_mine`, qui initialise cette matrice pour le début d'une partie, et qui renvoie finalement un void.

Question 3 Écrivez une fonction `marque`, qui prend en argument un champ de mines, les coordonnées `x,y` d'une cellule de ce champ ainsi que `nb_mines`. Cette fonction vérifie si la cellule en question n'est pas déjà marquée. Le cas échéant, elle «démarque» la cellule, c'est-à-dire qu'elle lui redonne le statut de `Non_decouverte`, sinon, lorsque le nombre de marquages est inférieur au nombre de mines, elle marque la cellule. Si les coordonnées `x,y` sont en dehors des coordonnées valides du champ de mines, la fonction n'effectuera aucune opération.
Attention : n'oubliez pas de mettre à jour le nombre de marquages (variable globale).

Question 4 Écrivez une fonction `découvre`, qui prend en argument un champ de mines et les coordonnées `x,y` d'une cellule de ce champ. Cette fonction découvre la cellule située aux coordonnées `x,y` du champ de mines si celle-ci ne l'est pas déjà. Si les coordonnées `x,y` sont en dehors des coordonnées valides du champ de mines, la fonction n'effectuera aucune opération.
Attention : ne pas oublier de mettre à jour le nombre de cellules découvertes.

Question 5 Modifiez la fonction de la question précédente de manière à ce que, lorsque la cellule aux coordonnées `x,y` a pour statut `Non_minee` et n'a aucune mine sur les cases adjacentes, la fonction découvre toutes ses cellules adjacentes, et recommence récursivement avec toutes celles qui ont aussi pour statut `Non_minee` et aucune mine sur les cases adjacentes.

Question 6 Écrivez une fonction `affiche_mines` qui prend en argument un champ de mines et qui affiche ce dernier. Les pourtours des cellules seront symbolisés de la manière suivante :

```
+-+
|x|
+-+
```

`x` est un caractère qui prendra les valeurs suivantes :
- `x = ' '` lorsque la cellule n'est pas découverte ;
- `x = 'M'` lorsque la cellule est marquée comme minée ;
- `x = 'n'`, où $n \in \{0, 1, 2, \ldots, 8\}$, lorsque la cellule est découverte. n est alors un chiffre égal au nombre de mines sur les cellules adjacentes.

Question 7 À chaque tour, le joueur rentre l'action qu'il désire effectuer en rentrant une chaîne de caractères de la forme `"x y a"`, où `x` et `y` sont des nombres correspondant aux coordonnées `x,y` d'une cellule du champ de mines, et où `a` est le caractère 'M' lorsque le joueur veut marquer la cellule, ou 'D' lorsqu'il veut découvrir la cellule. Écrivez une fonction `tour_de_jeu` qui prend en paramètres le champ de mines ainsi que le nombre de mines dans le champ. La fonction attend alors que le joueur entre une phrase de la forme `"x y a"` au clavier, elle exécute alors l'action correspondante et elle affiche le champ de mines. Elle renvoie un booléen indiquant si le joueur a perdu (il a découvert une case minée).

Question 8 Écrivez enfin une fonction `void démineur()` qui demande au joueur de rentrer la taille de la matrice et le nombre de mines du champ. Elle crée alors le champ de mines et itère les tours de jeu jusqu'à ce que le joueur ait perdu ou qu'il ait fini la partie.

Exercice 97 Lorsque des caractères rentrent en conflit avec les caractères structurant une URL, HTTP impose qu'ils soient encodés spécialement : %xy représente le caractère dont le code ASCII est xy. Ainsi %20 est l'espace. Écrivez une fonction convertissant en place une URL.

