tout n'est pas la...

Introduction à la Programmation

http://dept-info.labri.u-bordeaux.fr/~wacren/IP

Pierre-André Wacrenier@labri.u-bordeaux.fr
Bureau 115, Poste 69 08

Objectifs

Maîtriser la programmation en langage C sous UNIX

- ☞ connaître les éléments du C
  - ➼ déclaration, expression, instruction
  - ➼ mise en oeuvre d'algorithmes classiques
- ☞ utiliser un environnement
  - ➼ outils GNU (EMACS, GCC, MAKE, GDB)
  - ➼ bibliothèque C standard
- ☞ adopter une méthode
  - ➼ présentation des programmes
  - ➼ découpage des programmes en modules
  - ➼ traitement des «anomalies»

Références Méthodologie de la programmation en C — J.P. Braquelaire

Le langage C, norme ANSI — B. Kernighan & D. Richtie

Programme = Donnée

1936 Machine universelle d'Alan Turing (GB)

1945 ENIAC, calculateur électronique par Eckert et Mauchly (USA)
Architecture de Von Neumman
PLANKALKUL, premier langage de programmation par Zuse (D)

1947 MADM, Premier ordinateur universel (GB)

1950 Premières commercialisations EDSAC (GB) et EDVAC (USA)

1953 Premiers compilateurs algébriques

Avantages des langages de «haut niveau» : *abstraction, communication, portabilité*
Améliorer la productivité du programmeur, favoriser le travail en équipe.

	fortran	traitement scientifique (1957)
	algol	description des algorithmes (IAL, 1958)
5 langages majeurs	**cobol**	traitement de fichiers (1959)
	lisp	traitement de listes (1960)
	simula	langage orienté objet (1962)

IAL 58 ⟶ ALGOL 60 ⟶ CPL 63 ⟶ BCPL 69 ⟶ B 70 ⟶ C 72

1972 Création par Dennis Richtie (BELL) pour l'implantation du système UNIX sur le PDP/11.

1978 Définition du langage C par la publication du «K&R» : «*The C programming language*» de B. Kernighan et D. Ritchie.

1989 Le langage C-ANSI est créé après un travail de 6 années de normalisation.

C, le bas de gamme du haut niveau

Haut niveau : typage, structures de contrôles et de données.

Bas de gamme : pas de manipulation globale de données complexes (conversion, tableaux, listes...), pas d'expression du parallélisme.

Extensions du langage C : orienté objet (C++, Objective-C), parallélisme massif.

Premier programme

```
1 /* Affiche les nombres premiers de 1 a 40 */
2 #include <stdio.h>
3  main (){
4   long i, j ;
5   printf("2␣3") ; /* on regle le cas pair et celui de 3*/
6   for (i = 3 ; i <= 40 ; i+=2){
7     for (j = 3 ; i % j != 0 && i > j * j ; j += 2) ;
8     if (i % j != 0) printf ("␣%ld", i);
9   }
10 }
```

```
$ gcc premier1.c
$ ./a.out
2 3 5 7 11 13 17 19 23 25 29 31 35 37$  %$
```

Le programme donne le résultat attendu... 1/20 à l'exam.

```
$ gcc -Wall premier1.c
premier1.c:3: warning: return-type defaults to ‘int’
premier1.c: In function ‘main’:
premier1.c:10: warning: control reaches end of non-void function
$
```

Second programme

```c
#include <stdio.h>
#include <stdlib.h>
#define BORNE_INF 2
#define BORNE_SUP 40

/* premier(x) = x est un nombre premier */
int
premier (const unsigned long x)
{
  unsigned long j;

  if (x <= 3)
    return x == 2 || x == 3;
  if (x % 2 == 0)
    return 0;
  for (j = 3; x % j && j * j < x; j += 2)
    ;
  return (x % j != 0);
}
```

Second programme

```

int
main ()
{
  unsigned long i;

  /* 2 est le seul pair premier */
  if (BORNE_INF <= 2 && BORNE_SUP >= 2)
    printf (" 2");

  /* initialiser i au plus petit nombre impair >=  BORNE_INF */
  for (i = BORNE_INF + 1 - (BORNE_INF & 1); i <= BORNE_SUP; i+=2)
    {
      if (premier (i))
        printf (" %ld", i);
    }
  return EXIT_SUCCESS;
}
```

Vie du logiciel

❶ Expression d'une demande d'automatisation

❷ Analyse de la situation

❸ Rédaction d'un cahier des charges (spécifications)

❹ Contractualisation (appels d'offre, négociations, signatures)

❺ Conception d'une architecture (analyse détaillée, choix des outils)

❻ Implémentation (codage)

❼ Test unitaire (intra modulaire)

❽ Test d'intégration (inter modulaire)

❾ Livraison (formation, implantation du système)

Maintenance corrective, maintenance évolutive.

Finalité obtenir au meilleur coût des applications fonctionnelles

Objectifs

- favoriser la maintenance corrective ;
- favoriser la maintenance évolutive ;
- favoriser le développement de nouvelles applications.

But écrire du code

❶ lisible,

❷ maintenable,

❸ portable,

❹ extensible,

❺ réutilisable.

Utiliser un environnement performant pour appliquer des normes et une méthode de programmation : la **modularisation** des programmes – abstraction des données, factorisation du code, masquage de l'implémentation – dans un environnement performant.

```c
#include <stdlib.h>
#include <stdio.h>
#define UN_FRANC_EN_EURO 0.15244902
#define PREMIER 10
#define DERNIER 90
#define PAS 10

double
franc_vers_euro(double somme)
{
  return somme * UN_FRANC_EN_EURO ;
}

int
main()
{
  double franc ;

 for(franc = PREMIER ; franc <= DERNIER ; franc = franc + PAS)
   printf("%.2f F = %.2f e\n", franc, franc_vers_euro(franc))
 return EXIT_SUCCESS ;
}
```

Exemple de compilation

```
$ gcc -W -Wall franc_vers_euro.c
$ franc_vers_euro.c: In function ‘main’:
$ franc_vers_euro.c:21: parse error before ‘return’
$ franc_vers_euro.c:22: warning: control reaches end of non-void function
```

correction...

```
$gcc -W -Wall franc_vers_euro.c
$./a.out
10.00 F = 1.52 e
   ...
80.00 F = 12.20 e
90.00 F = 13.72 e
$ gcc -Wall -o table_fr_euro franc_vers_euro.c
$./table_fr_euro
10.00 F = 1.52 e
   ...
```

Les étapes de la compilation

prétraitement expansion des directives par le préprocesseur.

compilation traduction en assembleur (.s).

assemblage traduction en langage machine (.o).

édition des liens mise en correspondance des différents fichiers objets et génération d'un exécutable (a.out).

Principales options du compilateur gcc (`man gcc`):

-E exécution du seul préprocesseur.

-S génération du code assembleur.

-c génération du code intermédiaire (objet).

-W -Wall contrôle syntaxique poussé.

-o fichier force le nom du fichier en sortie.

-g génération d'informations utiles au débogueur gdb.

-pg génération de code pour l'évaluateur de performance gprof.

-On optimisation de niveau n.

-MM fichiers affiche la liste de dépendance de chaque fichier.

Nécessité de définir des unités (briques) de programmation.

Découpage du fichier `franc_vers_euro.c` en 2 fichiers : `monnaie.c` et `table_franc_euro.c`

`monnaie.c` :

```
#define UN_FRANC_EN_EURO 0.15244902

double
franc_vers_euro(double somme)
{
  return somme * UN_FRANC_EN_EURO ;
}

double
euro_vers_franc(double somme)
{
  return somme / UN_FRANC_EN_EURO ;
}
```

`franc_vers_euro.c` :

```
#include <stdlib.h>
#include <stdio.h>
#include "monnaie.c"
#define PREMIER 10
#define DERNIER 90
#define PAS 10

int
main()
{
  double franc ;

 for(franc = PREMIER ; franc <= DERNIER ; franc = franc + PAS)
   printf("%.2f␣F␣=␣%.2f␣e\n", franc, franc_vers_euro(franc)) ;
 return EXIT_SUCCESS ;
}
```

Compilation par `gcc -W -Wall table_franc_euro.c`

Les utilisateurs du module monnaie ont accès aux détails de son implémentation.

Séparer la définition (le «quoi») de la réalisation (le «comment») afin de *libérer* à la fois :

– les utilisateurs du module des détails de l'implémentation,

– les programmeurs du module des détails de l'utilisation.

Intérêt vital pour la maintenance, la réutilisation et la communication.

