Langage C et bibliothèque scientifique GSL

Ludovic Grossard

Université de Limoges

Historique

1970 Ken Thompson invente le langage B pour le premier système UNIX

1978 Brian Kernighan et Dennis Ritchie publient « The C programming language ». Bible en la matière

1983 normalisation du C par le « American National Standards Institute ». Naissance du C ANSI fin 1988. Le langage C devient indépendant de la machine

Avantages

- portabilité (objectif initial UNIX multiplate-forme),
- exécutables de petite taille et rapides à l'exécution.

Bibliothèques

- le langage C possède très peu d'instructions,
- fait appel à de nombreuses bibliothèques externes,
- il existe de très nombreuses bibliothèques de calcul scientifique.

Objectifs du cours

- maîtriser le langage C,
- savoir utiliser une bibliothèque externe,
- savoir lire sa documentation,
- savoir utiliser un compilateur.

Premier programme en C

Listing 1 – Premier programme en C

```
1 #include <stdio.h>
2 main()
3 {
4 printf("bonjour tout le monde\n");
5 }
```

à l'exécution, on obtient :

```
Bonjour tout le monde
```

Compilation

Pour arriver à l'exécution du programme, il faut :

- saisir le **code source** avec un éditeur de texte (.c)
- compiler (résultat : binaire ou exécutable)
- démarrer le programme

Analyse du code source

- tout programme c **doit** inclure une fonction dont le nom est **main**,
- l'exécution commence par la fonction main,
- main n'est pas forcément au début du code source,
- les parenthèses sont importantes. Avec main(), on définit une **fonction**,
- les accolades servent à définir les limites du **corps** de la fonction main,
- la première ligne signifie qu'on va faire appel à la bibliothèque stdio (standard input/output). C'est une instruction préprocesseur (avant compilation) qui aura pour effet la copie du contenu du fichier sdtio.h au début de notre code source.

Analyse du code source

- printf : imprime quelque chose à l'écran,
- c'est une fonction de bibliothèque,
- "bonjour tout le monde" est une chaîne de caractères,
- c'est l'**argument** de la fonction printf,
- \n signifie retour à la ligne (newline).

Instructions et expressions

- la ligne printf... est une **instruction** (demande à la machine de faire quelque chose),
- les instructions se terminent par des points-virgules,
- l'utilisation simple d'une fonction est techniquement une expression,
- l'expression est **évaluée**, et possède une **valeur**.

Présentation

Le C n'est pas sensible à l'apparence du programme. On ajoute autant d'espaces, tabulations et retours à la ligne que l'on veut **entre** les instructions.

```
1 #include <stdio.h>
2 main(){
3 printf("bonjour␣tout␣le␣monde\n");}
```

Présentation

Il est **indispensable** de commenter ses programmes. Tout le texte compris entre /* et */ sera ignoré au moment de la compilation.

```
1  /* L. Grossard, 10/09/2004
2     ce programme affiche du texte à l'écran */
3
4  #include <stdio.h>
5  main()
6  {
7  /* affiche du texte à l'écran */
8  printf("bonjour␣tout␣le␣monde\n");
9  }
```

Définition

- variable = zone réservée en mémoire pour stocker une valeur,
- la valeur de la variable peut changer pendant l'exécution du programme,
- les variables portent un **nom** :
  - composé de chiffres et de lettres,
  - le 1er caractère doit être une lettre,
  - norme ANSI : au moins 31 caractères sont pris en compte.

Types de variables

Le C comporte peu de types de base.

signé ?	qualificatif	type	description	octets	plage
signed	short	int	nombres entiers	2	-32768 . . .32767
unsigned	short	int	nombres entiers	2	0 . . .65535
signed	-	int		4	-2 147 483 648 . . .2 147 483 647
unsigned	-	int		4	0 . . .4 294 967 295
signed	long	int		4	-2 147 483 648 . . .2 147 483 647
unsigned	long	int		4	0 . . .4 294 967 295
signed	-	char	un caractère	1	-127 . . .128
unsigned	-	char		1	0 . . .255
		float	nombre à virgule flottante, simple précision	4	maxi : 1.10^{37} mini : 1.10^{-37} c.s. : 6
		double	nombre à virgule flottante, double précision	8	maxi : 1.10^{37} mini : 1.10^{-37} c.s. : 15
	long	double		12	maxi : 1.10^{37} mini : 1.10^{-37} c.s. : 17

Représentation interne

À l'intérieur de la machine, tous les nombres sont représentés par une suite de 0 et de 1.

type	taille (octets)	taille (bits)	mini	maxi
char	1	8	0	255
unsigned int	4	32	0	$2^{32} - 1$
float	4	32	0	-
double	8	64		-

Représentation interne

Les flottants sont représentés comme suit :

bit de signe	exposant (8 bits)	mantisse (23 bits)

Les doubles sont représentés comme suit :

bit de signe	exposant (11 bits)	mantisse (52 bits)

Déclaration et initialisation d'une variable

- une déclaration précise un type puis une liste de une ou plusieurs variables de ce type

```
int mini, maxi, intervalle;
char c, ligne[1000]
```

- on peut également initialiser les variables au moment de la déclaration

```
int i = 0;
float eps = 1.0e-5;
```

= est l'opérateur d'affectation

- le qualificatif **const** indique que les valeurs ne seront pas modifiées

```
const double e = 2.718;
```

Opérateurs arithmétiques

- `+` addition,
- `-` soustraction,
- `*` produit,
- `/` quotient,
- `%` modulo (int et char uniquement).