Exercice 98 Qu'imprime le programme suivant ?

```
/* $Id: strct.c,v 1.1 1997/09/24 18:26:18 queinnec Exp $
 * ftp://ftp.inria.fr/INRIA/Projects/icsla/WWW/Queinnec.html
 *
 * Passage de structures par copies.
 */

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

struct S {
  char   name[8];
  int    age;
  struct S *mother;
};

void
print (struct S *cell)
{
  printf("{%8s, %d, ", cell->name, cell->age);
  if ( cell->mother ) {
    print(cell->mother);
  }
  printf("}");
}

struct S
pop (struct S cell, char *name, int age)
{
  strncpy(cell.name, name, 8);
  cell.age = age;
  return *(cell.mother);
}

struct S
push (struct S cell, struct S *mother)
{
  strncpy(cell.name, "nothing", 8);
  cell.age  = 0;
  cell.mother = mother;
  return cell;
}

void
main ()
{
  struct S one = { "one", 1, NULL };
  struct S two = { "two", 2, &one };
  struct S three = { "three", 3, &two };
  struct S x;

  /* three n'est pas modifie et x est two. */
```

```
  print(&three);  printf("\n");
  x = pop(three, "cuatro", 4);
  print(&three);  printf("\n");
  print(&x);  printf("\n");

  /* La, x est modifie. */
  x = push(three, &two);
  print(&three);  printf("\n");
  print(&x);  printf("\n");

  exit(EXIT_SUCCESS);
}

/* end of strct.c */
```

2.8 Entrées/Sorties

Exercice 99 Écrivez le programme cat d'unix, qui affiche sur la sortie standard le contenu d'un fichier (le nom du fichier est passé en paramètre du programme).

Exercice 100 Écrivez un programme `tail` qui affiche les n dernières lignes d'un fichier dont le nom est passé en paramètre du programme. Par défaut, $n = 10$. Par exemple, `tail toto.txt` affiche l'ensemble du fichier `toto` si celui-ci a moins de 10 lignes, et les 10 dernières lignes de `toto` sinon, et `tail 5 toto.txt` réalise la même opération, mais avec les 5 dernières lignes seulement. Vous n'imposerez pas une limite supérieure au nombre de lignes passé en paramètre.

Exercice 101 Réalisez un programme agissant comme un filtre Unix, retirant les commentaires d'un texte correspondant à un programme en langage C++. On retiendra deux types de commentaires, les commentaires enserrés entre /* et */ et ceux débutant par // et s'achevant en fin de ligne.
Les caractères seront lus avec fgetc puis émis avec fputc.

Exercice 102 Écrivez un cross-referencer qui affiche la liste de tous les mots contenus dans un document et, pour chaque mot, la liste des lignes sur lesquelles il est écrit. Le nom du document sera passé en paramètre sur la ligne de commande.

Exercice 103 Écrivez un programme qui affiche la liste de toutes les variables employées dans un programme C spécifié comme paramètre du programme sur la ligne de commande.

Exercice 104 Écrivez un programme qui compare deux fichiers, et qui affiche la première ligne où ils different.

Exercice 105 Écrivez un programme qui prend en entrée un fichier et qui le formatte pour l'envoyer a l'imprimante : sur chaque page, un en-tête spécifie le nom du fichier ainsi que le numero de page, suivi d'une ligne composé de 80 `'-'`, elle-même suivie de 54 lignes de texte.

Exercice 106 Écrivez un programme `MonMcWrite` qui prend en argument le nom d'un document texte et qui compresse ce fichier au format MacWrite (que nous étudierons en détail dans le cours de compression) avec le même nom de fichier, mais avec l'extension `.mcw`.

L'algorithme utilisé par le traitement de texte MacWrite pour compresser son texte consiste à coder les caractères les plus souvent employés, à savoir «␣acdefhilnoprst», sur 4 bits. Les autres caractères sont codés grâce à leur code ASCII, donc sur 8 bits, précédés par un caractère d'échappement de 4 bits qui indique que ce qui suit est un code ASCII. Ils sont donc codés sur 12 bits. De plus, l'algorithme de compression traite chaque paragraphe séparément. Il essaye le codage décrit ci-dessus. Si ce codage a une taille inférieur à un codage tout en ASCII, il le garde, sinon l'algorithme écrit tout le paragraphe en ASCII. Afin de différencier un paragraphe

écrit en ASCII d'un paragraphe compressé, MacWrite utilise en début de paragraphe un bit de positionnement (flag).