En C l'interface est placé dans le fichier *en-tête source* appelé «le point h» :

`monnaie.h`

```
#ifndef MONNAIE
#define MONNAIE
extern double franc_vers_euro(double somme) ;
extern double euro_vers_franc(double somme) ;
#endif /* MONNAIE */
```

Le fichier `table_franc_euro.c` doit être modifié :

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

Compilation séparée

```
$ gcc -W -Wall table_franc_euro.c
/tmp/ccQkN6iC.o: In function ‘main’:
/tmp/ccQkN6iC.o(.text+0x3b): undefined reference to ‘franc_vers_euro’
collect2: ld returned 1 exit status

$ gcc -W -Wall -c table_franc_euro.c
$ gcc -W -Wall -c monnaie.c
$ gcc table_franc_euro.o monnaie.o -o ma_table
$ ./ma_table
10.00 F = 1.52 e
10.00 F = 1.52 e
20.00 F = 3.05 e
30.00 F = 4.57 e
40.00 F = 6.10 e
50.00 F = 7.62 e
60.00 F = 9.15 e
70.00 F = 10.67 e
80.00 F = 12.20 e
90.00 F = 13.72 e
```

Compilation séparée

modification du fichier `monnaie.c` :

```
#include "monnaie.h"
#define UN_EURO_EN_FRANC  6.55957

double
franc_vers_euro(double somme)
{
  return somme / UN_EURO_EN_FRANC ;
}

double
euro_vers_franc(double somme)
{
  return somme * UN_EURO_EN_FRANC ;
}
```

Compilation du *seul* module `monnaie.c` puis édition des liens :

```
$ gcc -W -Wall -c monnaie.c
$ gcc table_franc_euro.o monnaie.o -o ma_table
$ ./ma_table
10.00 F = 1.52 e
...
```

```
# makefile table_franc_euro
CC = gcc
CFLAGS = -W -Wall -g

table_franc_euro: table_franc_euro.o monnaie.o
table_franc_euro.o: table_franc_euro.c monnaie.h
monnaie.o: monnaie.c monnaie.h

clean:
	rm -f *.o *~
tar:
	rm -f *.o *~ table_franc_euro
	tar -c --remove-files -f LISTE_PREMIER.tar .
	gzip LISTE_PREMIER.tar
# ^---tabulation
```

```
$ make
gcc    -W -Wall -g   -c table_franc_euro.c -o table_franc_euro.o
gcc    -W -Wall -g   -c monnaie.c -o monnaie.o
gcc      table_franc_euro.o monnaie.o   -o table_franc_euro
$
```

```
$ touch monnaie.c
$ make
gcc    -W -Wall -g   -c monnaie.c -o monnaie.o
gcc      table_franc_euro.o monnaie.o   -o table_franc_euro
$
$ touch table_franc_euro.c
$ make
gcc    -W -Wall -g   -c table_franc_euro.c -o table_franc_euro.o
gcc      table_franc_euro.o monnaie.o   -o table_franc_euro
$ touch monnaie.o
$ make
gcc      table_franc_euro.o monnaie.o   -o table_franc_euro
$
$ gcc -MM monnaie.c table_franc_euro.c
monnaie.o: monnaie.c monnaie.h
table_franc_euro.o: table_franc_euro.c monnaie.h
$
```

Tour d'horizon du langage C

❶ variables et types de base,

❷ expressions,

❸ instructions,

❹ fonctions, appels de fonctions,

❺ entrées / sorties,

❻ arguments de la fonction main(),

❼ conseils quant à la présentation des programmes.

Type = catégorie d'objet, ensemble de définition, format de donnée, fonction d'interprétation.

type	description	taille	MIN	MAX
short	entier court	2	-32 767 -1	32 767
int	entier	2 à 4	–	–
long	entier long	4	-2 147 483 647	2 147 483 647
float	flottant	4	10^{-37}	10^{37}
double	fl. double précision	8	10^{-307}	10^{308}
long double	fl. précision étendue	12	10^{-4931}	10^{4932}
char	caractère	1	-128	127
size_t	taille d'objet	(long)		in stddef
ptrdiff_t	distance entre adresses	(int)		in stddef
wchar_t	caractère large	4		

modificateur pour les types caractère et entiers: signed, unsigned.

Variable = zone particulière de la mémoire désignée par un nom (identificateur) et décrite par un type.

Déclaration :

type identificateur ⟨ = expression⟩ ⟨, identificateur ⟨ = expression ⟩ ⟩* ;

L'identificateur doit signifier l'objet mémorisé par la variable. Les commentaires sémantiques sont donc inutiles.

Exemples :

```
int nombre_de_lapins = 0 , nombre_de_chasseurs = 5 ;
char dernier_caractere_lu ;
double vitesse_initiale = 25.4 , distance_terre_lune ;
long population_active_francaise = 25755000 ;
float indice_cac40 = 4440.87 ;
```

Remarques utiliser une seule langue (dans ce cours, le français) ;
les indentificateurs doivent différer sur leurs 31 premiers caractères pour les objets locaux et sur leurs 6 premiers pour les objets exportés.

Expression primaire identificateur, constante, chaine de caractères, (expression).

Expression unaire préfixée (`-x *x`), postfixée (`x++ a->b a[b]`).

Expression binaire affectation, comparaison, logique, mathématique...

Expression ternaire conditionnelle.

Grammaire (cf. K&R page 241) :

```
expression
    expression-d'affectation
    expression , expression-d'affectation

expression-d'affectation
    expression-conditionnelle
    expression-unaire opérateur-d'affectation expression-d'affectation

expression-conditionnelle
    expression-ou-logique
    expression-ou-logique ? expression : expression-conditionnelle
```

Liste par priorité décroissante :

Opérateur	Asso.	Arité
`() []` `-> .`	G	- 2
`! ~ ++ -- + -` `(<type>) * & sizeof`	D	1
`* / %`	G	2
`+ -`	G	2
`<< >>`	G	2
`< <= > >=`	G	2
`== !=`	G	2

<table>
<tr><th>Opérateur</th><th>Asso.</th><th>Arité</th></tr>
<tr><td>&</td><td>G</td><td>2</td></tr>
<tr><td>^</td><td>G</td><td>2</td></tr>
<tr><td>|</td><td>G</td><td>2</td></tr>
<tr><td>&&</td><td>G</td><td>2</td></tr>
<tr><td>||</td><td>G</td><td>2</td></tr>
<tr><td>?:</td><td>D</td><td>3</td></tr>
<tr><td>= += -= *= /= %=
>>= <<= &= ^= |=</td><td>D</td><td>2</td></tr>
<tr><td>,</td><td>G</td><td>2</td></tr>
</table>

Sachant que $-$ et $+$ sont de même priorité et sont *associatifs de gauche à droite*, calculer la valeur de $3 - 5 + 2 - 2$
Algorithme : parenthéser les opérandes de l'opérateur le plus à gauche.

Sachant que $*$ est *prioritaire* par rapport à $+$, parenthéser $1 + 2 * 3 + 4 * 5$.

Algorithme : parenthéser les multiplications puis les additions.

Sachant que $=$ est associatif de droite à gauche, moins prioritaire que $+$ et que $a = b$ se réduit en b, parenthéser $x = 3 + 2$, $x = y = z$, $x = 1 + y = 1$.

Parenthéser $a = b + c\ ?\ 3 * 2 : 5 - 4$.

Insructions élémentaires

`instruction ::= ;`

`instruction ::= expression ;`

Structures de contrôle : Bloc, Test, Boucle tant-que, Itération, Aiguillage.

Bloc	**{**⟨`liste de déclarations`⟩⟨`liste d'instructions`⟩**}**
Liste d'instructions	`instruction` ⟨`liste d'instructions`⟩
Test	**if (**`expression`**)** `instruction` ⟨**else** `instruction`⟩
Boucle tant-que	**while (**`expression`**)** `instruction` **do** `instruction` **while (**`expression`**)**
Itération	**for (**⟨`expression`⟩;⟨`expression`⟩;⟨`expression`⟩**)** `instruction`
Aiguillage	**switch (**`expression`**)** **{** liste de choix ⟨**default :** liste d'instructions⟩**}**
Liste de choix	**case** `constante : liste d'instructions` ⟨`liste de choix`⟩

Test

if (expression) instruction ⟨**else** instruction⟩

```
...                                          # assembleur mips
if (x < 0)            lw   r1, [x]           # load word
  x = -x;             bgez r1, suite         # branch if greater or eq. 0
y = x + 1;            sub r1, 0, r1          #
...            suite:
                      add r2, r1, 1


...
if (x < 0)            lw   r1, [x]       #
  x = -x;             bgez r1, sinon     # sauter a sinon si r2
else                  sub  r1, 0, r1     #
  y = x + 1;          j suite            # passer a la suite
...           sinon:
                      add r2, r1, 1
              suite:
```

```
...
if (x < 0)
  x = -x;
else if (y > 0)
  {
    x = 2 * x - 2;
    y = 2 * y;
  }
else
  y = 6 * p ;
...
```

Difficultés :

```
if (x < 0)                 |if (x < 0)
  if (y < 0)               |   if (y < 0)
     x = x + y ;           |      x = x + y ;
  else                     |else
     z = x - y ;           |   z = x - y ;
...
if (x = 0)
   x = 1 ; /* instruction jamais executee */
```

Boucles tant-que

while (`expression`) `instruction`

```
...                                 lw   r1, [x]
while (x != 1000)                   beq  r1, 1000, suite
  x = x + 1 ;               debut:
...                                 add  r1, r1, 1
                                    bne  r1, 1000, debut
                            suite:
```

do `instruction` **while** (`expression`)

```
...                                 lw   r1, [x]
do                          debut:
  x = x + 1;                        add  r1, r1, 1
while (x != 1000)                   bne  r1, 1000, debut
...
```

