# Shell / Bash – Programmation en C

Francesco Bronzino
ArchiSys

# Shell / Bash

## Session Shell

Pour vous connecter, vous devez saisir vos informations d'identification sur le terminal :

```
login : <nom d'utilisateur>
password : <mot de passe>
```

Pour se déconnecter : `CTRL+D`, `exit` ou `shutdown` (superutilisateur seulement)

## Session Shell

Il est possible d'utiliser un terminal *distant* en utilisant SSH.

Sur une autre machine en réseau, tapez dans le terminal :

```
ssh <nom d'utilisateur>@<adresse IP de la machine>
```

Il utilise le protocole Secure Shell, qui transmet les commandes et leurs résultats sous forme cryptée sur le réseau.

# Commandes de base

Ils sont saisis dans le terminal. Ils lancent l'exécutable correspondant

**Format** :

```
commande [options] [arguments]
```

**Exemple** : imprimer le contenu de `file.txt`.

```
cat file.txt
```

**Exemple** : liste le contenu du dossier `dir`, y compris les fichiers cachés (commençant par `.`) :

```
ls -a dir
```

```
ls --all dir
```

## Commandes de base

Il est possible d'avoir plusieurs commandes sur une seule ligne, en les séparant par `;`.

```
commande1; commande2; ...
```

**Autres comportements** :

- Les commandes peuvent être concaténées via le caractère `|`.
- La sortie d'une commande peut être redirigée vers un fichier en utilisant le caractère `>`.

## Commandes de base

**Manuel en ligne** : les commandes sont documentées

```
man <commande>
```

Renvoie la page de manuel du `<commande>`.

## Commandes de base

**Autres commandes de base** :

- `apropos` : rechercher tous les manuels de commande
- `whereis` : trouver le binaire, le source et le manuel d'une commande
- `date` : affiche la date
- `who` : montre les utilisateurs actuellement connectés
- `uptime` : durée de vie d'un système, nombre d'utilisateurs connectés, charge du système dans les dernières 1, 5, 15 minutes
- `hostname` : nom de la machine

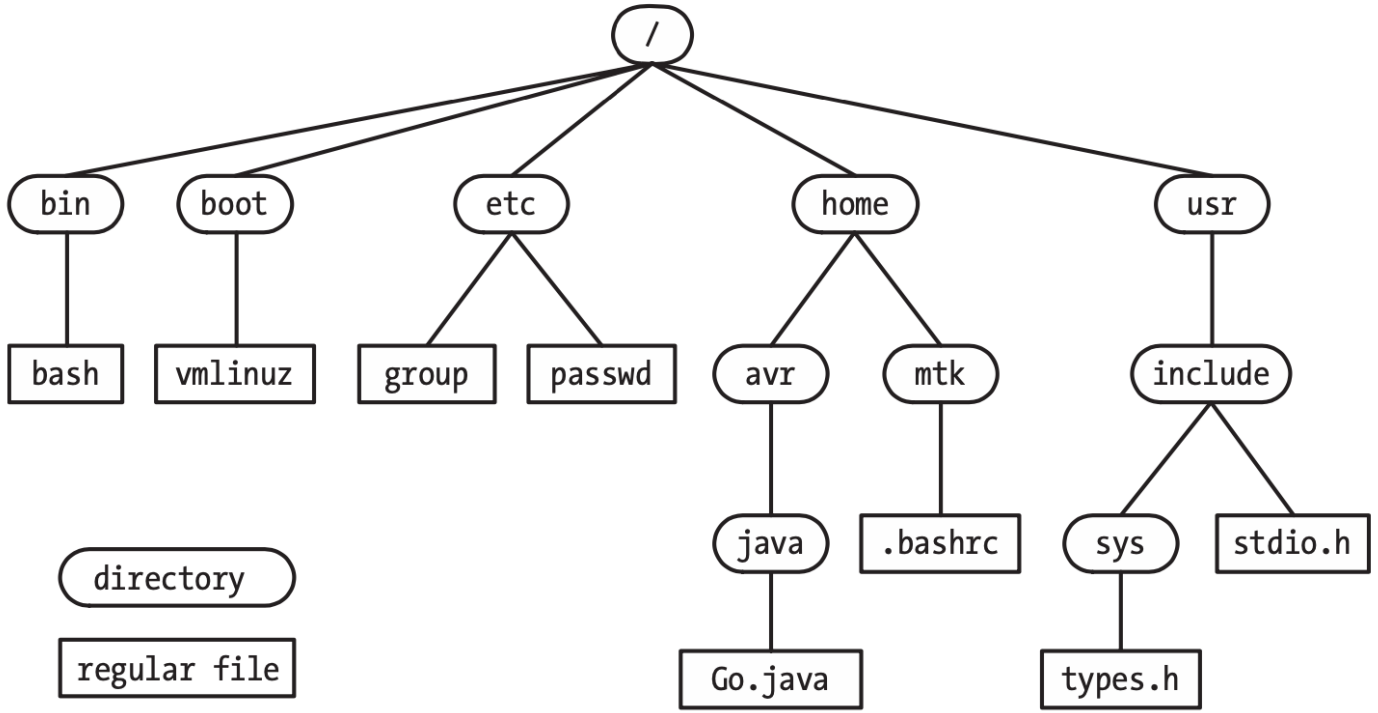
## File System

Le système de fichiers de Linux est hiérarchique.

- Organisé en dossiers imbriqués les uns dans les autres
- Le dossier racine est `/`.
- Tous les dossiers du système sont contenus dans le répertoire racine.
  Exemple : Le domicile de l'utilisateur est situé à
  `/home/username`

# File System

Exemple d'une arborescence de dossiers (partielle) sur un système Linux



## File System

Un **chemin** identifie un fichier ou un dossier.

Un terminal (et les commandes qui sont exécutées à partir de celui-ci) est toujours **positionné** sur un dossier.

- On se déplace d'un dossier à l'autre avec la commande `cd <path>`.
- Un chemin peut être :
  - Absolu : commence par `/` et indique un chemin complet à partir de la racine.
  - Relatif : **non** commence par `/` et indique un chemin relatif au dossier courant
  - Vous pouvez faire référence au dossier courant avec `.`.
  - Le dossier parent du répertoire courant est désigné par `..`.

## File System

Sur tous les systèmes Linux, le système de fichiers est organisé avec les dossiers suivants.

- `/` : racine
- `/bin` : fichiers exécutables du système - `ls`, `pwd`, `cp`, `mv` ....
- `/boot` : fichiers nécessaires pour démarrer le système, chargeur de démarrage ...
- `/dev` : fichiers spéciaux décrivant les périphériques - disques, carte son, ports série ...
- `/etc` : fichiers exécutables, scripts, initialisation, configuration du système, fichiers de mots de passe, ...

## File System

- `/home` : les répertoires personnels des utilisateurs
- `/lib` : bibliothèques système
- `/lost+found` : contient des fichiers endommagés
- `/mnt` : point de montage du système de fichiers
- `/proc` : système de fichiers virtuel qui contient des informations sur les programmes en cours d'exécution.
- `/sys` : programmes système
- `/tmp` : répertoire temporaire

## File System

- `/usr` : fichiers relatifs aux applications installées
- `/usr/include` : fichiers d'en-tête de la bibliothèque C standard
- `/usr/bin` : fichiers binaires disponibles pour les utilisateurs
- `/var` : les fichiers système qui changent fréquemment
- `/var/spool` : zones de spooling temporaires

## File System

**Commande** `ls [options] [fichier]` : liste le contenu du répertoire. Options principales :

- `-1` : impression sur une seule colonne
- `-l` : format long
- `-n` : comme -l mais affiche les IDs au lieu du nom du propriétaire et du groupe
- `-t` : trier par date
- `-s` : montrer la taille du fichier en blocs
- `-a` : affiche tous les fichiers incluant . et .
- `-R` : liste le contenu récursivement

# File System

Exemple:

```
$ ls
compile.txt  style.css  SYS-L2.pptx
```

```
$ ls -l
total 16
-rw-rw-r-- 1 wontoniii wontoniii  102 sep 30 14:16 compile.txt
-rw-rw-r-- 1 wontoniii wontoniii  199 sep 30 15:27 style.css
drwxrwxr-x 3 wontoniii wontoniii 4096 oct  1 18:33 SYS-L2.pptx
```

## File System

**commande** `rm [–rfi] [nom du fichier]` : supprime le ou les fichiers sélectionnés.

**commande** `cd <dir>` : changer de répertoire.

**commande** `mkdir <dir>` : créer un sous-répertoire.
**commande** `rmdir <dir>` : supprime le sous-répertoire, seulement s'il est vide. Sinon, il échoue.