Le fichier en sortie de votre programme aura pour en-tête le texte suivant : «MonMcWrite 1.0», suivi par le texte au format spécifié ci-dessus. Lorsque le fichier d'entrée a déjà l'extension `.mcw`, si le début du fichier contient «MonMcWrite 1.0», vous écrirez sur la sortie standard d'erreur le message `«Ce fichier a deja ete compresse avec MonMcWrite»` puis vous terminerez le programme, sinon vous écrirez sur la sortie standard d'erreur le message `«Impossible de creer le fichier : il existe deja»` et vous terminerez le programme.
Note : vous pourrez utiliser avec profit les fichier `prog_bitio.h` et `prog_bitio.c` qui réalisent des opérations de lecture/écriture fichiers de bits plutôt que d'octets.

Exercice 107 Le but de cet exercice est d'implémenter une version basique de RSA, l'un des systèmes de cryptographie le plus utilisé dans le monde. Pas de panique, ce n'est pas extrêmement compliqué. RSA (l'acronyme de Rivest, Shamir, Adlemann, ses concepteurs) est un système de clé publique/clé privée. L'idée en est la suivante : votre programme calcule deux clés, l'une, publique, qui est lisible par tout le monde (située par exemple dans un répertoire accessible en lecture par n'importe qui), l'autre, privée, n'est connue que de vous et n'est accessible que par vous. Un correspondant veut vous envoyer un message et ne veut surtout pas que celui-ci soit lisible par autrui. Il encode donc son message en utilisant la clé publique qu'il récupère sur votre répertoire. Une fois le message encodé, il vous l'envoie. Maintenant vous utilisez votre clé privée pour le décoder. Comme vous êtes seul à posséder cette clé, en théorie vous seul pouvez décoder le message. Pour qu'un tel système fonctionne, il faut trois prérequis : i) il doit être relativement simple de générer les deux clés ; ii) connaissant la clé publique, il doit être extrêmement compliqué de trouver une clé privée correspondante ; iii) l'encodage des messages par la clé publique et le décodage par la clé privée doivent être rapides.

RSA vérifie ces conditions. Il utilise deux principes simples de théorie des nombres : i) il est extrêmement difficile de décomposer un grand nombre en produit de nombres premiers, par contre, connaissant des nombres premiers, il est très facile de calculer leur produit ; ii) connaissant un grand nombre, il est extrêmement difficile de calculer sa fonction d'Euler, par contre, connaissant la décomposition d'un nombre en facteurs premiers, il est très facile de calculer sa fonction d'Euler. L'idée de RSA est d'utiliser les opérations faciles pour paramétrer le système cryptographique et de ne laisser aux cryptanalystes (ceux qui veulent décrypter des messages qui ne leur sont pas adressés) que les opérations extrêmement complexes.

Bon, maintenant que j'ai embrouillé tout le monde, nous allons voir comment ça marche, mais, d'abord, quelques rappels élémentaires de théorie des nombres (moi-même, je les ai découverts pas plus tard qu'hier) : un nombre entier x strictement positif est premier si et seulement si ses seuls diviseurs sont ± 1 et $\pm x$. Deux nombres x et y sont relativement premiers entre eux s'il n'ont aucun facteur premier en commun, autrement dit si leur PGCD est égal à 1. Arithmétique modulaire : soit a un nombre entier positif ou nul et b un nombre entier strictement positif, alors si $a = qb + r$, où $0 \leq r < b$ et $q = \lfloor a/b \rfloor$, $a \equiv r \bmod b$. Les propriétés de l'arithmétique modulaire correspondent grosso modo aux propriétés des opérations classiques sur les entiers (à part le calcul d'inverses) :

propriété	expression
arithmétique	$[(a \bmod n) + (b \bmod n)] \bmod n = (a + b) \bmod n$
	$[(a \bmod n) - (b \bmod n)] \bmod n = (a - b) \bmod n$
	$[(a \bmod n) \times (b \bmod n)] \bmod n = (a \times b) \bmod n$
commutativité	$(a + b) \bmod n = (b + a) \bmod n$
	$(a \times b) \bmod n \equiv (b \times a) \bmod n$
associativité	$[(a + b) + c] \bmod n \equiv [a + (b + c)] \bmod n$
	$[(a \times b) \times c] \bmod n \equiv [a \times (b \times c)] \bmod n$
distributivité	$[a \times (b + c)] \bmod n \equiv [(a \times b) + (a \times c)] \bmod n$
élément neutre	$(0 + a) \bmod n = a \bmod n$
	$(1 \times a) \bmod n = a \bmod n$