Itération

for (⟨expression⟩;⟨expression⟩;⟨expression⟩) `instruction`

`for (initialisation; condition; progression) instruction`

```
 ...                                  lw  r1, [y]
for (i = 1; i < 1000; i++)            add r2, 0, 1          # i = 1
   y *= y;                            bgt r2, 1000, suite   #greater than
 ...                           debut:
                                      mul r1, r1, r1        # y *= y
                                      add r2, r2, 1         # i++
                                      ble r2, 1000, debut   #less or equal
                               suite:
```

Possibilité de faire plusieurs initialisations ou progressions :

`for (i = 1 , j = 2 ; i <12 ; j += 3 , i++ ) ...`

Interdiction d'ouvrir un bloc (et donc de déclarer une variable) dans la partie initialisation. cependant on recommande

```
for (i = 1; i < 12; i++)
 {
  int j;
  for (j = 1; j > 12; j++)
    ...
```

```
}
```

Aiguillage

switch (expression)

{ liste de choix ⟨**default**: liste d’instructions⟩**}**

liste de choix ::= **case** constante : liste d’instructions ⟨liste de choix⟩

```
  switch (car)           lb  r1, [car]  #load byte
   {                     beq r1, 65, L1 #ascii('a') = 65
     case 'a' :          beq r1, 97, L1 #ascii('A') = 97
     case 'A' :          beq r1, 66, L2
       x = 1 ;           beq r1, 98, L2
       break ;           j   L3
                     L1:
     case 'b' :          add r2, 0, 1    # x = 1
     case 'B' :          j   suite
       x = 2 ;       L2:
                         add r2, 0, 2    # x = 2
     default :       L3:
       x = 0 ;           add r2, 0, 0    # x = 0
   }              suite:
```

Fonction sans paramètre


```
void
bonjour (void)
{
  printf ("bonjour") ;
}

double
pi (void)
{
    return 3.1415926535898;
}

int
main()
{
  double surface = 2 * pi ();
  bonjour ();
  return EXIT_SUCCESS ;
}
```

Fonction avec paramètre


```
double
moyenne(double x, double y)
{
  return (x + y) / 2;
}

int
main()
{
  double a = 10 , b = 20 , moyenne_ab ;

  moyenne_ab = moyenne (a + b, a * b) ;
  a = moyenne( moyenne (10, a * a), a) ;
  return EXIT_SUCCESS :
}
```

Paramètres formels : `y` et `x`.

Paramètres effectifs : `a + b` et `a * b`.

Le *passage* des paramètres se fait toujours par valeur.

L'opérateur unaire & permet de calculer l'adresse d'une variable.

La fonction printf()

Un format est une chaîne de caractères contenant des directives de formatage introduites par le caractère %.

Fonction `printf(format, liste d'arguments)`

```
printf("un int %d un long %d un float %f \n un double %.20f \n "
        "un caractère %c une chaine %s",
         12, 123000, 12.333, 1.0 / 3, 'y',"pourquoi pas");
```

```
un int 12 un long 123000 un float 12.333000
un double 0.33333333333333331483
un caractère u une chaine pourquoi pas
```

Le nombre d'arguments n'est pas borné.

Fonction `scanf(format, liste d'arguments)`

```
 scanf("%d %ld %f %lf %c", &entier, &entier_long,
       &flottant, &flottant_double , &caractere);
```

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

argc : nombre d'arguments + 1 (*argument count*)

argv : vecteur d'arguments (*argument vector*)

char *argv[] : vecteur (tableau) de chaînes de caractères.

argv[0] désigne le nom de l'exécutable, **argv[i]** désigne le i^e^ paramètre.

```c
#include <stdio.h>  /* Hal.c */
#include <stdlib.h>
int
main(int argc, char *argv[])

    printf("Mon nom est %s.\n", argv[0]);
    return EXIT_SUCCESS;
}
```

```
$ gcc -W -Wall -o Hal Hal.c
$Hal
Mon nom est Hal
$mv Hal Personne
$Personne
Mon nom est Personne
```

Exemple: la fonction main de liste_premier.c

```c
static void
usage ()
{
  assert (nom_commande);
  fprintf (stderr, "Usage: %s entier_positif1 entier_positif2 \n",
           nom_commande);
  exit (EXIT_FAILURE);
}

int
main (int argc, char *argv[])
{
  unsigned long  borne_inf, borne_sup;

  nom_commande = argv[0];
  if (argc != 3)
    usage ();
  borne_inf = valeur_chaine_en_ul (argv[1], usage);
  borne_sup = valeur_chaine_en_ul (argv[2], usage);
  imprimer_premier(borne_inf, borne_sup);
  return EXIT_SUCCESS;
}
```

Conseils de rédaction

- ☞ Utiliser des identificateurs significatifs.
- ☞ Améliorer la lisibilité plutôt que commenter.
- ☞ Définir et initialiser les variables au plus près de leur utilisation.
- ☞ Limiter la portée des variables.
- ☞ Respecter une norme d'indentation (commande indent sous unix).
- ☞ Réutiliser le code existant.
- ☞ Traquer le code dupliqué.
- ☞ Récrire le code plusieurs fois.

1. Type d'objets
2. Type de fonctions
3. Types incomplets
4. Visibilité des variables
5. Implémentation des variables

Type d'objet détermine l'ensemble des valeurs que peut prendre la zone mémoire associée à l'objet et la façon d'interpréter celle-ci.

Type de fonction détermine le type du résultat retourné, le nombre et le type des paramètres de la fonction.

Type incomplets
- vecteurs non dimensionnés (en paramètre de fonction)
- structures incomplètes (pour masquer l'implémentation d'une structure)
- type void (déclaration de procédure, de fonction sans paramètre, pointeur générique).

Les types permettent d'améliorer la lisibilité des programmes :

```
typedef definition_du_type identificateur_du_type_defini ;
```

```
typedef double euro, franc ;
euro franc_vers_euro(franc somme) ;
```

Un identificateur doit être différent d'un mot *réservé* (voir Braquelaire p 48, 282, 323) :

- mots-clefs (while)
- fonction d'une bibliothèque standard (printf)
- commençant par is, to, str, mem,...

Un identificateur est un mot de
`{a,...,z,A,...,Z,_}{a,...,z,A,...,Z,_,0,...,9}`*

[mot commençant par une lettre non accentuée ou un blanc souligné (*underscore*) suivi d'un certain nombre de lettres, de chiffres ou de blancs soulignés]

Les identificateurs doivent différer 2 à 2 sur leurs 31 premiers caractères pour les objets locaux et sur leurs 6 premiers pour les objets exportés.

Règle locale : un idendificateur doit contenir au moins une lettre minuscule, on évitera même l'emploi de lettres majuscules.

Types entiers signés : `short, int, long`

Codés en complément à 2 (voir Braquelaire p51)

Types entiers non signés :`unsigned short, unsigned int, unsigned long`

Dans `/usr/include/limits.h` on trouve la définition des constantes :

```
UCHAR_MAX CHAR_MAX SHRT_MIN           INT_MIN            LONG_MIN
          CHAR_MIN SHRT_MAX USHRT_MAX INT_MAX  UINT_MAX LONG_MAX ULONG_MAX
```

Constantes entières :

```
#ifdef __alpha__
#define ULONG_MAX    18446744073709551615UL
```

L'arithmétique est exacte modulo `UTYPE_MAX+1` pour `UTYPE` non signé.

Valeur absolue

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

long
valeur_absolue (long x)
{
 if (x < 0)
   return -x ;
 else
   return x ;
}

int
main()
{
 printf ("|%ld|␣=␣%ld\n", -5L, valeur_absolue(-5L)) ;
 printf ("|%ld|␣=␣%ld", LONG_MIN, valeur_absolue(LONG_MIN)) ;
 return EXIT_SUCCESS ;
}
```

```
|-5| = 5
|-2147483648| = -2147483648
```

Valeur absolue

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

unsigned long
valeur_absolue (long x)
{
  if (x < 0)
    return ((unsigned long) -(x + 1)) + 1;
  else
    return x ;
}

int
main()
{
  printf ("|%ld| = %ld\n", -5L, valeur_absolue(-5L)) ;
  printf ("|%ld| = %lu", LONG_MIN, valeur_absolue(LONG_MIN)) ;
  return EXIT_SUCCESS ;
}
```

```
|-5| = 5
|-2147483648| = 2147483648
```

Factorielle

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

unsigned long
factorielle (short x)
{
  short i ;
  unsigned long resultat = 1;

  for(i = 2; i <= x ; i++)
    resultat = resultat * i;
  return resultat;
}
```

0! = 1 1! = 1 2! = 2 3! = 6 4! = 24 5! = 120 6! = 720 7! = 5040 8! = 40320

9! = 362880 10! = 3628800 11! = 39916800 12! = 479001600

13! = 1932053504

14! = 1278945280

15! = 2004310016

Factorielle bis