**commandes** `cp <fichier1> <fichier2>` **et** `mv <fichier1> <fichier2>` : copier/déplacer des fichiers ou des répertoires.

## File System

**Commande** `ln [-s] <source> <destination>` : crée un **lien**. Sous Linux, il existe deux types de liens :

- **Lien "soft"** : est une simple référence croisée vers un autre chemin. Si le chemin cible n'existe pas ou est déplacé, le lien ne fonctionne tout simplement pas. L'option `-s` est utilisée.
  - **Flexible** : peut créer un lien vers un autre système de fichiers ou répertoire
- **Lien "hard"** : associe un second chemin d'accès au contenu du fichier. Si le premier fichier est déplacé, le lien reste valide et fonctionnel. Il s'agit de l'option par défaut
  - **Robuste** : ne peut jamais être invalide. Il ne peut pas être utilisé entre différents disques, ni pour lier des dossiers.

## File System

**Commande** `find [path] [-n name] [-print]` : recherche récursive de répertoires

**Exemple:** rechercher les fichiers se terminant par `.txt` dans le répertoire `/tmp` :

```
find /tmp -name *.txt
```

Il est possible d'intégrer diverses **propriétés** des fichiers ou des répertoires :

- Temps de création/modification
- Utilisateur ou propriétaire du groupe
- Taille

**Note** : Ne recherche pas le contenu du fichier

## File System

**Commande** `cat <file>` : imprime le contenu d'un fichier

**commande** `touch <file>` : créer le fichier s'il n'existe pas

**Exemple** : créer un fichier `a.txt`, l'ouvrir avec un éditeur et y écrire `ciao`, puis imprimer le fichier.

```
$ touch a.txt
... éditer avec l'éditeur
$ cat a.txt
Bonjour
```

## File System

**commande** `less <file>` : ouvre le fichier dans un visualiseur interne de l'interpréteur de commandes où vous pouvez faire défiler dans les deux sens

Il existe plusieurs autres commandes permettant d'afficher le contenu d'un fichier.

- Commandes pour imprimer des fichiers binaires
- Commandes pour imprimer les premières ( `head` ) ou les dernières ( `tail` ) lignes d'un fichier
- Éditeurs avancés qui peuvent être utilisés à l'intérieur du shell.
  - Nano` le plus simple
  - Il en existe beaucoup. Les éditeurs concurrents sont `emacs` et `vi` , c'est ce qu'on appelle la *Guerre des éditeurs*.

## Utilisateurs et autorisations

Un appareil fonctionnant sous Linux OS peut avoir plusieurs **utilisateurs**.

- Ils peuvent se connecter à un shell ou à un terminal distant.
- Chaque utilisateur a son propre **répertoire personnel** dans `/home/<user>`.
  Il sert à permettre à l'utilisateur de stocker des fichiers personnels tels que des documents, des images, des programmes.

## Utilisateurs et autorisations

Un utilisateur peut être affecté à un ou plusieurs **groupes**.

- Chaque utilisateur doit avoir au moins un groupe, appelé le **groupe primaire**.
- Un utilisateur peut être assigné à plus d'un groupe.

Mécanisme de groupe d'utilisateurs utilisé pour gérer l'accès aux fichiers et aux ressources.
L'utilisateur **root** existe toujours et possède les privilèges maximums.

## Utilisateurs et autorisations

**Gestion** : Commandes pour créer ou supprimer des utilisateurs et des groupes : `useradd`, `groupadd`, `userdel`, `groupdel`.

- Sur de nombreux systèmes d'exploitation Linux, il existe des commandes plus simples : `adduser`, `addgroup`, `deluser`, `delgroup`.

**Autres commandes** :

- `groups` : imprime les groupes auxquels l'utilisateur actuel appartient
- `whoami` : imprime l'utilisateur actuel
- `su <user>` : changer d'utilisateur (demander le mot de passe)
- `sudo <command>` : exécuter une commande en tant qu'utilisateur `root`, après avoir demandé le mot de passe

## Utilisateurs et autorisations

Les fichiers et les dossiers ont trois types d'**permissions** :

- Droit de **Lecture** : Pour les fichiers, accéder au contenu. Pour les dossiers, listez les fichiers.
- Droit d'**Écrire** : Pour les fichiers, modifier le contenu. Pour les dossiers, créez des fichiers ou des sous-dossiers.
- Droit de **Exécuter/Adresser** :
  - Pour les **fichiers**, il existe une autorisation d' **exécution**. Nécessaire à l'exécution des programmes.
  - Pour les **dossiers**, il existe une **autorisation de croisement**. Nécessaire pour accéder aux sous-dossiers.

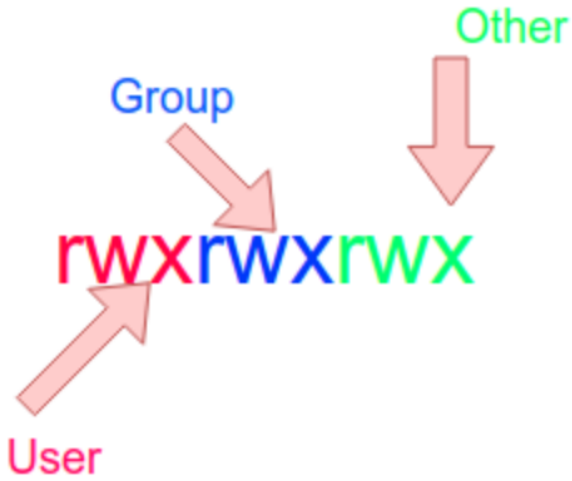
## Utilisateurs et autorisations

Les fichiers et les dossiers ont un **utilisateur propriétaire** et un **groupe propriétaire**.
Les autorisations de fichiers peuvent être gérées séparément pour :

- Utilisateur propriétaire
- Utilisateurs propriétaires
- Tous les autres utilisateurs

# Utilisateurs et autorisations

Au total, chaque fichier ou dossier a $3 \times 3$ de autorisations.




Esempio:

```
$ ls -l
-rw-rw-r-- 1 wontoniii enseignants  102 set 30 14:16 compile.txt
-rw-rw-r-- 1 wontoniii enseignants  199 set 30 15:27 style.css
```

## Utilisateurs et autorisations

**Changer les permissions d'un fichier** : utilisez la commande `chmod [-r] <autorisations> <fichier>`.
Les permissions peuvent être spécifiées avec différentes syntaxes :

- **Absolu**, avec trois chiffres octaux, qui représentent respectivement les permissions à l'utilisateur, au groupe et autre. Chaque chiffre comporte 3 bits et représente les autorisations de lecture, d'écriture et d'exécution/traversée. Exemple : `chmod 750 fichier.txt` donne les autorisations totales à l'utilisateur ($7_8 = 111_2$), lecture/exécution au groupe ($5_8 = 101_2$) et rien aux autres ($0_8 = 000_2$).