La fonction d'Euler d'un nombre entier x, notée $\phi(x)$, correspond au nombre de nombres, inférieurs strictement à x, qui sont relativement premiers avec x. C'est une fonction extrêmement cahotique et, d'une manière générale, il est très compliqué de la calculer pour un grand nombre. Cependant, il existe quelques cas où elle est très simple

à calculer : pour n'importe quel nombre premier, elle vaut le nombre premier moins 1. Et pour le produit de deux nombres premiers pq, elle vaut $(p-1)(q-1)$. Outre sa fonction, Euler nous a aussi légué un beau théorème qui dit que si x est un nombre premier et si a est un nombre positif non divisible par x, alors $a^{x-1} \equiv 1 \bmod x$. Autrement dit, $a^{\phi(x)} \equiv 1 \bmod x$. De cette formule, on en déduit que pour n'importe quel nombre entier k strictement positif, $a^{k\phi(x)} \equiv 1 \bmod x$, ou encore $a^{k\phi(x)+1} \equiv a \bmod x$.

Passons maintenant au principe de fonctionnement de RSA :

1/ Choisissez deux très grands nombres premiers (dans cet exercice, nous traiterons des entiers définis sur 512 bits), appelons-les p et q. Soit $n = pq$. Calculez $\phi(n) = (p-1)(q-1)$.

2/ choisissez aléatoirement un petit nombre entier e de telle sorte qu'il soit relativement premier avec $\phi(n)$. Lorsque e est choisi, votre clé publique est le couple (n, e).

3/ il vous faut maintenant une clé privée afin de décoder les messages qui vous sont envoyés : cette clé, appelée d, doit vérifier la formule suivante : $ed = k\phi(n) + 1$, pour un certain k. Cela revient à dire que $d \times e \equiv 1 \bmod \phi(n)$.

4/ Considérons maintenant l'encodage : le texte à encoder est séparé en blocs m_i dont la taille doit impérativement être inférieure à $\phi(n)$. Pour chaque bloc m_i, que l'on considère comme un nombre stocké sur un grand nombre de bits, on calcule $C_i \equiv (m_i)^e \bmod n$. Le nombre C_i est alors le texte encrypté, et c'est celui-ci que l'on envoie.

5/ Considérons la phase de décodage : $(C_i)^d = ((m_i)^e)^d \bmod n = (m_i)^{de} \bmod n = (m_i)^{k\phi(n)+1} \bmod n$. Par conséquent, d'après le théorème d'Euler, $(C_i)^d = m_i \bmod n = m_i$. Donc un simple calcul modulaire permet de retrouver m_i.

Voyons maintenant l'implémentation de notre cryptosystème. C'est à partir de maintenant qu'il faut paniquer. Nos entiers seront stockés dans des tableaux de 512 bits. On définira le type suivant pour les représenter :

```
typedef int* bignum;
```

1/ Écrivez une fonction `bignum alloue()` qui alloue un bignum ayant pour valeur le nombre entier 0. Écrivez une fonction `void desalloue(bignum nombre)` qui, comme son nom l'indiquait furieusement, désalloue un `bignum`.

2/ Écrivez les fonctions d'addition, de soustraction, de multiplication, modulo un bignum.

3/ Écrivez une fonction qui prend en argument m_i, e et n et qui renvoie le bignum correspondant à $(m_i)^e \bmod n$.

4/ Écrivez une fonction qui calcule le PGCD de deux bignums. Vous pourrez vous inspirer de l'exercice 28. En déduire une fonction qui, pour un bignum n premier retourne un bignum e relativement premier avec $\phi(n)$.