Opérateurs de comparaison et opérateurs logiques

- opérateurs de comparaison : `>` `>=` `<` `<=`
- opérateurs d'égalité : `==` `!=`
- opérateurs logiques : || (ou) && (et)

```
exemple :

x = 1 ; y = 2;              (x>y) vaut 0 (faux)

(x>0) && (y==1) vaut 0 (faux)
```

Opérateurs d'incrémentation et de décrémentation

`++` ajoute 1 à son opérande

```
i++;      équivalant à i=i+1;
```

`--` retranche 1 à son opérande

```
i--;      équivalant à i=i-1;
```

`+=` ajoute le second opérande au premier

```
x+=y;     équivalant à x=x+y;
```

`-=` retranche le second opérande au premier

```
x-=y;     équivalant à x=x-y;
```

Conversions implicites

Définition

Une conversion implicite est une modification du type d'une expression par le programme, sans que le programmeur ne l'ait demandé.
Elle a lieu quand on fait intervenir dans la même expression des variables de types différents.

Quelques règles

Préservation de la valeur numérique
conversion d'un nombre signé ou non signé d'une représentation binaire plus petite vers une représentation binaire plus grande.
exemple : short int vers long int

Quelques règles

Préservation de la valeur numérique
conversion d'un nombre signé ou non signé d'une représentation binaire plus petite vers une représentation binaire plus grande.
exemple : short int vers long int
Troncage de la représentation binaire
conversion directe d'un type de départ de la représentation binaire supérieur à celle du type d'arrivée.
exemple : int vers char

Quelques règles

Préservation de la valeur numérique
conversion d'un nombre signé ou non signé d'une représentation binaire plus petite vers une représentation binaire plus grande.
exemple : short int vers long int
Troncage de la représentation binaire
conversion directe d'un type de départ de la représentation binaire supérieur à celle du type d'arrivée.
exemple : int vers char
Conservation de la représentation binaire
conversion d'un type signé vers un type non signé. Les valeurs numériques ne sont pas conservées.
exemple : unsigned int vers int

Quelques règles

Préservation de la valeur numérique
conversion d'un nombre signé ou non signé d'une représentation binaire plus petite vers une représentation binaire plus grande.
exemple : short int vers long int
Troncage de la représentation binaire
conversion directe d'un type de départ de la représentation binaire supérieur à celle du type d'arrivée.
exemple : int vers char
Conservation de la représentation binaire
conversion d'un type signé vers un type non signé. Les valeurs numériques ne sont pas conservées.
exemple : unsigned int vers int
Troncage à la virgule
conversion d'un type flottant vers un type entier.
exemple : float vers int

Quelques règles

Listing 2 – conversions implicites

```
1  #include <stdio.h>
2  main()
3  {
4     double x , y;
5     x = 1 + 2 /3   ;
6     y = 1 + 2 /3.0 ;
7  }
```

- la priorité de l'opérateur / est supérieure à celle de l'opérateur +, la division est faite en premier
- 2/3 : les deux opérandes sont entières donc la division réalisée est entière et le résultat est 0
- 2/3.0 : la division est flottante, le résultat est 0.6666... Il y a eu conversion implicite de 2 en 2.0

Quelques règles

C'est l'opérateur de conversion de type.

```
x = 1 + 2 / (double) 3
```

La valeur entière 3 est convertie (correctement) en la valeur flottante 3.0 et le résultat du calcul est correct.

Les entrées/sorties

Les fonctions d'entrées / sorties sont définies dans le fichier stdio.h. Il est nécessaire d'ajouter la ligne suivante au début du programme :

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

Quand le nom qui suit `include` est placé entre < et >, le fichier d'en-tête est recherché dans un certain nombre d'emplacements standards, par exemple `/usr/include` sous Linux.

Afficher du texte à l'écran

putchar(c) envoie le caractère c sur la sortie standard, par défaut l'écran. Exemple : `putchar('a');`

puts(chaine) écrit la chaîne de caractères et un caractère de fin de ligne sur la sortie standard. Exemple : `puts("bonjour tout le monde");`

Remarque : les caractères sont entourés par des guillemets simples, et les chaînes de caractères par des guillements doubles.

Récupérer du texte saisi au clavier

getchar() renvoie le caractère saisi au clavier. Exemple :

```
char c;
c = getchar();
```

Les parenthèses sont obligatoires car sinon le compilateur considèrerait getchar comme une variable.

gets(chaine) lit une ligne entière et la place dans la chaîne de caractères "chaine". Exemple :

```
char nom[10];
gets(nom);
```

Exemple

Listing 3 – conversion de minuscules des caractères saisis

```
1  #include <stdio.h>
2  #include <ctype.h>
3  main()
4  {
5     char c;
6     while( (c=getchar()) != EOF)
7       putchar(tolower(c));
8  }
```

- EOF vaut -1 et est défini dans stdio.h. Il signifie End Of File. On l'obtient sous Linux avec la combinaison de touches Ctrl+Z.
- tolower est définie dans ctype.h

La fonction printf

La fonction de sortie printf convertit des données internes en caractères.

Syntaxe

```
int printf("chaine de formatage" , arg1 , arg2 , ... )
```

- printf convertit, met en forme et imprime ses arguments sur la sortie standard
- sous le contrôle de la chaîne de formatage
- elle retourne le nombre de caractères imprimés.

La fonction printf

La chaîne de formatage est composée de :

- caractères ordinaires
- spécificateurs de conversion. Ils commencent par un % et se terminent par le caractère de conversion.