# Utilisateurs et autorisations

**Changer le propriétaire et le groupe** d'un fichier ou d'un dossier

- `chown file user` : modifier le propriétaire user
- `chgrp file group` :
- `chown user:file group` : modifier les deux en même temps
- **Note** :
  - Je ne peux attribuer un dossier qu'à un groupe dont je suis le propriétaire.
  - Sur la plupart des systèmes d'exploitation, seul `root` peut changer l'utilisateur propriétaire.
  - Option `-r` : appliquer la commande récursivement aux dossiers et aux fichiers contenus

## Processus et programmes

Un processus est un programme en cours d'exécution.
Sous Linux, chaque processus est identifié par un identifiant appelé **PID**.
Le PID est utilisé pour effectuer des opérations sur le processus.

`kill <PID>` : tuer le processus

`top` : affiche interactivement les processus en cours. Similaire à un gestionnaire de tâches via Shell

## Processus et programmes

`ps [options]` : affiche des informations sur les processus actifs.

- `-a` : informations sur tous les processus (pas seulement ceux générés par la session actuelle du shell)
- `-x` : montre aussi n processus d'arrière-plan
- `-f` : imprime les processus afin que vous puissiez voir leur relation parent-enfant
- `-u` : imprime plus d'informations

**Exemple** : `ps fax`.
**Note:** `ps` est parmi les rares programmes où les options ne doivent pas commencer par `-`. C'est une relique des toutes premières versions d'Unix où les options n'avaient pas le `-`.

## Autres commandes

`lsusb` : liste des périphériques usb

`lspci` : liste des périphériques sur le bus pci

`lsblk` : liste des disques

`ifconfig` : liste des interfaces réseau

`pwd` : affiche le répertoire courant

`free` : montre la quantité de mémoire RAM libre et occupée dont dispose le système.

## Autres commandes

**`df [-htv]`** : Affiche des informations sur les systèmes de fichiers du système.

- `-t` : nombre total de blocs libres et de noeuds i
- `-v` : pourcentage de blocs et de i-nœuds
- `-h` : imprime en GB/MB au lieu du nombre d'octets

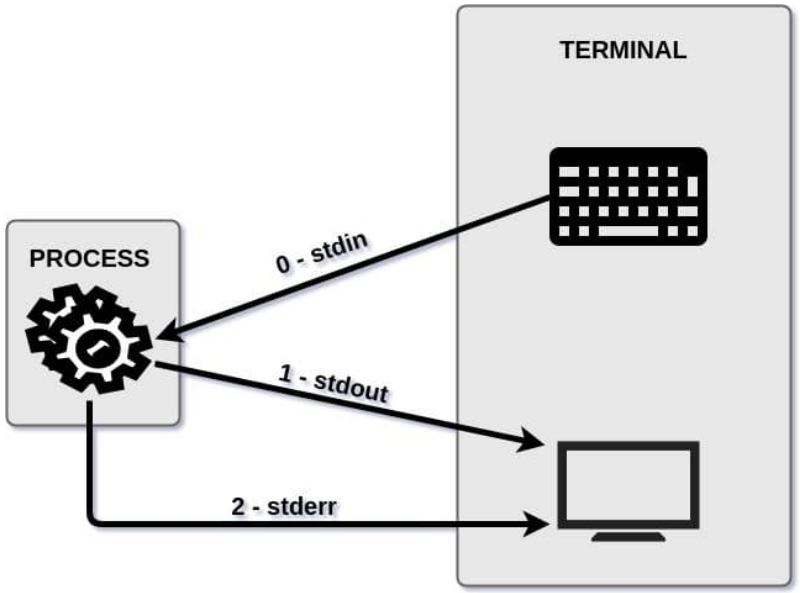
## Pipe et Redirect

Sous Linux, chaque processus dispose de 3 canaux de communication standard.

- **Entrée standard** ( `stdin` ) : pour recevoir les données d'entrée.
- **Sortie standard** ( `stdout` ) : pour imprimer la sortie.
- **Erreur standard** ( `stderr` ) : pour imprimer les erreurs éventuelles

# Pipe et Redirect

Par défaut, un programme reçoit une entrée standard du clavier et imprime une sortie standard et une erreur standard sur la console.



## Pipe et Redirect

Cela implique ce que nous avons déjà vu :

- `read` lit depuis `stdin`, qui est par défaut le clavier.
- `echo` imprime vers `stdout` qui est par défaut la console.
- Pour imprimer dans `stderr`, vous pouvez utiliser : `echo "Une erreur !" >&2`. Par défaut, `stderr` est affiché à l'écran

Tous les programmes bien écrits doivent s'en tenir à l'utilisation de ces *canaux standard*.

- Cela permet une grande flexibilité
- Tous les programmes par défaut de Linux font cela

## Pipe et Redirect

**Redirect vers fichier** : vous pouvez exécuter un programme et rediriger le `stdout` vers un fichier au lieu de l'imprimer.
**Format** : `commande > fichier` ou `commande 1> fichier`.
C'est parce que $1$ indique `stdout` alors que $2$ indique `stderr`.
**Exemple** : `date > données.txt` La date actuelle est enregistrée dans `données.txt` et n'est pas imprimée sur la sortie.
**Note** : si `fichier` existe, le contenu est écrasé.

## Pipe et Redirect

***Append* au fichier** : similaire à la redirection. Le fichier n'est pas supprimé, mais le programme `stdout` est ajouté à la file d'attente.

**Format** : `commande >> fichier` ou `commande 1>> fichier`.

**Exemple** :

```
date > file.txt
sleep 5
date >> file.txt
```

## Pipe et Redirect

**`stderr` vers fichier** : redirige `stderr` vers un fichier.
**Format** : `command 2> file`.
C'est parce que $1$ indique `stdout` alors que $2$ indique `stderr`.

**`stdin` du fichier** : vous permet de `stdin` un programme à partir d'un fichier plutôt qu'à partir du clavier
**Format** : `commande < fichier`.

## Pipe et Redirect

**Exemple** : Ecrivez un programme qui reçoit deux arguments. Elle recherche dans le dossier actuel tous les fichiers qui ont le nom du premier argument et enregistre la liste dans le fichier dont le nom est le deuxième argument

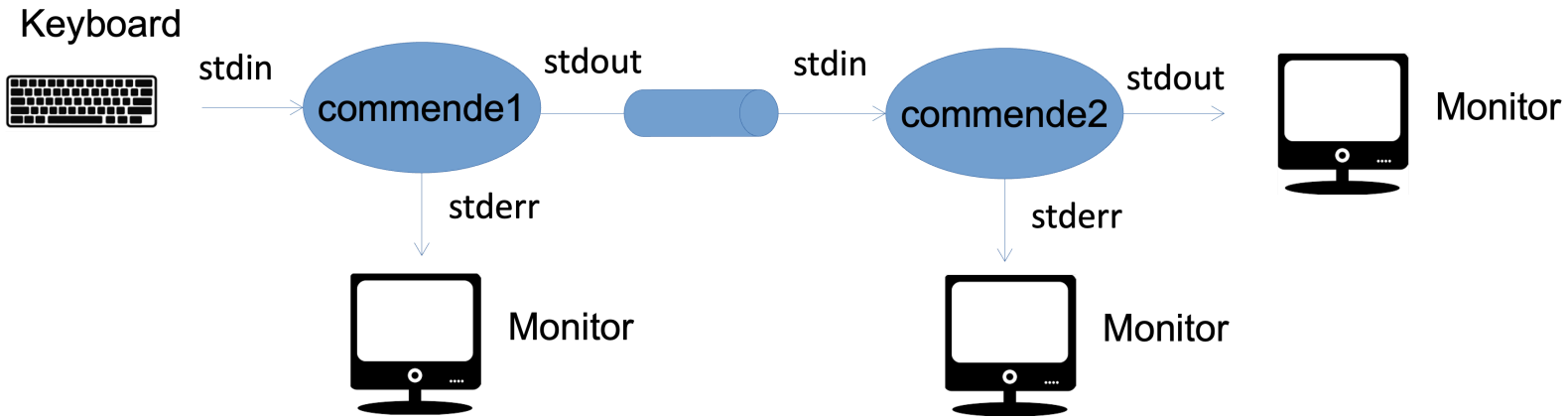
```
#!/bin/bash

if [ "$#" != "2" ]
then
     echo "You need two arguments"
else
     find . -name $1 > $2
fi
```

# Pipe et Redirect

**Pipe** : Vous pouvez rediriger le `stdout` d'une première commande vers le `stdin` d'une seconde.

**Format** : `command1 | command2`



## Pipe et Redirect

**Substitution** : Vous pouvez utiliser le `stdout` d'une commande comme une variable.
**Format** : `$(command)`.
**Exemples** :

- `a=$(ls /tmp)` : la chaîne `a` contient la liste des fichiers dans `/tmp`.
- `rm $( find / -name "*.tmp" )` : supprime tous les fichiers du système se terminant par `.tmp`.