5/ Vous allez maintenant écrire une fonction qui, pour un e et un n donnés, renvoie un nombre d tel que $ed \equiv 1 \bmod \phi(n)$. En fait, cela revient à calculer $d = e^{-1} \bmod \phi(n)$. L'algorithme pour réaliser cela est une variation du calcul du PGCD :

```
00  fonction inverse(e,phi)
01  (X1, X2, X3) ← (1, 0, phi) ; (Y1, Y2, Y3) ← (0, 1, e)
02  tant que TRUE faire
02     si Y3 == 0 alors X3 = PGCD(e, phi), pas d'inverse, exit
03     si Y3 == 1 alors Y3 = PGCD(e, phi), Y2 = d^-1 mod phi, return Y2
04     Q = ⌊X3/Y3⌋
05     (T2, T2, T3) ← (X1 − QY1, X2 − QY2, X3 − QY3)
06     (X1, X2, X3) ← (Y1, Y2, Y3)
07     (Y1, Y2, Y3) ← (T2, T2, T3)
08  fin tant que
```

6/ Écrivez une fonction qui teste si un bignum x est premier. Comment déterminer si un nombre x est premier ou non ? L'idée la plus simple pour cela est de s'assurer qu'aucun nombre strictement supérieur à 2 et inférieur à $\sqrt{x}$ ne divise x. Malheureusement, cela prend beaucoup trop de temps pour des grands nombres. L'algorithme que nous allons utiliser va donc s'appuyer sur deux propriétés des nombres premiers : i) si x est premier et impair, alors l'équation d'inconnue y : $y^2 \equiv 1 \bmod x$ n'a que deux solutions : $y \equiv 1$ et $y \equiv -1$. Par conséquent si, en résolvant cette équation on tombe sur une racine de valeur absolue différente de 1, on sait que le nombre x n'est pas premier. L'inverse n'est malheureusement pas vrai. ii) La deuxième propriété est encore le théorème d'Euler : puisque $a^{x-1} \equiv 1 \bmod x$ lorsque x est premier, si on calcule $a^{x-1} \bmod x$ et qu'on ne trouve pas 1, c'est que x n'est pas premier. De ces deux propriétés, Miller et Rabin en ont déduit l'algorithme suivant que vous allez programmer :

```
00  fonction Witness(a,n)
01  soit b_k b_{k-1} ... b_0 la représentation binaire de n - 1
02  d ← 1
03  pour i variant de k à 0 faire
04     x ← d
05     d ← (d × d) mod n
06     si d == 1 et x ≠ 1 et x ≠ n - 1
07        alors return TRUE
08     si b_i == 1
09        alors d ← (d × a) mod n
10  finpour
11  si d ≠ 1
12     alors return TRUE
13  return FALSE
```

Comme on peut le remarquer les lignes 03 à 09 de la boucle for finissent par calculer $d = a^{n-1}$, d'où le test des lignes 11–13 qui correspond à tester le théorème d'Euler. Quand au test de la ligne 06, il vérifie que l'on n'a pas $d^2 \equiv 1 \bmod n$ et, en même temps, $d \neq \pm 1 \bmod n$.

Lorsque la fonction Witness vous renvoie TRUE, vous pouvez être sûr que n n'est pas premier. Par contre, si elle renvoie FALSE, vous n'êtes sûr de rien. Il s'avère que la probabilité que Witness retourne FALSE est inférieure à 0,5. Donc Miller et Rabin ont proposé de lancer k fois leur algorithme avec k valeurs de a différentes (inférieures à n quand même), assurant ainsi que si l'on obtient à chaque fois FALSE, on a une probabilité supérieure à $1 - 2^{-k}$ que le nombre soit effectivement premier. Écrivez une fonction `bool is_not_prime(bignum *nombre)` qui lance, disons 20 fois au maximum, la fonction Witness et qui renvoie un booléen indiquant la primalité de `nombre`.