Spécificateur de conversion

%	-	n_1	.	n_2	c
début du spécificateur	aligner à gauche	largeur minimale du champ d'impression		précision	caractère de conversion

Principaux spécificateurs de conversion

Principaux spécificateurs de conversion :

Caractère	**type d'argument ou type d'impression**
d,i	int
c	char
s	string
f	float
lf	double
Lf	long double

Exemple

```
1  #include <stdio.h>
2  main()
3  {
4      float f = 1.234567;
5      printf("x = %6.3lf" , f);
6  }
```

affichera :

```
x =  1.235
```

Autre exemple

autre exemple :

```
1  #include <stdio.h>
2  main()
3  {
4      int i = 3;
5      float x = 1.2;
6      printf("t[%d] = %lf", i , x );
7  }
```

affichera :

```
t[3] = 1.2
```

Autre exemple

Attention, la fonction ne vérifie pas si le nombre d'arguments est correct. On peut utiliser les caractères non imprimables suivants :

\n	retour à la ligne
\t	tabulation
\a	signal sonore
\"	guillemet

La fonction sprintf

Elle effectue les mêmes conversions que `printf`, mais elle stocke la sortie dans une chaîne de caractères.

```
int sprintf(chaine_cible , "chaine de formatage" ,
            arg1 , arg2 , ...)
```

La fonction scanf

Syntaxe

```
int scanf("chaine de formatage" , ...)
```

- lit les caractères saisis,
- les interprète selon les spécifications incluses dans la chaine de formatage,
- stocke les résultats dans les arguments suivants,
- les caractères de conversion sont les mêmes que pour la fonction printf,
- les arguments suivants sont les **adresses** des variables dans lesquelles on veut placer le résultat,
- l'adresse d'une variable s'obtient en plaçant un **&** devant son nom,
- exception : un nom de tableau ou de chaîne de caractères est déjà une adresse de variable. Ne pas mettre de & dans ce cas.

Exemple

```
1  #include <stdio.h>
2  main()
3  {   /* calculateur rudimentaire */
4     double somme , v;
5     while( scanf("%lf" , &v ) == 1 )
6        printf("\t%.2f\n", somme +=v);
7     return 0;
8  }
```

Erreur courante

L'erreur la plus courante est d'écrire

```
scanf( "%d" , n );
```

au lieu de

```
scanf( "%d" , &n );
```

Erreur non détectée à la compilation.

Les structures de contrôle

Introduction

Les instructions de contrôle permettent au programme de prendre des décisions et précisent l'ordre dans lequel s'effectuent les traitements.

Les instructions et les blocs

- une expression devient une instruction lorsqu'elle est suivie d'un point virgule,
- les accolades { et } servent à regrouper des déclarations et des instructions pour obtenir une **instruction composée** ou **bloc**,
- ce bloc est syntaxiquement équivalent à une instruction unique,
- l'accolade fermante qui termine un bloc n'est pas suivie d'un point-virgule.

L'instruction if-else

Objectif

permet la prise de décision

Syntaxe

```
1 if (expression)
2    instruction 1
3 else
4    instruction 2
```

- l'expression est évaluée,
- si elle est vraie (si expression a une valeur non nulle), l'instruction 1 s'exécute,
- si elle est fausse (expression vaut zéro), l'instruction 2 s'exécute,
- la partie `else` est facultative,
- instruction1 et instruction2 peuvent être des blocs (rappel : les blocs ne se terminent pas par des points-virgule).

Exemple

```
1  ...
2  if(i%2 == 0)
3     printf("i␣est␣pair");
4  else
5  {
6     print("i␣est␣impair");
7     i++;
8  }
9  ...
```

Écriture raccourcie

`if` se borne à tester la valeur numérique de `expression`. On peut alors écrire :

```
if (expression)
```

au lieu de

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

exemple :

```
if(converge)
   printf("l'algorithme a convergé");
```

L'instruction else-if

Objectif

Permet de programmer un choix entre plusieurs possibilités

Syntaxe

```
1  if(expression)
2     instruction
3  else if(expression)
4     instruction
5  else if(expression)
6     instruction
7  else
8     instruction
```

- les expressions sont évaluées dans l'ordre où elles apparaissent,
- dès que l'une d'entres elles est vraie, on exécute l'instruction correspondante et la séquence s'arrête,
- le dernier else s'occupe du cas par défaut. Il peut être omis.

L'instruction switch

Objectif
permet la prise de décision à choix multiple. L'instruction regarde si la valeur d'une expression fait partie d'un certain nombre de constantes entières, et effectue des traitements associés à la valeur correspondante.