```c
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#define FACTORIELLE_MAX 13

unsigned long
factorielle (short x)
{
  short i ;
  unsigned long resultat = 1 ;

  assert(x <= FACTORIELLE_MAX) ;
  for(i = 2; i <= x ; i++)
    resultat = resultat * i ;
  return resultat ;
}
```

```
0! = 1 1! = 1 2! = 2 3! = 6 4! = 24 5! = 120 6! = 720 7! = 5040 8! = 40320
9! = 362880 10! = 3628800 11! = 39916800 12! = 479001600 13! = 1932053504
a.out: facto_sur.c:12: factorielle:  l'assertion ‘x <= 13’ a échoué.
Aborted (core dumped)
```

Permet d'augmenter la lisibilité des programmes en rendant explicite certains codages:

```
                    /* Déclaration d'un type booleen */
enum bool {Faux = 0, Vrai = 1};  | typedef enum  {Faux = 0, Vrai = 1} bool;
enum bool debut_de_ligne = Vrai; | bool debut_de_ligne = Vrai;

enum boisson {Cafe_court_sucre = 43, Cafe_court_non_sucre = 44,
              Cafe_long_sucre = 45,  Cafe_long_non_sucre = 46,
              Chocolat = 52, The_citron =55};

enum mois {Janvier = 1, Fevrier, Mars, Avril, Mai, Juin,
           Juillet, Aout, Septembre, Octobre, Novembre, Decembre};
                 /* Decembre "vaut" bien 12 */

enum etat_imprimante = {Au_repos, Active, En_panne, Hors_secteur};

typedef enum {Perdu, Dans_desordre, Dans_ordre} resultat_pari;
```

Types flottants

Trois types : `float, double, long double`

Codage IEEE-754 (voir `/usr/include/ieee754.h`) sous la forme «Signe Exposant Mantisse» : $S\ E_e...E_1E_0\ M_1M_2...M_m$

$$(-1)^S.2^E.(1+M)$$

Les constantes flottantes doivent contenir soit une partie décimale soit un exposant – exemples :

```
1E12 -5.6451E-6  1.0
```

```
define M_PIl 3.1415926535897932384626433832795029L  /* pi */
```

Attention : `double d = 1 / 3; /* d est initialisé à 0 */`

Les opérations sur les flottants peuvent avoir des résultats surprenant:

on peut avoir `(x+1)+1 != x + (1+1)` ou encore `x == x + 1` et même `x != x`.

Bibliothèque math.h

Contient la définition des fonctions mathématiques standards (cos, acos, cosh, sqrt, pow...). Toutes les fonctions sont basées sur le type `double`, par exemple:

```
double sqrt(double x);
```

```c
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
int
main()
{
  printf ("rac(%f) = %f\n", 2.0, sqrt (2.0));
  printf ("rac(%f) = %f\n", -2.0, sqrt (-2.0));
  return EXIT_SUCCESS ;
}
```

```
$ gcc -Wall sqrt.c
undefined reference to ‘sqrt’ -- collect2: ld returned 1 exit status
$ gcc -Wall -lm sqrt.c
$ ./a.out
rac(2.000000) = 1.414214
rac(-2.000000) = nan
```

Écrire une fonction qui retourne Vrai ssi x est premier:

```
int
premier (unsigned long x)
{
  unsigned long j;
  unsigned long racine_carre_x = (unsigned long)sqrt((double) x);

  if (x <= 3)
    return x == 2 || x == 3;
  if (x % 2 == 0)
    return 0;

  for (j = 3; x % j != 0 && j < racine_carre_x; j += 2)
    ;
  return (x % j != 0);
}
```

Codage

Le codage ASCII (American Standard Code for Information Interchange) code les caractères sur 7 bits, seuls sont codés les caractères anglo-saxons ('$').

Il existe 10 extensions ISO_8859 du code ASCII à 8 bits (man iso_8859_1) dont ISO-LATIN-1 qui comprend, en particulier, «notre» alphabet.

Constantes

D'après K&R, une constante caractère est **un entier** écrit sous la forme soit d'un caractère entre apostrophes ('0'), soit sous la forme d'une séquence d'*échappement* entre apostrophes :

`\ooo` : ooo est l'écriture octale de l'entier codant le caractère

`\xhh` : hh est l'écriture hexadécimale de l'entier codant le caractère

NUL	`\0`	caractère nul	BS	`\b`	retour arrière (BackSpace)
'	`\'`	apostrophe	FF	`\f`	fin de page (form feed)
\	`\\`	contre-oblique (backslash)	CR	`\r`	retour chariot (carriage return)
"	`\"`	guillemet anglais	HT	`\t`	tabulation horizontale
LF	`\n`	fin de ligne (line feed)	BEL	`\a`	sonnerie (Bell)

Afficher la liste des caractères

```
7  void
8  affiche_char ()
9  {
10   char c = 0;
11
12   do
13       printf ("%c", c);
14   while (c++ != 255);
15 }
```

```
$ gcc -c char.c
 char.c: In function ‘affiche_char’:
 char.c:14: warning: comparison is always 1 due to limited range of data type
```

```
#include <limits.h>

void
affiche_bien_char ()
{
  char c = CHAR_MIN;

  do
      printf ("%d[%c]",c,c);
  while (c++ != CHAR_MAX);
}
```

```
-128[]-127[]-126[]-125[]-124[]-123[]-122[]-121[]-120]-119[      ]-118[
                                                              ]-117]
-114[]-113[]-112[]-111[]-110[]-109[]-108[]-107[]-106[]-105[]-104[]-103[]-102[]
       /* 8 lignes ont été  effacées */
-36[Ü]-35[Ý]-34[þ]-33[ß]-32[à]-31[á]-30[â]-29[ã]-28[ä]-27[å]-26[æ]-25[ç]-24[è]
-23[é]-22[ê]-21[ë]-20[ì]-19[í]-18[î]-17[ï]-16[∂]-15[ñ]-14[ò]-13[ó]-12[ô]-11[õ]
-10[ö]-9[÷]-8[ø]-7[ù]-6[ú]-5[û]-4[ü]-3[ý]-2[þ]-1[ÿ]0[]1[]2[]3[]4[]5[]6[]7[]8]
9[      ]10[
]14[]15[]16[]17[]18[]19[]20[]21[]22[]23[]24[]25[]26[]27[
```

Jeu de caractères


```
#include <ctype.h>
```

déclarations de fonctions (macros) du type `int isxxxxx(int  caractere);`

```
isalpha isalnum isalpha iscntrl isdigit islower
isgraph isprint ispunct isspace isupper isxdigit
```

```
void
affiche_unsigned_char ()
{
  unsigned char c = 0;

  do
    if (isprint (c))
      printf ("%c", c);
  while (c++ != UCHAR_MAX);
}
```

```
 !"#$%&'()*+,-./0123456789:;<=>?@
ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`
abcdefghijklmnopqrstuvwxyz{|}~
```

```
#include <locale.h>
```

```
char *setlocale(int category, const char * locale);
```

Permet de définir certaines caractéristiques locales (équivalence de caractères, symbole monétaire, numérique...)

```
int
main ()
{
  affiche_bien_char () ;
  affiche_unsigned_char ();
  setlocale (LC_CTYPE, ""); /* utilise la variable LANG */
  printf ("\napres␣setlocale\n");
  affiche_unsigned_char ();
  return EXIT_SUCCESS;
}
```

```
apres setlocale
 !"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`
abcdefghijklmnopqrstuvwxyz{|}~¡?£??|§¨©?<<?????±²³´?¶·¸¹?>>¼½¾¿
ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖ×ØÙÚÛÜÝÞßàáâãäåæçèéêëìíîï∂ñòóôõö÷øùúûüýþÿ
```

Majuscule

Écrire une fonction qui transforme une lettre minuscule en lettre majuscule.

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

unsigned char
en_majuscule (unsigned char c)
{
 if (c >= 'a' && c <= 'z')
    return 'A' -  'a' + c ;
 return c;
}

int
main()
{
  unsigned char k ;
  for(k = '0'; k <= 'z'; k++)
    printf("%c",en_majuscule(k)) ;
  return EXIT_SUCCESS ;
}
```

```
0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

```
void
affiche_upper_unsigned_char ()
{
  unsigned char c = 0;

  do
    if (isprint (c) && toupper(c) != c)
      printf ("%c[%c]", c, toupper(c));
  while (c++ != UCHAR_MAX);
}
```

```
a[A]b[B]c[C]d[D]e[E]f[F]g[G]h[H]i[I]j[J]k[K]l[L]m[M]
n[N]o[O]p[P]q[Q]r[R]s[S]t[T]u[U]v[V]w[W]x[X]y[Y]z[Z]
apres setlocale
a[A]b[B]c[C]d[D]e[E]f[F]g[G]h[H]i[I]j[J]k[K]l[L]m[M]
n[N]o[O]p[P]q[Q]r[R]s[S]t[T]u[U]v[V]w[W]x[X]y[Y]z[Z]
à[À]á[Á]â[Â]ã[Ã]ä[Ä]å[Å]æ[Æ]ç[Ç]è[È]é[É]ê[Ê]ë[Ë]ì[Ì]
í[Í]î[Î]ï[Ï]∂[Ð]ñ[Ñ]ò[Ò]ó[Ó]ô[Ô]õ[Õ]ö[Ö]ø[Ø]ù[Ù]ú[Ú]
û[Û]ü[Ü]ý[Ý]þ[þ]
```

Un vecteur est une suite d'objets de même type :