7/ Écrivez une fonction qui choisisse au hasard un grand nombre premier et le renvoie sous forme de `bignum`. Pour réaliser cela, c'est très simple : appelons $\pi(x)$ le nombre de nombres premiers inférieurs ou égaux à x. Il s'avère que $\lim_{x\to+\infty} \pi(x) = \frac{x}{\ln x}$. Cela veut dire qu'en gros à moins de $\ln x$ autour de x, on peut raisonnablement espérer trouver un nombre premier. Or, même pour des nombres très grands, $\ln x$ n'est pas monstrueusement élevé : pour $x = 2^{200}$, $\ln(x) = 140$. Donc, en parcourant les 140 nombres se trouvant après x, on est à peu près sûr de trouver un nombre premier. On rabaisse ce nombre à 70 en remarquant qu'à part 2 (que l'on ne peut décemment qualifier de grand nombre), tous les nombres premiers sont impairs.

8/ Écrivez une fonction `void encode(char *texte, bignum *e, bignum *n, bignum *code, int *taille)` qui prend en argument une chaîne de caractère `texte`, qui encode ce texte selon la clé publique (e, n), et qui stocke le résultat dans un tableau (à allouer) de bignum `code`, contenant `taille` bignums (il faut bien un tableau puisque le texte est séparé en blocs d'au plus 512 bits). De même écrire une fonction `void decode(bignum *e, bignum *n, bignum *code, int taille, char *texte)` qui décode un texte.

9/ Enfin, écrivez un programme qui propose à l'utilisateur le menu suivant :

- a/ créer un couple clé publique/clé privée et sauvegarder chacune des clés dans un fichier dont le nom sera spécifié par l'utilisateur.
- b/ demander à l'utilisateur le nom d'un fichier texte, celui d'un fichier contenant une clé publique, encoder le fichier texte selon la clé et sauvegarder le résultat dans un fichier dont le nom sera spécifié par l'utilisateur.
- c/ demander à l'utilisateur le nom d'un fichier crypté, celui d'un fichier contenant une clé privée, décoder le fichier crypté selon la clé et sauvegarder le résultat dans un fichier dont le nom sera spécifié par l'utilisateur.

2.9 Préprocesseur et lignes de compilation

La plupart des exercices de cette sous-section proviennent des exercices de licence d'informatique de Paris 6 de Christian Queinnec.

Exercice 108 Soit le programme suivant :

```
int main (int argc, char *argv[]) {
  char *fmt = "%s\n";
  char *p = ARGV0;
  printf(fmt, *(ARGV1));
  return EXIT_SUCCESS;
}
```

On compile une première fois ce programme avec les définitions suivantes -DARGV0=fmt -DARGV1=argv puis avec les définitions -DARGV0=argv -DARGV1=p. Que donnent les exécutions de ces deux programmes?

Exercice 109 Écrivez une macro `swap` qui prend un type et deux arguments de ce type en entrée, et qui les inverse. Par exemple : `swap(int,x,y)` place la valeur de y dans x et celle de x dans y. Réecrire la macro en enlevant le type de manière à ce qu'elle fonctionne pour tous les nombres (float, long, int, etc).

Exercice 110 Définissez les macros BEGIN, END, IF, THEN, ELSE et ENDIF (sans tenir compte du fait qu'il n'y a pas de ENDIF en Pascal) pour qu'un texte de syntaxe superficiellement équivalente à Pascal soit traduit en C. Vous placerez les macros dans un fichier `pascal.h`. Vous pourrez vous exercer avec le fichier test/pascal que voici :

```
#include "c/pascal.h"

BEGIN
        IF x>2
        THEN x
        ELSE    IF x>y THEN y ENDIF
        ENDIF
END
```

Exercice 111 Définissez la macro C ODD dans le fichier odd.h pour tester si un nombre est impair et la macro EVEN dans le fichier even.h pour tester si un nombre est pair. S'arranger pour que dès que l'un d'entre eux est inclus, l'autre le soit aussi.