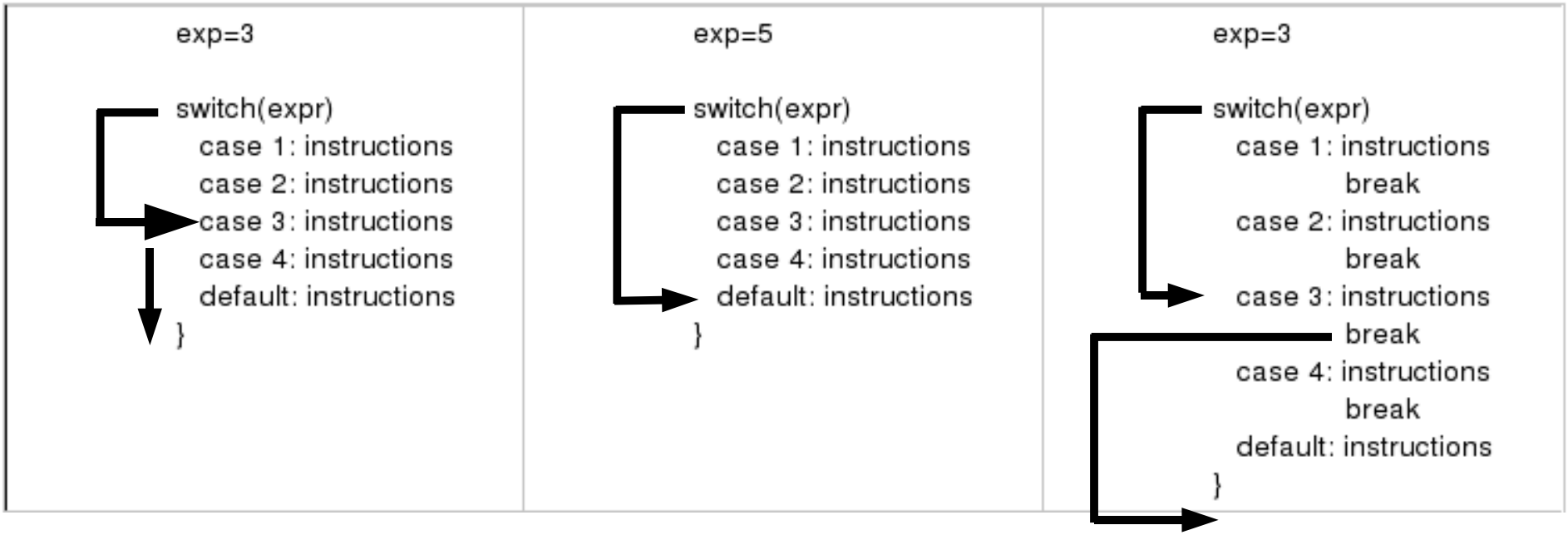
L'instruction switch

Syntaxe

```
1  switch(expression){
2     case expression-constante: instructions
3     case expression-constante: instructions
4     default: instructions
5  }
```

- chacun des cas possibles est étiqueté par une ou plusieurs constantes,
- l'exécution se produit sur le case dont la valeur associée correspond à la valeur de l'expression du « switch », L'exécution continue jusqu'à la fin du corps de l'instruction switch, ou bien jusqu'à la rencontre du mot clé « break »,
- si on n'est dans aucun des cas, l'exécution démarre au cas indiqué par défaut.

Exemples


exp=3
switch(expr)
case 1: instructions
case 2: instructions
case 3: instructions
case 4: instructions
default: instructions
}
exp=5
switch(expr)
case 1: instructions
case 2: instructions
case 3: instructions
case 4: instructions
default: instructions
}
exp=3
switch(expr)
case 1: instructions
break
case 2: instructions
break
case 3: instructions
break
case 4: instructions
break
default: instructions
}

Les boucles

Introduction

- les boucles (instructions répétitives) permettent de faire exécuter plusieurs fois certaines parties du programme,
- le langage C dispose de trois types de boucles :
  - `while`
  - `for`
  - `do...while`

L'instruction while

Objectif

permet de répéter l'exécution d'instructions tant que la valeur d'une expression est vraie

Syntaxe

```
1 while(expression)
2     instruction
```

`instruction` peut être une instruction unique ou un bloc

Exemple

```
1  #include <stdio.h>
2  main()
3  {
4      int z=3;
5      while(z>0)
6      {
7          printf("%d ",z);
8          z--;
9      }
10 }
```

Le programme produira à l'écran :

```
3 2 1
```

L'instruction for

Objectif

permet de répéter l'exécution d'instructions tant que la valeur d'une expression est vraie

Syntaxe

```
1 for( expr_I ; expr_C ; expr_R )
2     instructions
```

L'instruction peut être une instruction simple ou un bloc.

- `expr_I` expression initialisant les variables de contrôle avant d'entrer dans la boucle,
- `expr_C` condition de bouclage,
- `expr_R` expression permettant de réinitialiser les variables de contrôle utilisées.

Exemple

Le programme suivant affiche les nombres de 1 à 10

```
1  #include <stdio.h>
2  main()
3  {
4     int i;
5     for( i = 1 ; i < 11 ; i++ )
6        printf("%d ",i);
7  }
```

à l'exécution, on obtient :

```
0 1 2 3 4 5 6 7 8 9 10
```

Initialisations et incrémentations multiples

On les sépare par des virgules.

```
1  for( position = 0 , température = 301 ; (energie <
       500 ) || (temps==1000) ; temps++ )
```

les trois paramètres (initialisation, condition, incrémentation) peuvent être des variables différentes.

Équivalence entre les boucles for et while

```
#include <stdio.h>
main()
{
int i;
for(i=0;i<11;i++)
   printf("%d ",i);
}
```

```
#include <stdio.h>
main()
{
int i=0;
while(i<11)
{
   printf("%d ",i);
   i++;
}
}
```

Le choix d'un type de boucle se fait au cas par cas

L'instruction do . . . while

Objectif

- permet de réaliser l'exécution d'instructions tant que la valeur d'une expression est vraie
- le test se fait après l'exécution des instructions
- les instructions sont exécutées au moins une fois

L'instruction do . . . while

Objectif

- permet de réaliser l'exécution d'instructions tant que la valeur d'une expression est vraie
- le test se fait après l'exécution des instructions
- les instructions sont exécutées au moins une fois

Syntaxe

```
1 do
2    instruction
3 while(expression);
```

- `instruction` peut être une instruction unique ou un bloc d'instructions,
- notez le point-virgule à la fin de la boucle.

Exemple

```
1  #include <stdio.h>
2  main()
3  {
4      int i = 0;
5      do
6          printf("%d ",i);
7          i++;
8      while(i<11);
9  }
```

Les instructions `break` et `continue`

- permettent de sortir d'une boucle autrement qu'en testant une condition au début et à la fin,
- `break` permet de sortir directement de la boucle `for`, `while` ou `do...while`, tout comme pour `switch`,
- l'instruction `continue` relance immédiatement la boucle `for`, `while` ou `do...while` dans laquelle elle se trouve. La condition d'arrêt est immédiatement réévaluée

```
1  for( i=0 ; i<n ; i++)
2  {
3     if(t[i] < 0) /* sauter les éléments négatifs */
4        continue;
5     ... /* traiter les éléments positifs */
6  }
```

Environnement Linux

- Linux est un système d'exploitation de type UNIX,
- multiplate-forme,
- multiutilisateur,
- pourvu de beaucoup d'outils de développement, notamment en C,
- gratuit mais surtout libre.