```
#define DIMENSION 3
double vecteur_vitesse[DIMENSION] ;
```

désigne une collection de `DIMENSION` variables de type double:

```
vecteur_vitesse[0]              !!! ça commence forcément à 0 !!!
vecteur_vitesse[1]
...
vecteur_vitesse[DIMENSION-1]
```

Il n'existe pas d'opération manipulant des vecteurs en tant que tels :

```
double vitesse_train[DIMENSION];
double vitesse_relative_personne[DIMENSION];
double vitesse_absolue_personne[DIMENSION];
...
vitesse_absolue_personne = vitesse_train + vitesse_relative_personne ;
```

Exemples

```
#define NOM_MAX 35
char mon_nom[NOM_MAX], nom_president[NOM_MAX] ;

#define DIMENSION 3
double position_robot[DIMENSION], vitesse_robot[DIMENSION] ;

#define LONGUEUR_TIRAGE 6
short tirage_loto[LONGUEUR_TIRAGE], ma_grille[LONGUEUR_TIRAGE] ;

short premiers[] = {2,3,5,7,11,13,17} ;
char chiffre[] = "0123456789" ;
char chaine[] = {'c','\'','e','s','t',' ','l','o','n','g','\0'} ;
```

Utilisation des vecteurs

Connaissant l'arrivée d'un tiercé, calculer si un «ticket» de tiercé est gagnant dans l'ordre ou dans le désordre.

Analyse : une arrivée et un ticket de tiercé sont des triplets de numéros distincts deux à deux. On dit qu'un ticket est gagnant dans l'ordre si le triplet du ticket est égal au triplet de l'arrivée; il est gagnant dans le désordre si chacun de numéros apparait dans le triplet de l'arrivée.

```
#include <stdio.h>                          /* PARIS premiere version */
#include <stdlib.h>
#define TAILLE_COMBINAISON 3

typedef short combinaison[TAILLE_COMBINAISON] ;
typedef enum {Perdu, Dans_desordre, Dans_ordre} resultat_pari ;

resultat_pari evaluer_pari(combinaison mon_pari,
                           combinaison resultat_course);
int
main()
```

```
                int
main()
{
  combinaison mon_ticket = {1,2,3} ;
  combinaison arrivee = {3,2,1} ;

  switch(evaluer_pari(mon_ticket,arrivee))
    {
    case Dans_ordre:
      printf("Dans ordre") ;
      break ;
    case Dans_desordre:
      printf("Dans desordre") ;
      break ;
    case Perdu:
      printf("Perdu !") ;
    }
  return EXIT_SUCCESS ;
}
```

```
resultat_pari
evaluer_pari(combinaison mon_pari, combinaison resultat_course)
{
  short i ;

  for (i = 0; i < TAILLE_COMBINAISON
         && mon_pari[i] == resultat_course[i]; i++)
    ;
  if (i == TAILLE_COMBINAISON)
    return Dans_ordre ;
  for (i = 0; i < TAILLE_COMBINAISON
         && apparait(mon_pari[i],resultat_course) ; i++)
    ;
  if (i == TAILLE_COMBINAISON)
    return Dans_desordre ;
  return Perdu ;
}
```

```
int
apparait(short numero, combinaison combi)
{
  short i ;

  for (i = 0; i < TAILLE_COMBINAISON
           && numero != combi[i]; i++)
     ;
  return i < TAILLE_COMBINAISON ;
}
```

Vecteur de vecteurs

```
/* Declaration d'un vecteur de NOMBRE_MAX_MOTS mots
   ayant LONGUEUR_MAX_MOT caractères (y compris '\0') */
#define NOMBRE_MAX_MOTS 10
#define LONGUEUR_MAX_MOT 28

unsigned char mes_couleurs[NOMBRE_MAX_MOTS][LONGUEUR_MAX_MOT] =
      {"violet", "bleu", "vert", "jaune", "orange", "rouge"} ;

/* mes_couleurs[0] designe la chaine "violet" */
/* mes_couleurs[0][1] a pour valeur i */
/* mes_couleurs[0,1] designe la chaine "bleu" */


/* Declaration d'un ecran 512 X 768 en 256 couleurs */
typedef unsigned char pixel ;
pixel point[512][768] ;
```

Vecteur de vecteurs

```
/* Declaration d'une matrice X_DIM par Y_DIM par Z_DIM */
#define X_DIM 256
#define Y_DIM 256
#define Z_DIM 64

short scene_3d[X_DIM][Y_DIM][Z_DIM] ; /* 8Mo ! */

/* Definition d'un vecteur de paris */
#define NOMBRE_MAX_PARIS 5000
#define LONGUEUR_MAX_COMBINAISON 3
typedef short combinaison[LONGUEUR_MAX_COMBINAISON];

combinaison les_paris[NOMBRE_MAX_PARIS];

/* equivalent à
 short les_paris[NOMBRE_MAX_PARIS][LONGUEUR_MAX_COMBINAISON] ;
*/
```

```
entrez vos paris et terminez par le paris 0 0 0 :
paris numero 1 : 1 2 3
paris numero 2 : 3 2 1
paris numero 3 : 4 5 6
Sur 3 paris, il y a 1 gagnants dans l'ordre et 1 dans le desordre
```

```
#include <stdio.h>                          /* PARIS premiere version */
#include <stdlib.h>
#define TAILLE_COMBINAISON 3
#define NOMBRE_MAX_PARIS 3

typedef unsigned short combinaison[TAILLE_COMBINAISON] ;

typedef combinaison liste_paris[NOMBRE_MAX_PARIS] ;

typedef enum {Perdu, Dans_desordre, Dans_ordre} resultat_pari ;

resultat_pari evaluer_pari(combinaison mon_pari,
                           combinaison resultat_course);

/* lire_paris retourne le nombre de paris lus */
int lire_paris (liste_paris les_paris) ;
```

```c
                int
main()
{
  liste_paris les_tickets ;
  int nombre_paris, i;
  int nb_paris_dans_ordre = 0, nb_paris_dans_desordre = 0;
  combinaison arrivee = {3,2,1};

  nombre_paris = lire_paris(les_tickets);
  for (i = 0; i < nombre_paris; i++)
    switch(evaluer_pari(les_tickets[i],arrivee))
      {
      case Dans_ordre:
        nb_paris_dans_ordre++;
        break ;
      case Dans_desordre:
        nb_paris_dans_desordre++;
      case Perdu:
      }
  printf("Sur %d paris, il y a %d gagnants dans l'ordre "
         "et %d dans le desordre", nombre_paris,
         nb_paris_dans_ordre, nb_paris_dans_desordre);
  return EXIT_SUCCESS ;
}
```

```
                int
lire_paris (liste_paris les_paris)
{
  int paris_courant = 0;

  printf("entrez vos paris et terminez par le paris 0 0 0 :\n") ;
  do {
    printf("paris numero %d : ", paris_courant + 1) ;
    scanf(" %hu %hu %hu",
          &les_paris[paris_courant][0],
          &les_paris[paris_courant][1],
          &les_paris[paris_courant][2]);
  } while (les_paris[paris_courant][0] +
           les_paris[paris_courant][1] +
           les_paris[paris_courant][2] != 0
           && ++paris_courant < NOMBRE_MAX_PARIS);
  return paris_courant ;
}
```

Multiplication de matrices

Soient B et C deux matrices de dimensions respectives $i \times j$ et $j \times k$

$$A_{xy} = \sum_{p=1}^{p=j} (B_{xp}.C_{py})$$

```c
#include <stdio.h>
#include <stdlib.h>
#define PREMIERE_DIMENSION 400
#define SECONDE_DIMENSION 400

typedef double matrice[PREMIERE_DIMENSION][SECONDE_DIMENSION] ;

void matrice_multiplier(matrice resultat, matrice b, matrice c,
                        int b_premiere_dim, int dim_commune,
                        int c_seconde_dim);
int
main(){
matrice a, b, c ;
/* lecture des matrices b et c */
 matrice_multiplier(a,b,c,400,400,400) ;
 printf("%f",a[10][10]);
 return EXIT_SUCCESS;
}
```

Multiplication de matrices

```
void matrice_multiplier(matrice resultat, matrice b, matrice c,
                        int b_premiere_dim, int dim_commune,
                        int c_seconde_dim)
{
  int x;
  for(x = 0; x < b_premiere_dim; x++)
    {
      int y;
      for(y = 0; y < c_seconde_dim; y++)
        {
          int p ;
          resultat[x][y] = 0 ;
          for(p=0; p < dim_commune; p++)
            resultat[x][y] += b[x][p] * c[p][y] ;
        }
    }
}
```

Multiplication de matrices

```
void matrice_multiplier(matrice resultat, matrice b, matrice c,
                        int b_premiere_dim, int dim_commune,
                        int c_seconde_dim)
{
  int x;
  memset(resultat,sizeof(matrice),0) ;
  for(x = 0; x < b_premiere_dim; x++)
    {
      int p;
      for(p = 0; p < dim_commune; p++)
        {
          int y ;
          for(y=0; y < c_seconde_dim; y++)
            resultat[x][y] += b[x][p] * c[p][y] ;
        }
    }
}
```

```
> time ./premier_produit
0.00000011.41user 0.01system 0:11.42elapsed 99%CPU (blabla...)
> time ./second_produit
0.0000004.12user 0.00system 0:04.16elapsed 98%CPU (blabla...)
```

Type pointeur

Un pointeur peut être vu sous la forme d'un couple $\langle$ *adresse, type* $\rangle$ où *adresse* désigne un emplacement de la mémoire contenant un objet de type *type*.

Exemples de pointeurs :

```
int *i ; /* i est un pointeur d'entier, *i est un entier */

char *c ; /* c est un pointeur de caractère, *c est un caractère */

double *vitesse ;

int **pi ;  /* i est un pointeur de pointeur d'entier,
              *pi est un pointeur d'entier */
```

Affectation de pointeurs :

```
int page = 100, k;
int *pointeur = NULL ; /* désigne l'adresse 0 */
pointeur = &page; /* pointeur contient l'adresse de page */
```

Type pointeur

Opération fondamentale : l'**indirection**.