## Filtres et autres

Dans les systèmes d'exploitation Linux, il existe un certain nombre de commandes permettant de manipuler le texte.

- Filtrer, trier, composer

Ils s'attendent à travailler sur des données textuelles organisées en lignes, comme des fichiers texte (ou de configuration) normaux.

Ils vous permettent d'effectuer des opérations complexes avec peu de code.

- Ils sont souvent utilisés en conjonction avec `pipes` pour créer des pipelines de traitement.

# Filtres et autres

**`grep [options] pattern [file...]`** : imprime les lignes du fichier qui contiennent le motif. Si je ne mets pas le fichier en entrée standard : je peux utiliser grep en pipe. Quelques options :

- `-n` : imprime le numéro de ligne
- `-i` : insensible à la casse
- `-c` : imprime le numéro de match
- `-c` : imprime seulement les lignes qui **ne** contiennent pas le motif

**Exemple** :

- `grep -r main *.c` : affiche les lignes contenant `main` dans tous les fichiers se terminant par `.c`.
- `ps -ef | grep bash` : affiche tous les processus qui sont des instances du programme `bash`.

# Filtres et autres

```
#!/bin/bash
if  [  "$#"  !=  "2"  ]
then
        echo  "You need two arguments
else
   cat $1/*.c $1/*.h | grep $2 > /tmp/output.txt
fi
```

## Filtres et autres

`cut` : extrait les colonnes (ou champs) de l'entrée. Il dispose de plusieurs modes.

- **Modes octets** : extrait les octets spécifiés de chaque ligne. L'option `-b byterange` est utilisée.
- **Mode champ** : extrait les champs spécifiés, délimités par un séparateur spécifié. L'option `-d delimiter -c fields` est utilisée.

## Filtres et autres

**`tr [-cds] [set1] [set2]`** : lit les données et remplace les caractères spécifiés par d'autres caractères. Options communes :

- `-d` : supprime tous les caractères spécifiés. Un seul ensemble est nécessaire comme argument
- `-s` : remplace les répétitions du caractère spécifié par un caractère unique

**Exemple** : `tr a A < file1 > file2` : remplace la minuscule `a` par la majuscule `A`. Notez que le `stdin` de `tr` est lu depuis le fichier avec l'opérateur `<`.

## Filtres et autres

**`sort [-dfnru] [-o outfile] [file...]`` : Trie le fichier ou les données stdin`. Options principales :

- `-f` : traite les majuscules comme des minuscules.
- `-n` : reconnaît les nombres et les classe numériquement.
- `-r` : trie les données dans l'ordre inverse.
- `-k` : trie en fonction du numéro de colonne donné après k
- `-u` : trie et supprime les lignes en double
- `-t SEP` : utilise un séparateur de champ différent de celui par défaut (une *transition de non-blanc à blanc*)

## Filtres et autres

**`uniq [-cdu]`** : examine les données ligne par ligne en recherchant les lignes en double et peut :

- Par défaut, éliminer les doublons
- `-c` pour chaque ligne prépare le nombre d'occurrences
- `-d` n'imprime que les lignes dupliquées
- `-u` n'imprime que les lignes uniques
  **Note:** la commande `uniq` **ne trie pas** les lignes. Il est nécessaire de les fournir déjà triés.

**`wc [-lwc] [file]`** : compte les lignes ( `l` ), les mots ( `w` ) et les caractères ( `c` ) de l' `stdin` ou du fichier

# Programmation en C

## Histoire du C

**C** est un langage de programmation :

- **High-level** : il n'est pas écrit en instructions machine
- **Impératif** : le programme est une séquence d'instructions
- **Procédural** : les instructions qui exécutent une tâche sont regroupées en **fonctions**, pour permettre la propreté du code et sa réutilisation.

Parmi les langages de programmation de haut niveau, le C est celui qui se rapproche le plus du langage machine.

- Liberté d'utilisation des adresses mémoire
- Utilisé dans Linux pour écrire le noyau et les pilotes.

# Histoire du C

**Caractéristiques du C** :

- **Langage minimaliste** : quelques concepts simples, proches de ceux du langage machine
  - De nombreuses instructions directement mappables avec une instruction Assebly
  - Seulement 32 mots réservés
- **Rôle central des pointeurs** : les pointeurs sont des variables qui contiennent une adresse mémoire.
  - Ils permettent donc un **adressage indirect**. J'accède à une variable non pas par son nom, mais par son adresse
  - Le programmeur a un très haut degré de contrôle sur la mémoire de la machine, ce qui permet d'optimiser le code.
- **Type statique** : chaque variable a un type de données qui doit être déclaré explicitement par le programmeur.

## Compilation en C

Le C est un **langage compilé**.

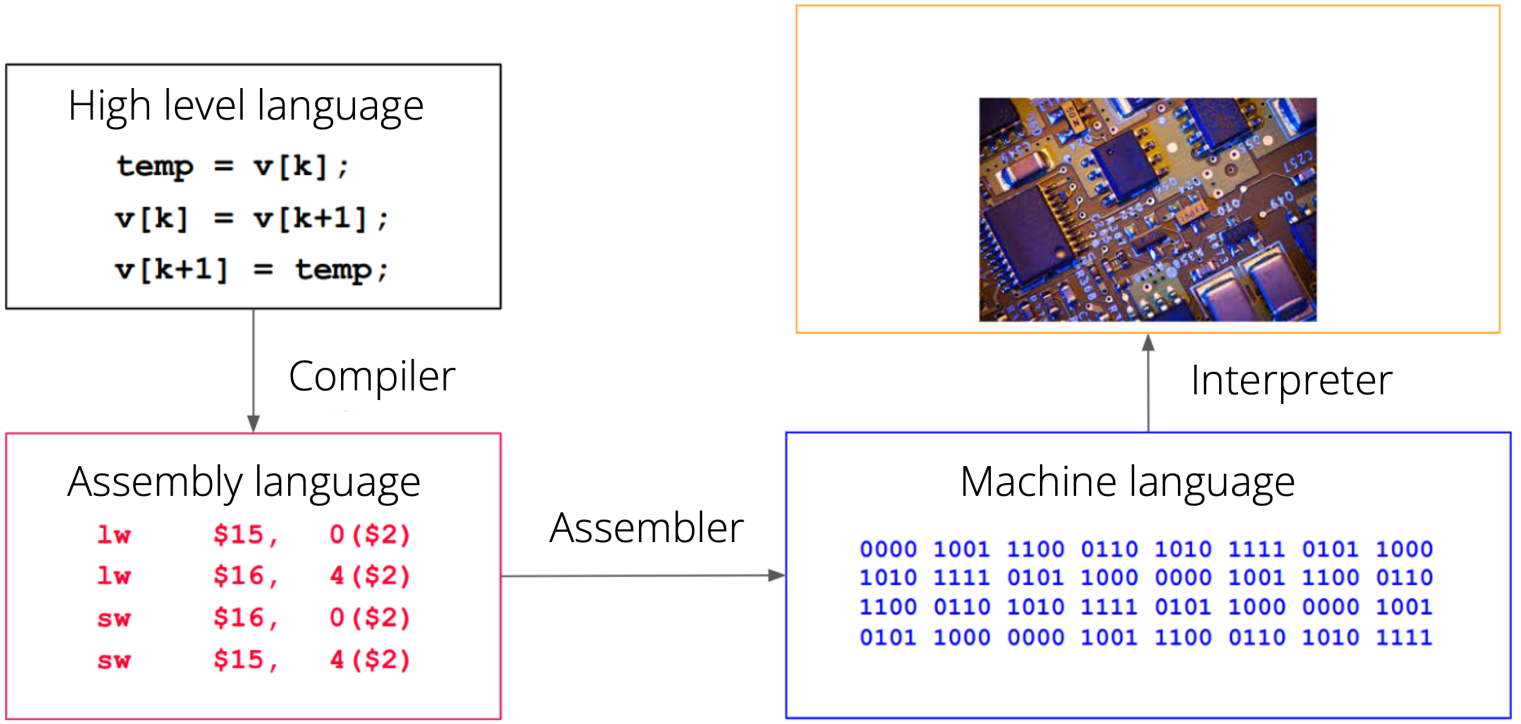
- Un logiciel appelé **compilateur** traduit le code source en un exécutable en langage machine.
- Autres langages compilés : C++, Go