Connexion au système

- pour utiliser l'ordinateur, il faut disposer d'un compte (nom d'utilisateur et mot de passe), assigné par l'administrateur du système,
- Voici une session fictive :

```
login: Michel  (tapez votre nom d'utilisateur)
password:      (tapez votre mot de passe, ne s'affiche p
$
```

- le $ signifie que la machine est prête à reçevoir des instructions,
- pour se déconnecter, taper logout et valider par entrée.

Syntaxe d'une commande

- après connexion, le logiciel qui permet d'interagir avec la machine s'appelle le shell
- une commande se tape au clavier à la suite de l'invite du shell ($). Pour valider la commande, appuyer sur entrée
- exemple :

```
$date
lun oct 25 09:25:47 CEST 2004
```

affiche la date et l'heure locale

Syntaxe d'une commande

- la majorité des commandes admettent un nombre variable d'options, spécifiées à la suite du nom de la commande

```
$date -u
lun oct 25 07:27:45 CEST 2004
```

affiche la date avec l'heure universelle.

- il est possible de rediriger la sortie d'une commande, en particulier dans un fichier, grâce à l'opérateur >

```
$date -u > mon_fichier.txt
$
```

- S'il existe, `mon_fichier.txt` est écrasé, sinon il est créé.
- ceci permet de sauvegarder la sortie des commandes à des fins d'analyse ultérieure

Système de fichiers

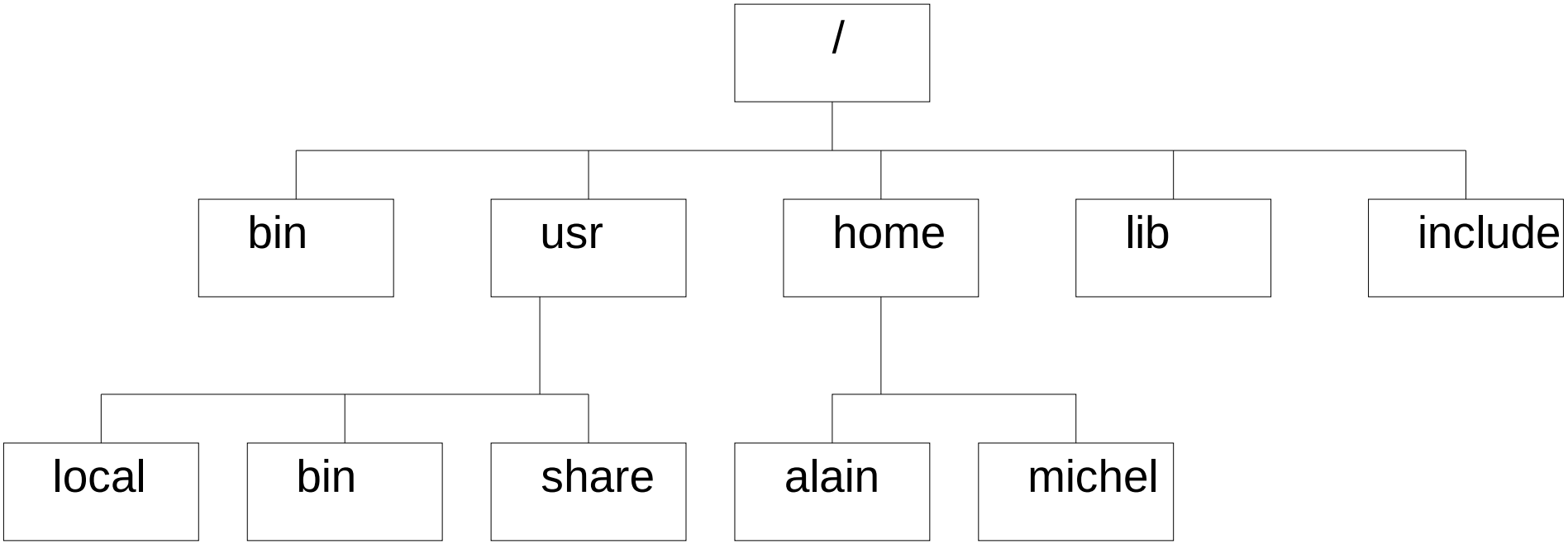
Les fichiers et leur dénomination

- fichier = collection de données formant un ensemble cohérent, accessible via le système d'exploitation grâce à un quelconque support
- chaque fichier est identifié par un nom (minuscules, majuscules, nombres)
- UNIX (Linux) est sensible à la casse
- les extensions (.txt par exemple) ne sont pas obligatoires

Système de fichiers

L'arborescence des fichiers

- la structure du système de fichiers UNIX est arborescent ou hiérarchique,
- chaque fichier est accroché à un répertoire,
- chaque répertoire est accroché à un répertoire parent.



L'arborescence des fichiers

- la commande **pwd** (print working direcctory) permet de connaître le nom du répertoire courant

  ```
  $pwd
  /home/alain
  ```

- UNIX tient à jour une liste de répertoires appelée `PATH`, dans lesquels il recherche le fichier correspondant à la commande appelée
- `PATH` est une variable d'environnement. Pour connaître sa valeur, saisissez :

  ```
  $printenv PATH
  ~/bin:/usr/bin:/bin
  ```

Commandes de base agissant sur les fichiers

commande	**description**
ls	affiche le contenu du répertoire courant
ls -l	affiche en plus des informations sur chaque fichier
ls rep	affiche le contenu du répertoire rep
cat fichier	affiche le contenu du fichier
cp fichier1 fichier2	le contenu du fichier1 est copié dans fichier2
mv fichier1 fichier2	renomme fichier1 en fichier2
rm fichier	supprime le fichier