Exercice 112 Étudiez comment compiler le programme `fact.c` que voici afin de pouvoir l'exécuter. En d'autres termes, déterminez les options de compilation nécessaires pour que le produit de la compilation contienne la factorielle et le point d'entrée main.

```
/* $Id: fact.c,v 1.5 1999/11/06 19:40:16 queinnec Exp $
 * http://videoc.lip6.fr/
 *
 * La factorielle eternellement recommencee. Ce programme est etrange
 * pour les besoins des exercices.
 */

#if !defined(WITHOUT_FACT)

int
fact (int n)
{
```

```
  if ( n == 1 ) {
    return n;
  } else {
    return n * fact(n-1);
  }
}

#endif /* WITHOUT_FACT */

#if defined(WITH_MAIN)

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

int
main (int argc,
      char *argv[])
{
  int n = 10;

  printf("Fact(%d) = %d\n", n, fact(n));
  exit(EXIT_SUCCESS);
}

#endif /* WITH_MAIN */

/* end of fact.c */
```

Exercice 113 Soit le labyrinthe défini par les fichiers `maze.h` et `maze.c` à propos desquels on remarquera qu'ils ne contiennent aucune ligne de code :

```
/* $Id: maze.h,v 1.1 1996/11/23 14:48:13 queinnec Exp $
 * ftp://ftp.inria.fr/INRIA/Projects/icsla/WWW/Queinnec.html
 *
 * Les types decrivant les noeuds d'un graphe.
 */

typedef struct Node0 {
  unsigned int arity;
} Node0;

typedef struct Node1 {
  unsigned int arity;
  Node0*       son1;
} Node1;

typedef struct Node2 {
  unsigned int arity;
  Node0*       son1;
  Node0*       son2;
} Node2;

typedef struct Node3 {
  unsigned int arity;
```

```
  Node0*       son1;
  Node0*       son2;
  Node0*       son3;
} Node3;

extern Node0* maze_entry;
extern Node0* maze_exit;

/* end of maze.h */

/* $Id: maze.c,v 1.1 1996/11/23 14:48:01 queinnec Exp $
 * ftp://ftp.inria.fr/INRIA/Projects/icsla/WWW/Queinnec.html
 *
 * Un petit labyrinthe statiquement defini. Pas une seule ligne de code ici!
 */

#include "maze.h"

#define DefineNode0(name)\
static Node0 node##name = {     \
  0 }

#define DefineNode1(name, sonA)\
static Node1 node##name = {     \
  1, (Node0*) &node##sonA }

#define DefineNode2(name, sonA, sonB)\
static Node2 node##name = {     \
  2, (Node0*) &node##sonA, (Node0*) &node##sonB }

#define DefineNode3(name, sonA, sonB, sonC)\
static Node3 node##name = {     \
  3, (Node0*) &node##sonA, (Node0*) &node##sonB, (Node0*) &node##sonC }

static Node0 node22;
static Node1 node0, node3, node4, node6, node8, node9, node12,
             node13, node15, node17, node20, node21, node23, node24;
static Node2 node2, node5, node7, node10, node11, node14, node18,
             node19, node25, node26;
static Node3 node1, node16;

DefineNode1(0,21);
DefineNode3(1,2,3,15);
DefineNode2(2,2,7);
DefineNode1(3,6);
DefineNode1(4,13);
DefineNode2(5,14,8);
DefineNode1(6,16);
DefineNode2(7,17,8);
DefineNode1(8,24);
DefineNode1(9,4);
DefineNode2(10,9,13);
DefineNode2(11,12,4);
```

```
DefineNode1(12,4);
DefineNode1(13,5);
DefineNode2(14,19,24);
DefineNode1(15,18);
DefineNode3(16,11,17,25);
DefineNode1(17,15);
DefineNode2(18,22,19);
DefineNode2(19,20,13);
DefineNode1(20,10);
DefineNode1(21,1);
DefineNode0(22);
DefineNode1(23,22);
DefineNode1(24,23);
DefineNode2(25,17,16);
DefineNode2(26,25,24);

Node0* maze_entry = (Node0*) &node0;
Node0* maze_exit  = (Node0*) &node23;

/* end of maze.c */
```

Écrivez un programme explorant ce labyrinthe, de son entrée définie par `maze_entry` à sa sortie `maze_exit`.

Exercice 114 Où est l'erreur dans ce programme ? Le but est de vérifier que les entiers longs sont suffisamment grands pour contenir des adresses de la machine.

```
int check_size()
{
#if sizeof(unsigned long int) >= sizeof(void*)
    return 1;
#else
    return 0;
#endif
}
```