Le C n'est pas un **langage interprété**.

- Un langage interprété est exécuté par un **interprète**, qui lit et exécute les instructions.
- Exemples de langages interprétés : Python, R.

# Compilation en C

## Fonction de compilation :



# Compilation en C

**Phases de compilation** :

1. Le **Préprocesseur** effectue toutes les substitutions textuelles dans le code source. Nécessaire pour les constantes et les macros.
2. Le **Compilateur** crée le code exécutable pour chaque fichier source C.
3. Le **Lieur** assemble le code exécutable en programme final, en le liant aux fonctions de la bibliothèque.
    - Chaque langage C fournit diverses fonctions de bibliothèque pour les calculs mathématiques, l'interaction avec le système d'exploitation, la réalisation d'interfaces graphiques.

## Compilation en C

**Compilation sous Linux** : le compilateur standard `gcc` est utilisé.

Syntaxe :

```
gcc [<options>] file1.c file2.c file3.c  … [-l libraries]
```

Normalement:

```
$gcc file.c
$gcc file.c -c
$gcc file.c -o outfile
$gcc file.c -o outputfile -l library
```

# Makefiles

## What is a Makefile?

- A file that contains a list of files and their dependencies along with commands that you would otherwise type at the command prompt.
- Type `make` at the command prompt and the system searches for a file called makefile or Makefile, and if found, the command for the first target in that file will be run.
- A makefile can have more than one target. If you wanted to execute other targets, you would have specify which one when you type make: `make <target>`

## Makefiles

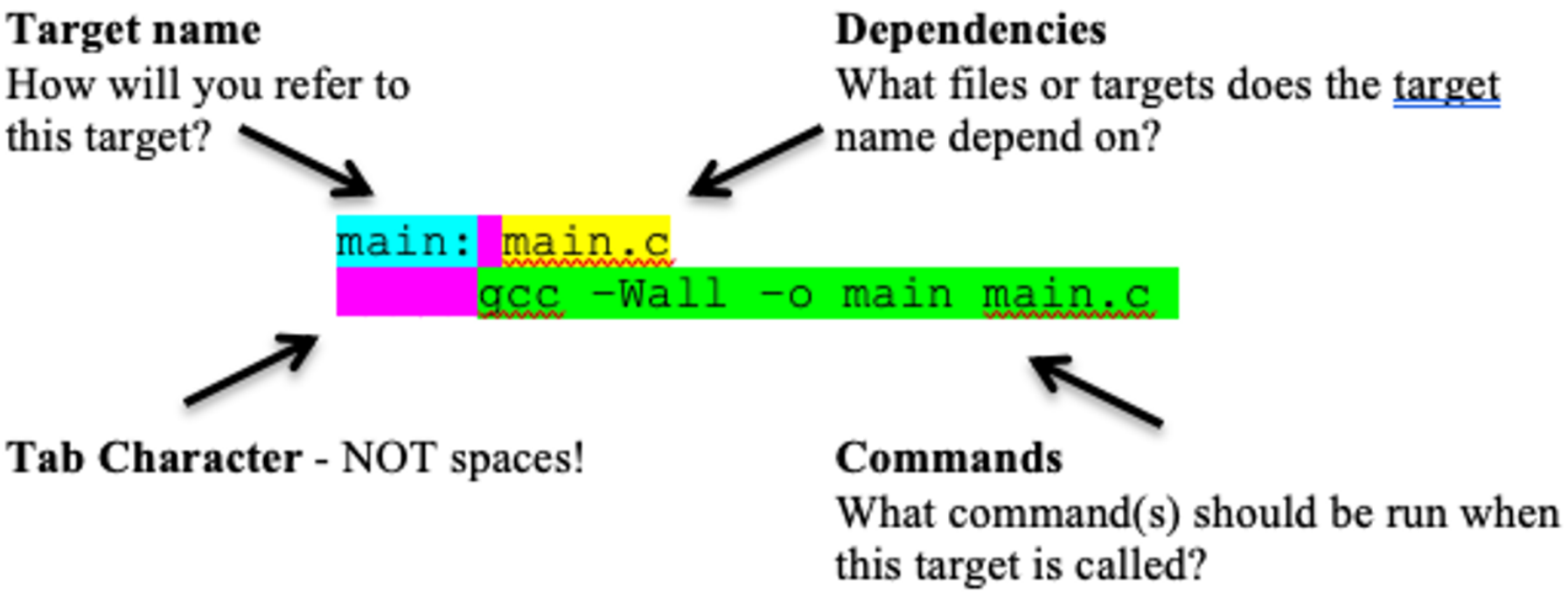
Exemple 1 :

```
main: main.c
	gcc –Wall –o main main.c
```

The above shows a very simple Makefile. If these two lines were put into a file called `Makefile` (or `makefile`), then whenever you would need to compile `main.c`, you would simply just type: `make`

# Makefiles

Exemple 1 :



# Makefiles

Exemple 2 :

```
main: main.o
	gcc -Wall -o main main.c

clean:
	rm -f *.o
	rm -f main
```

# Makefiles

Exemple 3 :

```
CC = gcc
CFLAGS = -g -Wall

main: main.o
	$(CC) $(CFLAGS) -o main main.c

clean:
	rm -f *.o
	rm -f main
```

# Makefiles

Exemple 4 :

```
CC = gcc
CFLAGS = -g -Wall

main: main.o
	$(CC) $(CFLAGS) -o $@ main.c

clean:
	rm -f *.o
	rm -f main
```

# Makefiles

Exemple 5 :

```
CC = gcc
CFLAGS = -g -Wall

.c.o:
        $(CC) $(INCLUDE) $(CFLAGS) -c $<

main: main.o
        $(CC) -o $@ $^ -L. -L/usr/lib -lc -lm

clean:
        rm -f *.o
        rm -f main
```

# Makefiles

Exemple 6 :

```
CC = gcc
CFLAGS = -g -Wall
LDFLAGS = -L. -L/usr/lib
LDLIBS = -lc -lm

.c.o:
        $(CC) $(INCLUDE) $(CFLAGS) -c $<

main: main.o
        $(CC) -o $@ $^ $(LDFLAGS $(LDLIBS)

clean:
    rm -f *.o
    rm -f main
```

# Compilation en C

**Utilisation des bibliothèques** :

Les bibliothèques peuvent être utilisées après les avoir mentionnées avec la directive :

```
#include <library.h>.
```

**Note:**

Les déclarations d'inclusion ne doivent pas être terminées par `;`.