Commandes de base agissant sur les répertoires

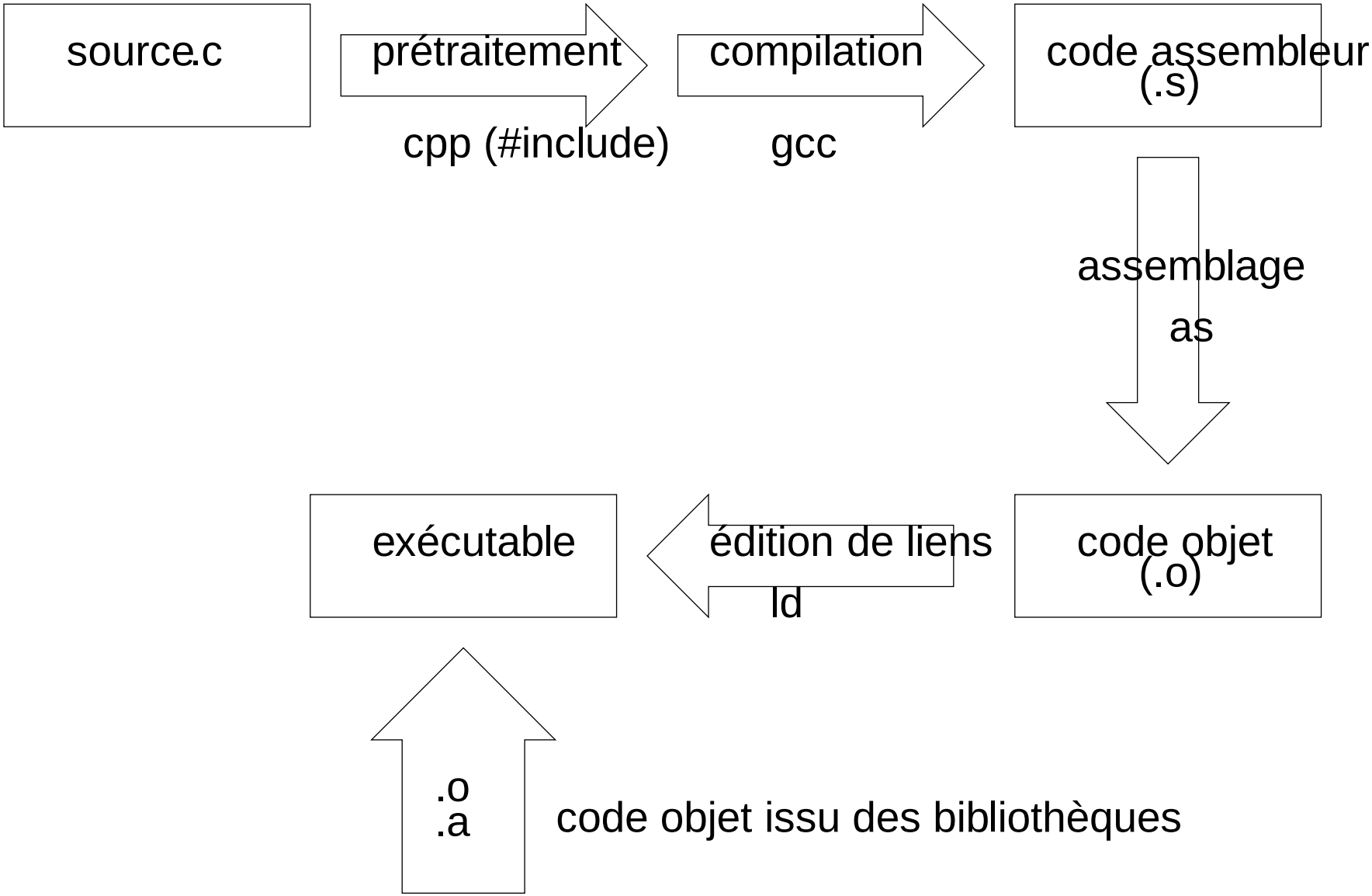
commande	**description**
mkdir rep	crée le répertoire rep
mv rep1 rep2	renomme le répertoire rep1 en rep2
rmdir rep	supprime le répertoire rep (s'il est vide)
rm -rf rep	supprime le répertoire rep (même si non vide). Attention, très dangeureux
cd rep	va dans le répertoire rep
cd ..	va dans le répertoire parent
cd	va dans le répertoire personnel

Le compilateur gcc

Introduction

- gcc signifie gnu compiler collection
- c'est un projet de la FSF (Free Software Fundation)
- c'est un ensemble d'outils (compilateurs, assembleurs)
- permet de compiler du C, C++, objective C, fortran, java, ada, pascal...

Du source à l'exécutable



Tout se fait automatiquement

Ligne de commande

- on suppose que le programme se trouve dans le fichier source.c,
- la compilation se fait avec la commande.

  ```
  gcc source.c
  ```

- une fois terminée, un fichier a.out est créé dans le même répertoire. Il s'agit de **l'exécutable**,
- pour changer le nom de l'exécutable, il faut utiliser l'option -o.

  ```
  gcc -o prgm source.c
  ```

- pour exécuter le programme, il faut se placer dans le répertoire qui le contient et saisir :

  ```
  ./prgm
  ```

Inclusion de bibliothèques

- on utilise l'option -l
- l'option se place toujours **à la fin** de la ligne de commande.

```
gcc -o prgm source.c -lm
```

- pour connaître le nom de la bibliothèque, ajouter le préfixe **lib** et le suffixe **.a**.
- l'argument -lm provoque l'inclusion de la bibliothèque `/usr/lib/libm.a`, bibliothèque contenant les fonctions mathématiques, associées à math.h

Inclusion de bibliothèques

-w	supprime tous les messages d'avertissement
-W	messages soulignant **certaines** techniques autorisées mais discutables
-Wall	messages soulignant **toutes** les techniques autorisées mais discutables
-Werror	les messages d'avertissement provoquent des erreurs et empêchent la compilation

Qu'est-ce que la GSL ?