```
k = *pointeur ;   /* equivalent à k = page i.e. 100 */
page = 10 ;       /* *pointeur vaut aussi 10 et k 100  */
*pointeur = 1000 ;/* page vaut aussi 1000 et k 100 */
pointeur = &k ;   /* *pointeur vaut 100 et page vaut 1000 */
```

Difficultés:

lors de la déclaration on iniatilise seulement le pointeur, jamais l'objet pointé.

```
int *pointeur = 100; /* pointe vers l'adresse 100, pourquoi ? */
char *car = 'a'; /* car pointe vers l'adresse 97 le code ascii de a*/
```

se référer à un objet inconnu

```
char car, *c ;
*c = 'e';              /* c pointe on ne sait où, dans les deux cas
car = *c;              il faut espérer une "segmentation fault" */
```

```
char t[] = "abcdef";  /* t est un pointeur constant,
                         t[0] est un objet modifiable */
char *s = "123456";   /* s est un objet modifiable
                         "123456" est une chaine constante */

t[0] = '1';  /* t contient "1bcdef" */

s[0] = 'a';  /* impossible : provoque une erreur de segment */
t = s;       /* impossible t est une constante */

s = t;       /* s et t désignent "abcdef" */
s[0] = '2';  /* s et t désignent "2bcdef" */
s = "azer";  /* t désigne toujours "2bcdef" */
s[0] = '1';  /* impossible "azer" est une constante */

s = &t[2];   /* s désigne "cdef" */
s = t + 4;   /* équivalent à s =  &t[4];  s désigne "ef" */
s[0] = 'i';  /* t désigne "2bcdif" et s désigne "if" */
```

```
void
incrementer(int *i)
{
  assert(i != NULL);/* equivalent a ++*i ou a (*i)++ */
  *i = *i + 1;      /* mais pas a *++i,  ni a *i++ */
}

void
supprimer_espace(char *c){
  char  *ecriture = c;

  assert(c != NULL);
  while(*c != '\0')
    {
      if(*c != ' ')
        {
          *ecriture = *c;  /*   equivalent a  */
          ecriture++;      /*  *ecriture++ = *c */
        }
      c++;
    }
  *ecriture = '\0' ;
```

```
int
main(){
  int x = 0;
  char une_chaine[] = "e␣s␣p␣a␣c␣e";
  char *pc = "␣aie␣!␣aie␣!␣aie!";
  incrementer(&x) ;
  printf("%d\n",x) ;
  supprimer_espace(une_chaine);
  printf("%s\n",une_chaine);
  supprimer_espace(pc) ;
  return EXIT_SUCCESS;
}
```

```
> gcc -Wall passage.c
> ./a.out
1
espace
Segmentation fault (core dumped)
```

carré de matrices

```
void matrice_multiplier(matrice resultat, matrice b, matrice c,
                        int b_premiere_dim, int dim_commune,
                        int c_seconde_dim)
{
  int x;
  memset(resultat,sizeof(matrice),0) ;
  for(x = 0; x < b_premiere_dim; x++)
```

```
 /* m est l'identite */
  matrice_multiplier(m,m,m,10,10,10);
 /* m est nulle */
```

```
void matrice_multiplier(matrice resultat, matrice b, matrice c,
                        int b_premiere_dim, int dim_commune,
                        int c_seconde_dim)
{
  int x;

  assert(resultat != b &&  resultat != c);
  memset(resultat,sizeof(matrice),0) ;
```

Copier une zone de mémoire

```
/* linux/lib/string.c  Copyright (C) 1991, 1992  Linus Torvalds */

#include <stdlib.h>
void *
memmove(void *dest, void *src, size_t count)
{
  char *tmp, *s;

  if (dest <= src)
    {
      tmp = (char *) dest;
      s = (char *) src;
      while (count--)
        *tmp++ = *s++;
    }else{
      tmp = (char *) dest + count;
      s = (char *) src + count;
      while (count--)
        *--tmp = *--s;
    }

  return dest;
}
```

Une **structure** est une construction juxtaposant (regroupant séquentiellement) des objets de type différents.

`struct` ⟨nom_de_structure⟩ `{` liste_de_déclarations `}` ⟨ liste_d'identificateurs ⟩ ;

```
#define MAX_NOM 30
#define MAX_PRENOM 35

struct personne
{
  char nom[MAX_NOM];
  char prenom[MAX_PRENOM];
  short taille_en_cm;
  short poids_en_kilo;
  float temps_au_100m;
} barbie, ken;          /* a eviter */

struct personne karl;

typedef struct personne individu;

individu martin;
```

```
typedef struct
{
 unsigned char rouge;
 unsigned char vert;
 unsigned char bleu;
} couleur_rvb;

couleur_rvb ecran[RESOLUTION_VERTICALE][RESOLUTION_HORIZONTALE];

typedef struct
{
  double reel;
  double imaginaire;
} complexe;

struct impossible
{
  int valeur;
  struct impossible suite; /* erreur */
};
```

```
complexe
produit(complexe a, complexe b)
{
  complexe c;

  c.reel = a.reel * b.reel - a.imaginaire * b.imaginaire ;
  c.imaginaire = a.imaginaire * b.reel + b.imaginaire * a.reel;
  return c;
}

int
main()
{
  complexe x = {1.0,0};
  complexe y = {0,1.0};
  complexe z ;
  z = produit(x,y);
  return EXIT_SUCCESS;
}
```

On ne peut pas comparer directement deux structures.

```
typedef struct
{
  char marque[LONGUEUR_MARQUE];
  char modele[LONGUEUR_MODELE];
  short puissance_fiscale;
  short nombre_de_portes;
} automobile;

typedef struct
{
  char marque[LONGUEUR_MARQUE];
  char modele[LONGUEUR_MODELE];
  float poids_en_kg;
  int cylindree_en_cc;
  float puissance_en_cv;
} motocyclette;

typedef struct
{
  char marque[LONGUEUR_MARQUE];
  char modele[LONGUEUR_MODELE];
  float poids_en_kg;
  int diametre_roue_en_mm;
} velocipede;
```

```c
void
decrire_automobile(automobile a)
{
  printf("Modele : %s Marque : %s, %dCV, %d Portes",
          a.modele, a.marque, a.puissance_fiscale,
          a.nombre_de_portes);
}

void
decrire_motocyclette(motocyclette m)
{
  printf("Modele : %s Marque : %s, %fkg %dcc %fcv",
          m.modele, m.marque, m.poids_en_kg,
          m.cylindree_en_cc,  m.puissance_en_cv);
}

void
decrire_velocipede(velocipede v)
{
  printf("Modele : %s Marque : %s, %fkg %dmm",
          v.modele, v.marque, v.poids_en_kg,
          v.diametre_roue_en_mm);
}
```

union

Une **union** est une construction *superposant* des objets de type différents.