Aucun espace ne peut être inséré en début de ligne.

# Compilation en C

**Les bibliothèques principales:**

- `<stdio.h>` : Fonctions de lecture/écriture de terminal et de fichier
- `<stdlib.h>` : Fonctions de base pour l'interaction avec le système d'exploitation
- `<unistd.h>` : API standard POSIX
- `<math.h>` : Fonctions mathématiques
- `<string.h>` : Fonctions de manipulation de chaînes de caractères
- `<ctype.h>` : Manipulation de caractères

**Autres bibliothèques:**

- `<complex.h>` : Manipulation des nombres complexes
- `<errno.h>` : Traitement des codes d'erreur des fonctions de la bibliothèque
- `<time.h>` : Pour obtenir et manipuler des dates et des heures
- `<limits.h>` et `<float.h>` : Constantes utiles pour travailler sur les nombres entiers et réels

# Compilation en C

**Librairies et appel système:**

- Ces bibliothèques sont rassemblées dans la **bibliothèque standard C (clib)**.
- Les fonctions de bibliothèque **NON** sont des appels système
  - Ils utilisent les appels système en interne
- La "libc" est implémentée sur différents systèmes d'exploitation.
  - Utilisation de différents appels système
- Permet de compiler le même code sur des OS différents

**Exemple:**

Pour ouvrir un fichier, vous utilisez la fonction de la `libc` appelée `fopen`.

- Sous Linux, il utilise l'appel système `open`.
- Sous Windows, il utilise l'appel système `CreateFileA`.

## Variables

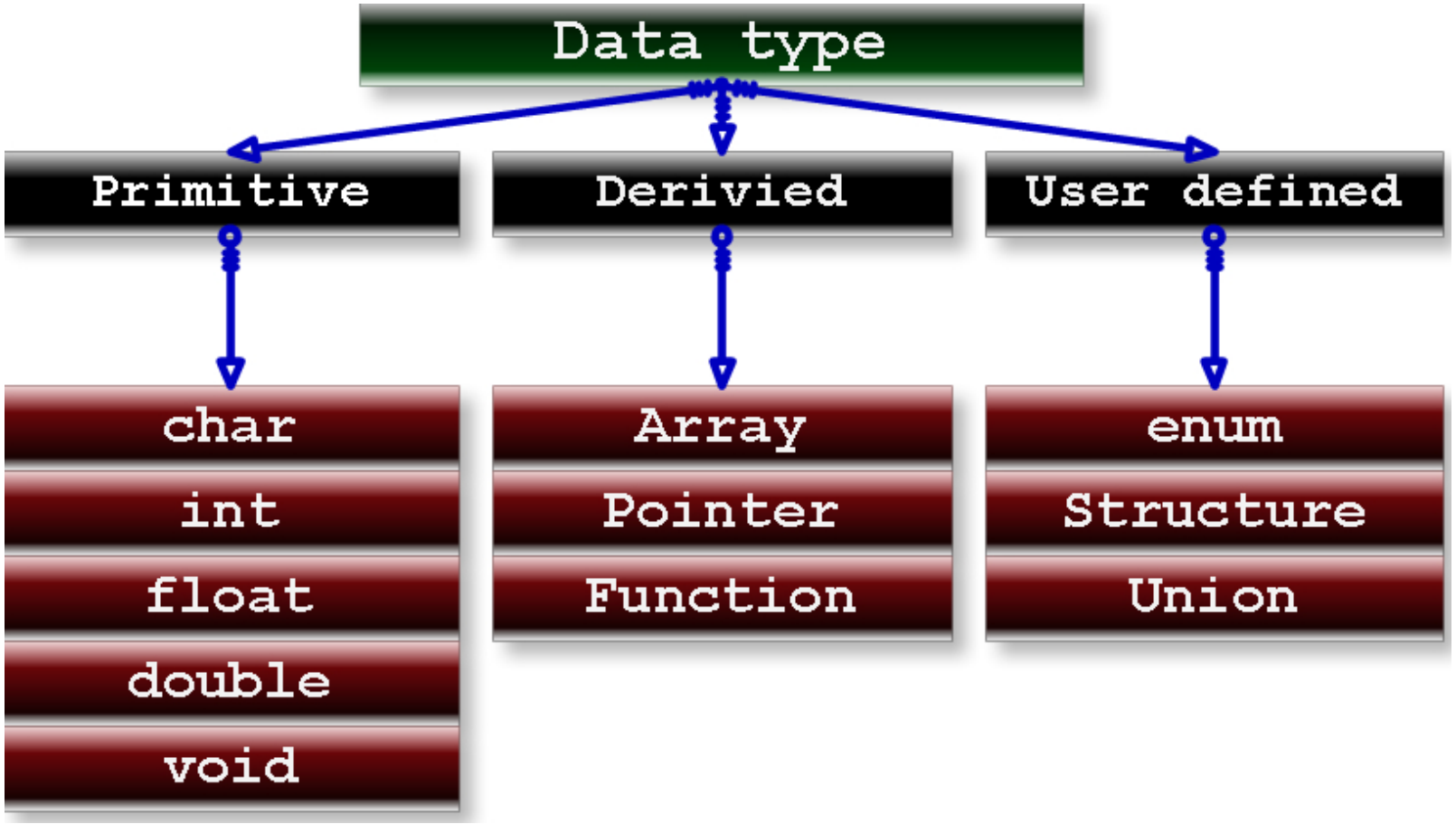
Le C est un langage typé

- Chaque variable ou constante a un type
- Le type est spécifié explicitement par le programmeur.

Les principaux types sont :

- Simple : `int`, `float`, `characters`.
- Dérivés : ensembles de types simples
  - Vecteurs
  - Structure
- Les pointeurs : contiennent les adresses mémoire des variables d'un certain type.

# Variables


Data type
Primitive
Derivied
User defined
char
int
float
double
void
Array
Pointer
Function
enum
Structure
Union

## Variables

**Note:**

En C, les types de données n'ont pas une largeur standard, mais varient d'un système à l'autre.

- Par exemple, un `int` peut être de 16, 32 ou 64 bits.
- Il permet à tout ordinateur de fonctionner à sa taille naturelle.
- Nécessité d'être prudent lors de l'écriture du code

## Les autres types

Les **types de base** en C sont :

- `int` : sont les entiers.
- `float` : sont des nombres à virgule flottante
- `double` : sont des nombres à virgule flottante en double précision
- `char` : sont des variables qui contiennent un caractère.
- `void` : *pas de type*, utilisé dans des situations particulières

## Les autres types

Les **modificateurs** peuvent être utilisés sur les types.
Exemple : `long int` signifie un nombre entier sur plus de bits (par exemple 64 au lieu de 32).

- `long` : force l'utilisation de plus de bits
- `short` : force l'utilisation d'un plus petit nombre de bits
  - `short int a;` indique un entier de plus de 16 bits si la valeur par défaut est de 32 bits.
- `signed` : indique que le type possède un signe. Appliqué par défaut
- `unsigned` : indique les variables qui ne prennent que des valeurs positives
- `const` : déclare une constante.
  - `const float pi = 3.14`.

## Les autres types

Si vous voulez avoir le contrôle sur le nombre de bits dans une variable, vous pouvez utiliser les types :

- `int8_t`
- `int16_t`
- `int32_t`
- `int64_t`
- `uint8_t`
- `uint16_t`
- `uint32_t`
- `uint64_t`

# Les autres types

## Types de données du système

La bibliothèque standard C définit des types de données *aliasés*, définis sur la page de manuel `system_data_types`.

- Ils aident à la portabilité du code.
- Le *alias* indique la cible du type, alors que sur différentes architectures, il est implémenté avec différents types.

### Exemples

- `size_t` : Indique une longueur. Il s'agit généralement de `unsigned int`.
- `off_t` : Indique un décalage. Il s'agit généralement de `int`.