GSL signifie GNU Scientific Library. Il s'agit d'une bibliothèque numérique pour les programmeurs C et C++. C'est un logiciel libre sous licence GNU General Public Licence

La GSL fournit plus de 1000 fonctions mathématiques au total !

Licence d'utilisation

Contrairement aux bibliothèques numériques propriétaires, la licence de la GSL ne restreint pas la coopération scientifique. Elle vous permet de partager librement vos programmes avec d'autres.

Pourquoi la GSL ?

- facilite la collaboration, la bibliothèque est disponible librement pour tous,
- vous pouvez adapter le code source à vos besoins,
- vous pouvez contribuer à son amélioration.

Sujets couverts par la GSL

- Complex Numbers
- Roots of Polynomials
- Special Functions
- Vectors and Matrices
- Permutations
- Sorting
- Linear Algebra
- Eigensystems
- Fast Fourier Transforms
- Quadrature
- Random Numbers
- Quasi-Random Sequences
- Random Distributions
- Statistics
- Histograms
- Monte Carlo Integration
- Simulated Annealing
- Differential Equations
- Interpolation
- Numerical Differentiation
- Chebyshev Approximation
- Series Acceleration
- Discrete Hankel Transforms
- Root-Finding
- Minimization
- Least-Squares Fitting
- Physical Constants
- IEEE Floating-Point

Sujets couverts par la GSL

- **Complex Numbers**
- Roots of Polynomials
- Special Functions
- **Vectors and Matrices**
- Permutations
- Sorting
- Linear Algebra
- Eigensystems
- **Fast Fourier Transforms**
- Quadrature
- Random Numbers
- Quasi-Random Sequences
- Random Distributions
- Statistics
- Histograms
- Monte Carlo Integration
- Simulated Annealing
- Differential Equations
- Interpolation
- Numerical Differentiation
- Chebyshev Approximation
- Series Acceleration
- Discrete Hankel Transforms
- Root-Finding
- Minimization
- Least-Squares Fitting
- **Physical Constants**
- IEEE Floating-Point

Plates-formes reconnues

La GSL peut être utilisée sur :

- Compatible PC / gcc
- SunOS 4.1.3 et Solaris 2.x (Sparc)
- Alpha GNU/Linux, gcc
- HP-UX 9/10/11, PA-RISC, gcc/cc
- IRIX 6.5, gcc
- m68k NeXTSTEP, gcc
- Compaq Alpha Tru64 Unix, gcc
- FreeBSD, OpenBSD et NetBSD, gcc
- Cygwin
- Apple Darwin 5.4
- Hitachi SR8000 Super Technical Server, cc

Important

Important

La bibliothèque est écrite par des physiciens et s'adresse à des scientifiques non informaticiens. Toute personne sachant programmer en C sera capable d'utiliser directement la GSL

Constantes physiques

La GSL fournit un grand nombre de constantes physiques, dans deux systèmes d'unité :

- MKSA (mètres, kilogrammes, secondes, ampères)
- CGSM (centimètres, secondes, grammes, gauss)

Les fichiers d'en-tête nécessaires sont :

- #include <gsl/gsl_const_mksa.h> pour MKSA
- #include <gsl/gsl_const_cgsm.h> pour CGSM
- #include <gsl/gsl_const_num.h> pour les constantes sans dimensions (purement numériques)

Exemple

```c
#include <stdio.h>
#include <gsl/gsl_const_mksa.h>
#include <gsl/gsl_const_num.h>
int main(void)
{
    double lambda = 1064 * GSL_CONST_NUM_NANO;
    double nu;

    nu = GSL_CONST_MKSA_SPEED_OF_LIGHT / lambda;

    return 0;
}
```

Les nombres complexes

Déclaration :

`gsl_complex z;` définit un nombre complexe nommé z

Fichiers d'en-tête

- `#include <gsl/gsl_complex.h>`
- `#include <gsl/gsl_complex_math.h>`

Initialisation

```
1 gsl_complex gsl_complex_rect(double x,double y)
2 gsl_complex gsl_complex_polar(double r,double t)
3 GSL_SET_COMPLEX(zp,x,y)
4 GSL_SET_REAL(zp,x)
5 GSL_SET_IMAG(zp,y)
```

Exemple

Listing 4 – Initialisation de nombres complexes

```
1  #include <stdio.h>
2  #include <gsl/gsl_complex.h>
3  #include <gsl/gsl_complex_math.h>
4
5  int main()
6  {
7  gsl_complex z;
8  z = gsl_complex_rect(1.2,-2.4);
9  z = gsl_complex_polar(2.68,-1.11);
10 GSL_SET_COMPLEX(&z,1.2,-2.4)
11
12 GSL_SET_REAL(&z,1.2);
13 GSL_SET_IMAG(&z,-2.4);
14
15 return 0;
16 }
```

Exemple 2

Listing 5 – rapport de deux nombres complexes

```
1  #include <stdio.h>
2  #include <gsl/gsl_complex.h>
3  #include <gsl/gsl_complex_math.h>
4
5  int main()
6  {
7  gsl_complex z1 , z2, z3;
8  z1 = gsl_complex_rect( 1.2 , -2.4 );
9  z2 = gsl_complex_rect( -3.2 , 1.1 );
10
11 z3 = gsl_complex_div( z1 , z2 );
12
13 return 0;
14 }
```

Les vecteurs

Déclaration

- `gsl_vector *v;`
- `gsl_vector_complex *v;`

Attention, il s'agit d'un pointeur

Les vecteurs

Déclaration

- `gsl_vector *v;`
- `gsl_vector_complex *v;`

Attention, il s'agit d'un pointeur

Fichiers d'en-tête

```
#include <gsl/gsl_vector.h>
```

Les vecteurs

Déclaration

- `gsl_vector *v;`
- `gsl_vector_complex *v;`

Attention, il s'agit d'un pointeur

Fichiers d'en-tête

```
#include <gsl/gsl_vector.h>
```

Réservation en mémoire

Il est nécessaire de réserver la place en mémoire pour stocker le vecteur. On utilise pour cela les fonctions

```
gsl_vector_calloc( int size )
gsl_vector_complex_calloc( int size )
```