```
typedef enum {Auto, Moto, Velo} type_vehicule ;
```

```
typedef struct
{
  char nom_proprietaire[LONGUEUR_NOM] ;
  type_vehicule sorte_vehicule;
  union
  {
    automobile voiture;
    motocyclette moto;
    velocipede velo;
  } information_vehicule ;
} vehicule ;
```

```
void
decrire_vehicule(vehicule v)
{
  printf("\nProprietaire : %s\nType de vehicule : ", v.nom_proprietaire);
  switch (v.sorte_vehicule)
    {
    case Auto:
      printf("Automobile\n");
      decrire_automobile(v.information_vehicule.voiture);
      break;
    case Moto:
      printf("Motocyclette\n");
      decrire_motocyclette(v.information_vehicule.moto);
      break;
    case Velo:
      printf("Velocipede\n");
      decrire_velocipede(v.information_vehicule.velo);
    }
}
```

```
int
main ()
{
  velocipede mon_velo = {"Gauloise","vite++", 12.3,700} ;
  vehicule un_vehicule = {"martin",Auto,{{"Peunault","X24",12,4}}} ;

  decrire_vehicule (un_vehicule);

  strcpy (un_vehicule.nom_proprietaire, "Martin");
  strcpy (un_vehicule.information_vehicule.velo.marque, "veloplus");
  strcpy (un_vehicule.information_vehicule.velo.modele, "rando++");
  un_vehicule.information_vehicule.velo.poids_en_kg = 14.2;
  un_vehicule.information_vehicule.velo.diametre_roue_en_mm = 700;
  un_vehicule.sorte_vehicule = Velo;

  decrire_vehicule (un_vehicule);
  un_vehicule.sorte_vehicule = Auto;
  decrire_vehicule (un_vehicule);
  un_vehicule.information_vehicule.velo = mon_velo;
  un_vehicule.sorte_vehicule = Velo;
  decrire_vehicule (un_vehicule);

  return EXIT_SUCCESS;
}
```

```
Proprietaire : martin
Type de vehicule : Automobile
Modele : X24 Marque : Peunault, 12CV, 4 Portes

Proprietaire : Martin
Type de vehicule : Velocipede
Modele : rando++ Marque : veloplus, 14.200000kg 700mm

Proprietaire : Martin
Type de vehicule : Automobile
Modele : rando++ Marque : veloplus, 12CV, 13107 Portes

Proprietaire : Martin
Type de vehicule : Velocipede
Modele : vite++ Marque : Gauloise, 12.300000kg 700mm
```

Permet découper une zone mémoire bit par bit. Exemple tiré de "ieee754.h" (simplifié au cas "BIG ENDIAN") :

```
union ieee754_double
{
  double d;
    /* This is the IEEE 754 double-precision format.  */
  struct
  {
    unsigned int negative:1;
    unsigned int exponent:11;
    unsigned int mantissa0:20;
    unsigned int mantissa1:32;
    } ieee;
  struct
  {
    unsigned int negative:1;
    unsigned int exponent:11;
    unsigned int quiet_nan:1;
    unsigned int mantissa0:19;
    unsigned int mantissa1:32;
    } ieee_nan;
  };
```

```



/* D'apres libc/sysdeps/ieee754/__isinf.c */

/* Return 0 if VALUE is finite or NaN, +1 if it
   is +Infinity, -1 if it is -Infinity. */
int
isinf(double value)
{
  union ieee754_double *u = (union ieee754_double *) &value;

  /* An IEEE 754 infinity has an exponent with the
     maximum possible value and a zero mantissa. */
  if ((u->ieee.exponent & 0x7ff) == 0x7ff &&
      u->ieee.mantissa0 == 0 && u->ieee.mantissa1 == 0)
    return u->ieee.negative ? -1 : 1;

  return 0;
}
```

Liste par priorité décroissante :

<table>
<tr><th>Opérateur</th><th>Asso.</th><th>Arité</th></tr>
<tr><td>() []
-> .</td><td>G</td><td>-
2</td></tr>
<tr><td>! ~ ++ -- + -
(<type>) * & sizeof</td><td>D</td><td>1</td></tr>
<tr><td>* / %</td><td>G</td><td>2</td></tr>
<tr><td>+ -</td><td>G</td><td>2</td></tr>
<tr><td><< >></td><td>G</td><td>2</td></tr>
<tr><td>< <= > >=</td><td>G</td><td>2</td></tr>
<tr><td>== !=</td><td>G</td><td>2</td></tr>
</table>

<table>
<tr><th>Opérateur</th><th>Asso.</th><th>Arité</th></tr>
<tr><td>&</td><td>G</td><td>2</td></tr>
<tr><td>^</td><td>G</td><td>2</td></tr>
<tr><td>|</td><td>G</td><td>2</td></tr>
<tr><td>&&</td><td>G</td><td>2</td></tr>
<tr><td>||</td><td>G</td><td>2</td></tr>
<tr><td>?:</td><td>D</td><td>3</td></tr>
<tr><td>= += -= *= /= %=
>>= <<= &= ^= |=</td><td>D</td><td>2</td></tr>
<tr><td>,</td><td>G</td><td>2</td></tr>
</table>

Toute expression entière est une expression logique (et réciproquement) :

x est vrai $\Leftrightarrow x \neq 0$

Opérateurs logiques ($Z \mapsto \{0,1\}$)

- comparaison `< <= > >=`
- égalité `== !=`
- «et alors» `&&`
- «ou sinon» `||`
- négation `!`

```
printf("%d",2 && 15) ; /* affiche 1 */
while(*x++ = *y++) ;
```

Expressions binaires «bit à bit»

`&` et `0011 & 0110 = 0010`

`|` ou `0011 | 0110 = 0111`

`^` ou-exclusif `0011 ^ 0110 = 0101`

`~` complément à 1 `~(0110) = 1001`

`<<` décalage à gauche (multiplication par 2) :

$(x_n x_{n-1} ... x_1 x_0) << 1 = x_{n-1} ... x_1 x_0 0$

`>>` décalage à droite

cas non signé : $(x_n x_{n-1} ... x_1 x_0) >> 1 = 0 x_n x_{n-1} ... x_1$ (division par 2)

cas signé : résultat dépendant de la machine (et du compilateur).

```
#define EST_IMPAIR(x) ((x) & 1)
#define OCTET_MAJEUR (((x)>>8) & 0xff)
#define SET_BIT(x,position) ((x) | (1 << (position)))
#define UNSET_BIT(x,position) ((x) & ~(1 << (position)))
#define COMPLEMENTER_BIT(x,position)  ((x) ^ ~(1 << (position)))
```

Utilisation des opérateurs bit à bit

```
unsigned long
puissance(unsigned long x, unsigned long y)
{
 long r = 1;   /*invariant: x puissance y =  r*(x puissance y)*/
 while (y != 1)
   if (y & 1)
     {
       r *= x ;
       x ^= 1 ;
     }
   else
     {
       x *= x;
       y >>= 1;
     }
  return x * r ;
}
```

Composition des opérateurs de déclaration * et [] 69

`[]` est prioritaire par rapport à `*`.

```
char *tab[] ;
/* *(tab[]) :  tab est un vecteur de pointeurs de char */

char (*tab)[] ;
/* *tab est un vecteur, tab est un pointeur de vecteur */

char *tab[][] ;
/* tab est un vecteur de vecteurs de pointeurs */

char  (*tab[])[] ;
/* tab est un vecteur de pointeurs de vecteur de char */

char *(*tab)[][] ;
/* tab est un pointeur de vecteur de  vecteurs de pointeurs */
```

Conseil: utiliser des types intermédiaires.

```
char *chaine = "globale";

void
fonction (char *chaine)
{
  printf("\nfonction:\t %s ",chaine);
}
void
seconde_fonction ()
{
  printf("\n2nde fonction:\t %s ",chaine);
}
void
troisieme_fonction (char *chaine)
{
  printf("\n3eme fonction:\t %s ",chaine);
  seconde_fonction ();
  {
    char *chaine = "bloc 3eme" ;
    printf("\nbloc 3eme:\t %s ",chaine);
  }
```

```c
int
main ()
{
  char *chaine = "principale";
  printf ("\nmain:\t %s ",chaine);
  {
    char *chaine = "seconde principale";
    printf ("\nbloc main:\t %s ",chaine);
    fonction (chaine);
    seconde_fonction ();
    troisieme_fonction (chaine);
  }
  return EXIT_SUCCESS ;
}
```

```
main:    principale
bloc main:       seconde principale
fonction:        seconde principale
2nde fonction:   globale
3eme fonction:   seconde principale
2nde fonction:   globale
bloc 3eme:       bloc 3eme
```

Déclaration: définition vs référence

Chaque variable ou fonction doit être définie (créée) exactement une fois.
Exemples:

```
int a;
char *chaine = "012345679";


int
 un()
{
  return 1;
}
```

Une référence correspond à l'affirmation de l'existence d'une définition de l'objet à un niveau 0 (dans un .h inclus, par exemple) :

```
extern int a;
extern char  *chaine;
int un() ; /* ou aussi extern int un() */
```

Définition: publique vs privée

Une variable (fonction) est publique si elle peut etre utilisée dans un autre fichier, elle est (en principe) définie dans un ".h".

Il est souvent nécessaire d'interdire l'utilisation d'une variable (fonction) en dehors du module (variable privée, locale au fichier).