Il y en a plusieurs : `pid_t` `uid_t` `gid_t` `time_t`

# Autres types

## Opérateur `sizeof`.

L'opérateur `sizeof` donne la taille en octets d'un type de données.

- Il retourne un `size_t`.

**Important** car la taille d'un type dépend de la machine.

**Exemple:** sur un PC 64 bits

```
printf("%lu\n", sizeof(char));   // --> 1
printf("%lu\n", sizeof(int));    // --> 4
printf("%lu\n", sizeof(float));  // --> 4
printf("%lu", sizeof(double));   // --> 8
```

## La fonction `printf`.

Utilisée pour imprimer un texte arbitraire sur la console.

- Pour interagir avec l'utilisateur
- Pour imprimer le résultat du traitement
- Pour imprimer les informations qui sont traitées par d'autres programmes via *pipe*.

Contenue dans la bibliothèque `stdio`.
Directive requise :

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

# La fonction `scanf`.

<médium>.

La fonction `scanf` vous permet de demander une entrée utilisateur depuis le terminal, de lire un `int` un `float` (ou autre type)

**Format:**

```
scanf("type", &variable ) ;
```

**Type:**

Pour lire un `int` : `%d`. Pour lire un `float` : `%f`.

**Variable:**

Insérer une variable de type `int` ou `float` déjà déclarée

- Précédé du symbole `&`.
  - Nous verrons que la raison est que la fonction `scanf` requiert un pointeur
  - Avec `&variable` on passe au `scanf` l'adresse de `variable`.

## Opérations de base

**Assignation** : l'opérateur `=` est utilisé.

**Exemples:**

```
int a;
a = 12;
int b;
b = a;
```

```
float f = 12;
f = f + 12;
```

# Opérations de base

**Opérations arithmétiques** :

- Addition : `a + b`
- Soustraction : `a - b`.
- Somme : `a * b`
- Division : `a / b`
  - Note : Si les deux opérandes sont `int`, le résultat l'est aussi.
- Reste de la division : `a % b`.
- Incrémenter : `i++`
- Décroissance : `i--`

## Opérations de base

**Conversion entre les types** : cela s'appelle une opération de casting.

Le format est : `(type) variable`. Par exemple : `(float) a`.

**Exemple** :

```
int a = 5;
int b = 2;
float c;
c = a/b; // --> 2
c = ( (float) a ) / ( (float) b ); // --> 2.5
```

**Parenthèses** : peuvent être utilisées pour imbriquer les opérations de la manière souhaitée.

## Opérations de base

**Opérateurs sur les bits** : effectuer des opérations logiques bit par bit

- `a & b` : *ET* bits à bits
- `a | b` : *OR* bits à bits
- `a ^ b` : *XOR* bit à bit
- `~a` : *NON* bit à bit (opérateur unaire)

**NOTE:** à ne pas confondre avec les opérateurs logiques ( `&&` , `||` , `!` , que nous verrons plus tard)

# Contrôle du flux

## Définition

La capacité d'exécuter des jeux d'instructions alternatifs en fonction de l'occurrence d'une condition.

- La base de presque tous les programmes.
- Une condition doit être définie, c'est-à-dire une **expression booléenne** qui peut être vraie ( `true` =) ou fausse ( `false` ).
- Nous verrons bien :
  - Opérateurs de comparaison : `==', !=', `<', >', `<', >=', `<=`.
  - Opérateurs booléens : `&` , `|` , `!``

# Contrôle du flux

## Instruction `if-else`.

```
if (condition)
{
     A;
}
else
{
     B;
}
```

# Contrôle du flux

## Instruction `switch`

```
switch (expression)
{
    case v1:
        A;
    break;
    case v2:
        B;
    break;
    default:
        C;
}
```

# Contrôle du flux

## Opérateur ternaire

Permet d'écrire de manière concise une expression `if then else`.

**Syntaxe** :

```
condition ? expression1 : expression2;
```

Équivalent à :

```
if(condition)
    expression1;
else
    expression2;
```

# Contrôle du flux

## Opérateur ternaire

### Limitation

`expression1` et `expression2` ne peuvent être que des expressions, pas des instructions !

L'opérateur ternaire peut être utilisé pour fournir une expression dans une instruction

### Exemples

Expression:

```
c = (a < b) ? a : b; //
```

Instruction:

```
a > b ? printf("%d\n", a) : printf("%d\n", b);
```

# Cycles

## `while` loop

S'exécute jusqu'à ce qu'une condition soit vraie.

```
while ( C )
{
    A;
}
```

Comportement :

1. Évaluer C.
2. Si C est faux, le bloc d'instructions est ignoré.
3. Si C est vrai, exécute le bloc d'instructions A et revient à l'étape 1

# Cycles

## `for` loop

```
for ( I; C; A )
{
    B;
}
```

- `I` est l'instruction **initialisation**.
- `C` est la **condition** de terminaison.
- `A` est la mise à jour **instruction**.

# Cycles

## Instructions spéciales

Dans les boucles `for` et `while`, vous pouvez utiliser les instructions spéciales suivantes.

- break` : termine la boucle immédiatement, en passant à l'instruction suivante dans la boucle.
- `continue` : passe immédiatement à l'itération suivante sans exécuter les instructions restantes du bloc.

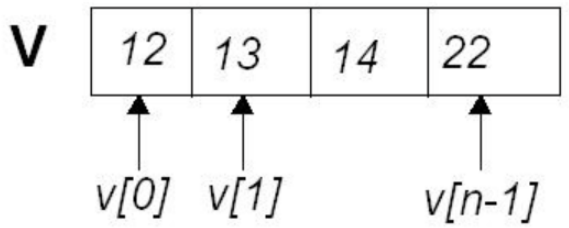
## Vecteurs

Les types de données simples ( `int` ou `float` ) ne peuvent contenir qu'un seul type de données à la fois.
En C, vous pouvez créer des types de données complexes, qui contiennent plusieurs valeurs. Nous verrons bien :

- Vecteurs ou `array`
- Structures ou `struct`

## Vecteurs

**Définition:** Un vecteur ou `array` est une collection de variables du même type. Il est constitué de `N` cellules, chacune identifiée par un **index**.



# Vecteurs

**Définition d'un vecteur :**

```
nome type [N];
```

Exemple:

```
int vecteur [10];
```

Définit un tableau appelé `vecteur` composé de 10$ entiers ( `int` ).

## Vecteurs

**Note:** La longueur du vecteur doit être connue lors de la **compilation**. Ce doit être une constante.

Le code suivant est incorrect :

```
int N;
scanf("%d",&N);
float data [N];
```

Il existe des méthodes pour créer des tableaux de longueur arbitraire en C (`malloc`).

## Vecteurs

**Définitions alternatives** : vous pouvez définir et initialiser un vecteur en même temps.

```
int v[4] = {2, 7, 9, 10} ;
```

Dans ce cas, vous pouvez omettre la longueur, qui est automatiquement saisie par le compilateur.

```
int v[] = {2, 7, 9, 10} ;
```

## Vecteurs

**Accès aux éléments:** L'index doit être spécifié. La syntaxe est la suivante.

```
nom du vecteur [valeur de l'indice]
```

Exemple :

```
int v [] = {4,5,6};
printf("%d\n", v[1]); // --> 5
```

## Les `struct`

Les **structures** ou `struct` sont des collections qui contiennent des variables pas nécessairement du même type.
Opération :

1. Vous définissez une `struct`, un nouveau type de données complexe composé de plusieurs enregistrements.
2. Vous créez et utilisez des variables du type que vous venez de créer.