Le vecteur contient alors size éléments tous initialisés à zéro

Exemples

```c
#include <gsl/gsl_vector.h>
int main(void)
{
    gsl_vector *v;
    v = gsl_vector_calloc(1024);
    ...
    return 0;
}
```

Exemples

```c
#include <gsl/gsl_vector.h>
int main(void)
{
   gsl_vector *v;
   v = gsl_vector_calloc(1024);
   ...
   return 0;
}
```

Déclaration et allocation mémoire peuvent se faire sur la même ligne

```c
#include <gsl/gsl_vector.h>
int main(void)
{
   gsl_vector *v = gsl_vector_calloc(1024);
   ...
   return 0;
}
```

Libération de la mémoire

Il est nécessaire de libérer la place en mémoire lorsqu'on n'a plus besoin du vecteur

```
1  void gsl_vector_free( gsl_vector *v  )
```

Exemple :

```
1  #include <gsl/gsl_vector.h>
2  int main(void)
3  {
4     gsl_vector *v;
5     v = gsl_vector_calloc(1024);
6     ...
7     gsl_vector_free( v );
8     return 0;
9  }
```

Accéder aux données

écrire des données dans le vecteur :

```
1  void gsl_vector_set( gsl_vector *v , int
       numero_element , double x )
```

Exemple :

```
1  #include <gsl/gsl_vector.h>
2  int main(void)
3  {
4     gsl_vector *v = gsl_vector_calloc(1024)
5
6     /* place 5.23 dans v[3] */
7     gsl_vector_set ( v , 3 , 5.23 );
8     return 0;
9  }
```

Accéder aux données

Lire des données depuis le vecteur :

```
1  double gsl_vector_get( gsl_vector *v , int
       numero_element)
```

Exemple :

```
1  #include <gsl/gsl_vector.h>
2  int main(void)
3  {
4     double a;
5     gsl_vector *v = gsl_vector_calloc(1024)
6
7     /* place v[3] dans a*/
8     a = gsl_vector_get ( v , 3 );
9     return 0;
10 }
```

Vecteurs et fichiers

Écrire un vecteur dans un fichier :

```
1  int gsl_vector_fprintf(FILE *fp, gsl_vector *v ,
       char* format)
```

écrit le vecteur `v` dans le fichier pointé par `fp` en utilisant la chaîne de formatage `format`. La fonction renvoie 0 en cas de succès.

Vecteurs et fichiers

Exemple :

```
1  #include <stdio.h
2  #include <gsl/gsl_vector.h>
3  int main(void){
4     FILE *fp;
5     fp = fopen("mon_fichier.dat" , "w");
6     gsl_vector *v = gsl_vector_calloc(1024);
7     /* on remplit v */
8     gsl_vector_fprintf( fp , v , "%.2f");
9     gsl_vector_free( v );
10    fclose( fp );
11    return 0;}
```

Vecteurs et fichiers

Lire un vecteur depuis un fichier

```
1  int gsl_vector_fscanf(FILE *fp, gsl_vector *v)
```

lit le vecteur v dans le fichier pointé par `fp` . La fonction renvoie 0 en cas de succès.

Vecteurs et fichiers

Exemple :

```
1  #include <stdio.h
2  #include <gsl/gsl_vector.h>
3  int main(void){
4     FILE *fp;
5     fp = fopen("mon_fichier.dat" , "r");
6     gsl_vector *v = gsl_vector_calloc(1024);
7     gl_vector_fscanf( fp , v );
8     /* on travaille avec v */
9     gsl_vector_free( v );
10    fclose( fp );
11    return 0;}
```

Transformées de Fourier

Définition :

Les transformées de Fourier rapides sont des algorithmes efficaces pour calculer les transformées de Fourier discrètes

$$x_j = \sum_{k=0}^{N-1} z_k \exp(-2\pi ijk/N)$$

Transformée de Fourier

Forward Fourier Transform

$$x_j = \sum_{k=0}^{N-1} z_k \exp(-2\pi ijk/N)$$

Inverse Fourier Transform

$$z_j = \frac{1}{N} \sum_{k=0}^{N-1} x_k \exp(2\pi ijk/N)$$

Backward Fourier Transform

$$z_j = \sum_{k=0}^{N-1} x_k \exp(2\pi ijk/N)$$

La TF inverse n'est pas normalisée. Plus rapide et suffisant si seule la forme de la TF est recherchée

Fichiers d'en-tête

- `#include <gsl/gsl_fft_real.h>`
- `#include <gsl/gsl_fft_complex.h>`

La GSL utilise des fonctions différentes selon que les données dont on veut faire la TF sont réelles ou complexes.

Nous ne traiteront que le cas où les données sont complexes.

Fonctions gsl

```
1 int gsl_fft_complex_radix2_forward (data, stride, n
      )
2 int gsl_fft_complex_radix2_backward (data, stride,
      n)
3 int gsl_fft_complex_radix2_inverse (data, stride, n
      )
4 int gsl_fft_complex_radix2_transform (data, stride,
       n, sign)
```

- Ces fonctions ne fonctionnent que si la taille du tableau data est une puissance de 2.
- n est la dimension du tableau data
- Le nombre stride permet de faire la TF en utilisant par exemple 1 point sur 2 si stride vaut 2. On prendra stride = 1
- forward : sign=-1, backward : sign=+1

Format du tableau

```
data[0] = Re(z[0])
data[1] = Im(z[0])
data[2] = Re(z[1])
data[3] = Im(z[1])
data[4] = Re(z[2])
data[5] = Im(z[2])
```