```
static int a;
static char *chaine = "012345679";


static int
un()
{
  return 1;
}
```

Il faut limiter la portée des variables ; l'utilisation de variables ou de fonctions publiques doit donc être restreinte.

Différentes zones de mémoire :

1. code : constantes (chaînes de caractères) ;
2. tas (*heap*) : variables *statiques*, variables globales et *allocation dynamique* ;
3. pile (*stack*) : variables locales *automatique*, paramètres, valeurs intermédiaires ;
4. registre (*register*) : variables, constantes, valeurs intermédiaires (optimisation du compilateur)

Durée de vie des objets :

Permanente emplacement réservé lors de la compilation (adresse fixe définie lors de l'édition des liens);

Temporaire emplacement réservé lors de l'exécution du programme dans la pile ou les registres (variable locale, paramètre) ou dans le tas (allocation dynamique).

variable locale statique


```
int
nb_appels_automatique ()
{
  int nb = 0;
  return ++nb;
}

int
nb_appels_statique ()
{
  static int nb = 0;
  return ++nb;
}

int
main()
{
  int i;
  for(i=0;i < NB_APPELS; i++)
    printf("appel␣automatique␣?n%d␣statique␣?n%d\n",
           nb_appels_automatique (), nb_appels_statique ());
  return EXIT_SUCCESS;
}
```

```
appel automatique n 1 statique n 1
appel automatique n 1 statique n 2
appel automatique n 1 statique n 3
appel automatique n 1 statique n 4
appel automatique n 1 statique n 5
appel automatique n 1 statique n 6
```

Il faut préférer les variables locales statiques aux variables globales privées puis celles-ci aux variables publiques.

Utilités des variables

- globales privée : mémoire partagée par plusieurs fonction (état du module, mémoire de stockage, paramètre du module);
- globale publique : très rarement utilisées puisqu'elle peuvent être modifiée sans restriction (ex. `errno` du module `errno.h`, paramètre de module (ex. currency_symbole).

```
unsigned long
factorielle (short x)
{
  short i ;
  unsigned long resultat = 1;

  assert(x <= FACTORIELLE_MAX);
  for(i = 2; i <= x ; i++)
    resultat = resultat * i;
  return resultat;
}

...  int a = 2;
S:   unsigned long f = factorielle(a+1); ...
```

❶ évaluer les paramètres

❷ empiler les paramètres (ou les placer dans des registres)

❸ appeler la fonction (empiler l'adresse de retour et faire un saut)

❹ empiler les variables locales non-statiques (ou les placer dans des registres)

Appel de fonction, modèle pile

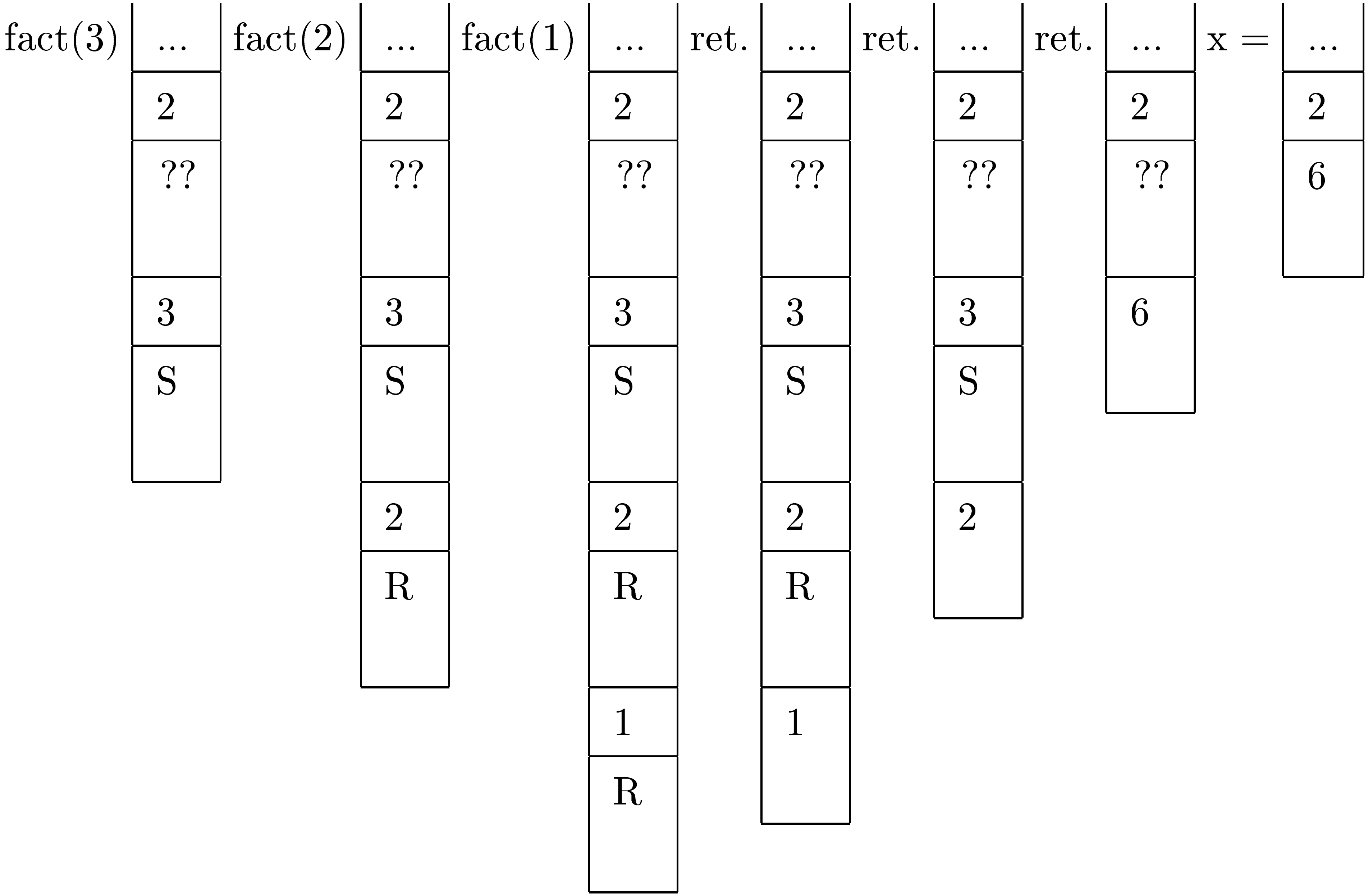
...
paramètre
adresse retour
variables locales

...	...	i=2	...	... ...	...	return 6	...	f=	...
int a	2		2		2		2		2
unsigned long f	??		??		??		??		6
short x	3		3		3		6		
adresse retour	S		S		S				
short i	??		2		4				
unsigned long resultat	1		1		6				

fonction récursive

```
unsigned long          /* le plus mauvais exemple classique !*/
factorielle (short x)
{
    assert(x <= FACTORIELLE_MAX);
    if (x < 2)
      return 1;
R:  return x * factorielle (x - 1);
}
      ...  int a = 2;
           unsigned long f= factorielle(a+1);
S: ...
```

fonction récursive



Variable automatique, registre et volatile

Une variable temporaire peut être stockée soit dans la pile, soit dans un registre. On peut «forcer» le compilateur a utiliser un mode de stockage plutot qu'un autre :

```
auto int i ; /* i doit etre socke dans la pile */

register int j ; /*j doit etre si possible stocke dans un registre */
                 /* &j n'est pas defini */
```

On utilisera très rarement ces constructions : un bon compilateur optimise généralement mieux qu'un bon informaticien ! pour optimiser un code il faut connaître intimement le code généré (l'assembleur) et l'architecture de l'ordinateur : pipeline du micro-processeur, organisation et performances des mémoires caches, répartition des données en mémoire...

Variables «volatile» : les compilateurs sont capables de simplifier un code et de supprimer des variables «jugées» inutiles, la construction volatile empêche la suppression de la variable. (Voir cours P.S.)

```
const int deux = 2 ; /* deux = 3 est interdit */

const int premier[] = {2,3,5,7,11} ; /* premier[0] = 1  est interdit */

int *const second = premier + 1;
             /* second = premier est interdit
                second[0] = 1 est autorisé */

const int *const troisieme = premier + 2 ;
             /* troisieme = premier est interdit
                troisieme[0] = 1 est autorisé */
```

Permet de supprimer des commentaires au niveau des déclarations de fonction.

```
void *realloc(const void *adresse, size_t nouvelle_taille) ;
```

Incapacité de définir une taille *raisonnable* pour un objet en mémoire : l'utilisation de tableaux déclarés à l'aide de constantes n'est pas toujours satisfaisant.

Exemple : lecture d'une chaine de caractères, stockage d'une séquence de scènes 3D définies à l'aide de polygones.

Fonctions d'allocation dynamique de la bibliothèque stdlib.h

```
void *malloc(size taille)

void free(void *adress)

void *realloc (void *adresse, size_t taille)

void *calloc (size_t nombre_elements, size_t taille_element)
```

Ces fonctions retournent le pointeur NULL en cas de problème.

Module Matrice

Il s'agit d'allouer un tableau de double aux "bonnes dimensions" et d'utiliser une fonction d'accés aux éléments.

On applique désormais le masquage d'implémentation.

Voir les sources sur la page web.

Module pile d'entiers

Il s'agit de comparer l'approche "cellulaire" à l'approche "tableau redimensionable".

Voir les sources sur la page web.

Module liste bilatère

Il s'agit de déterminer un type autorisant la manipulation de liste avec curseur.

On manipule les pointeurs (insertion, suppression de cellule, duplication)

Module arbre binaire

Après avoir vu la notion de pointeur de fonction (voir la fonction qsort de la libc) on réalise module générique d'A.B.R.

Application au stockage de macros.

On évoque ensuite le pb de la libération de l'espace mémoire.

Sauts

Instructions continue, break et goto.

Fonction exit, atexit, onexit.

Evocation de setjmp et longjmp.