# Les `struct`

## Définir une `struct` :

```
struct nom {
     champs
} ;
```

## Exemple:

```
struct point{
     float  x ;
     float  y ;
} ;
```

## Les `struct`

**Utilisation de `typedef`** : pour éviter de devoir préfixer `struct` à chaque fois que vous créez une variable, vous pouvez utiliser le mot-clé `typedef`, avec la syntaxe suivante.

```
typedef struct{
    champs
} nom ;
```

**Exemple:**

```
typedef struct{
    float x ;
    float y ;
} point ;
point p1, p2 ; // Vous pouvez omettre struct
```

# Les `struct`

## Observations

Il n'est pas possible de comparer deux `struct` avec les opérateurs `==` ou `!=`.

Il est nécessaire de comparer tous les champs

```
point p1, p2 ;
if (p1==p2) // Faux !
    ...
if (p1.x==p2.x && p1.y == p2.y) // Correct
    ...
```

Initialisation : peut être effectuée comme pour les vecteurs.

```
point p1 = {1.1, 2.4} // x=1.1 et y = 2.4
```

## Les `struct`

Les `struct` sont largement utilisées pour créer de nouveaux types de données complexes :

- Ils sont comme des enregistrements dans une base de données
- Exemples : numéro complexe, adresse de rue, etc...

Ils sont largement utilisés dans les bibliothèques C :

- Ils permettent de créer des types de données arbitraires
- Pour représenter les structures du système d'exploitation.
- Exemples : variables de synchronisation, paquets de réseau, etc...

## Chaînes de caractères

- Une **chaînes de caractères** est une séquence de caractères.
- En C, les **Null-terminated string** sont utilisées pour faciliter les opérations
- Un caractère de terminaison est toujours ajouté à une chaîne de caractères.

**Exemple**: représenter dans un vecteur de longueur 10 la chaîne de caractères `ciao`.

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| Valeur | 'c' | 'i' | 'a' | 'o' | '\0' | ? | ? | ? | ? | ? |

## Fonctions sur les chaînes de caractères

Les fonctions courantes sur les chaînes de caractères sont implémentées dans la bibliothèque C standard.

Nécessaire d'inclure :

`#include <string.h>`.

## Fonctions

Une **fonction** est un ensemble d'instructions qui réalise une tâche commune.

- Pour rendre le code ordonné
- Permettre la réutilisation du code
- Pour avoir un code générique

## Fonctions

Une fonction délimite un morceau de code réutilisable.

- Il peut recevoir des arguments d'entrée
- Il peut fournir une valeur de retour

Après avoir été définie, la fonction est **invoquée**, c'est-à-dire utilisée.

Le `main` est une fonction. Il est invoqué par le système d'exploitation lorsque le programme est lancé.

- Il reçoit des arguments (pas toujours, nous verrons)
- Retourne un `int`.

# Fonctions

**Arguments d'une fonction:**

Spécifier les données sur lesquelles la fonction doit travailler.

- Ils rendent la fonction générique
  - Mais une fonction ne peut pas recevoir d'arguments
- La fonction ne doit pas opérer **seulement** sur des arguments

**Esempio**:

```
float  racine  ( float numero ){...}
```

# La fonction `main`

## Definition :

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

## Arguments:

- `int argc` : nombre de paramètres sur la ligne de commande.
  - En l'absence de paramètres, cela vaut 1.
  - Incrémenté pour chaque paramètre.
- `char* argv[]` : vecteur de paramètres.
  - C'est un vecteur de pointeurs de caractères.
  - Chaque pointeur de caractère dans le vecteur est un argument sous la forme d'une chaîne de caractères.
  - argv[0] `est toujours le nom du programme. Les paramètres réels commencent par` argv[1]`.

# La fonction `main`

**Valeur de retour** :
le `main` **peut** retourner dans `int` qui est examiné par l'OS.
Il indique s'il y a eu une erreur dans l'exécution. Par convention.

- $0$ indique qu'il n'y a pas eu d'erreur
- Un nombre autre que $0$ indique une erreur.
    - La signification du numéro est spécifique au programme.

Ce système est largement utilisé :

- Dans les scripts Bash qui ont besoin de savoir si les programmes exécutés ont eu une erreur
- Pour les modules du système d'exploitation, qui doivent lancer des services, des démons et des programmes en arrière-plan.

## Les pointeurs en C

Un **pointeur** est une variable qui contient une adresse mémoire.
Nous ne nous soucions pas de la manière dont l'adresse est structurée.

- Il s'agit toutefois d'une adresse virtuelle, qui est traduite en une adresse physique par l'unité de gestion de la mémoire.

En C, une variable pointeur contient une adresse mémoire où se trouve une variable d'un certain type.

- Chaque fois que je déclare un pointeur, je dois aussi déclarer quel type de données l'adresse mémoire qu'il contient
- Indispensable pour une utilisation pratique

# Les pointeurs en C

**Déclaration d'un pointeur:** `type * name;`

Exemple :

```
int * pi ; // Pointeur vers int
float * pf ; // Pointeur vers le float
```

Le type `int *` indique une variable pointeur d'entier : il contient l'adresse mémoire à laquelle une variable entière peut être trouvée.

## Les pointeurs en C

**Pour assigner un pointeur:** vous pouvez utiliser l'opérateur `&` pour obtenir l'adresse d'une variable existante.

Exemple :

```
int a = 5 ;
int * pi ;
pi = &pa ; // pi contient l'adresse d'un
```

**Note:** les adresses sont des entiers. On ne sait pas combien de temps, cela dépend de l'architecture.

# Les pointeurs en C

**Accès à la variable pointeur:** l'opérateur `*` appliqué à un pointeur est utilisé pour obtenir la variable pointeur. Appelé l'opérateur de déréférencement.

Exemple :

```
int a = 5, b ;
int * pi ;
pi = &a ; // pi contient l'adresse de a
b = *pi ; // b contient la valeur 5
```

L'opérateur `*` est l'inverse de `&` :

Par conséquent : `*(&a) == a` et `&(*pi) = pi`.

## Pointeurs et vecteurs

**Différences entre les pointeurs et les vecteurs** :

- Un pointeur peut être réaffecté pour pointer vers une autre adresse.
- Un vecteur est techniquement un pointeur constant. On ne peut pas lui attribuer une autre valeur.

```
char v[10], *pv ;
pv = v ; // Autorisé
pv = v + 3 ; // Autorisé
v = pv ; // Erreur !
v = pv + 2 ; // Erreur !
```

## Pointeurs vers `struct`.

**Pointeurs vers `structure` :** un pointeur peut sans risque pointer vers une `structure`.

```
struct point {float  x ;  float  y;}
struct point p1 = {1,  4} ;
struct point * pp ;
pp = &p1 ;  // J'assigne à pp l'adresse de p1
```

S'il peut accéder aux éléments d'une structure par pointeur.

```
(*pp).x ;  // Contient la valeur 1
(*pp).y ;  // Contient la valeur 4
```

## Pointeurs vers `struct`.

**L'opérateur `->`** : est utilisé sur les pointeurs vers `struct`.
vous permet d'accepter directement un champ de la `structure` pointée.
**Syntaxe :** `pointeur->champ`.
**Exemple:**

```
struct point {float x ; float y;}
struct point p1 = {1, 4} ;
struct point * pp = &p1 ;
```

Les instructions suivantes sont équivalentes et retournent le `int` 1.

```
(*pp).x ;
pp->x ;
```

## Pointeurs vers les fonctions

Il est possible de passer des fonctions comme arguments à une fonction.
Il est utilisé pour rendre le code générique et modulaire, pour le multithreading, etc...

Nous verrons quelque chose lorsque nous traiterons des *threads*.

- Exemple : j'utilise une fonction pour créer un thread, et comme argument, je spécifie quelle fonction de mon programme le thread exécute.

```
void  mytask(){....}
pthread_create(..., mytask, ...)
```

## Pointeurs de fonction

Un **pointeur de fonction** représente la position d'une fonction.

**Déréférencer** signifie **invoquer** la fonction.

Un pointeur de fonction est tapé :

- Il peut pointer vers des fonctions qui renvoient un type bien défini
- Et accepter un certain type d'arguments