

# Techniques et Outils d'Ingénierie Logicielle

...

John Gliksberg

# Objet du Cours

- Maîtriser un *shell* et ses automatisations
- Avoir un environnement de développement productif
- Savoir compiler un programme
- Les phases de la compilation
- Le cycle de vie d'un programme
- Gestion de code source (*versioning*)
- Standards de codage
- Validation des programmes (*testing*)

# Temporalité (Peut bouger car auto-adaptatif !)

- Cours 1:
  - Maîtriser un shell et ses automatisations
  - Avoir un environnement de développement productif
  - Savoir compiler un programme (lib, makefile, gcc, search paths)
  - Comprendre les phases de la compilation
- Cours 2:
  - Cycle de vie d'un programme
  - Gestion de code source (Versionning) == GIT
  - Principe de forge (Issue, MR, fork, Pull)
  - Notion d'open Source
  - Gestion des conflits de code-source
- Cours 3:
  - Méthodologie pour le refactoring
  - Importance des standards de codage
    - Exemple de PEP8
    - Exemple de RUST
  - Expressions régulières
  - Validation des codes et invariants
  - Utilisation du versionnage comme filet
  - Documentation
  - La place des LLM
- Cours 4:
  - Débogage de programmes
  - Principe et structure des débogueurs
  - Utilisation de GDB
  - Structure d'un binaire
  - Structure d'un binaire lors de l'exécution
  - Débogage mémoire
- Cours 5:
  - Optimisation de code
  - Outils de mesure
  - Méthodes de mesure
  - Optimisation d'algorithmes

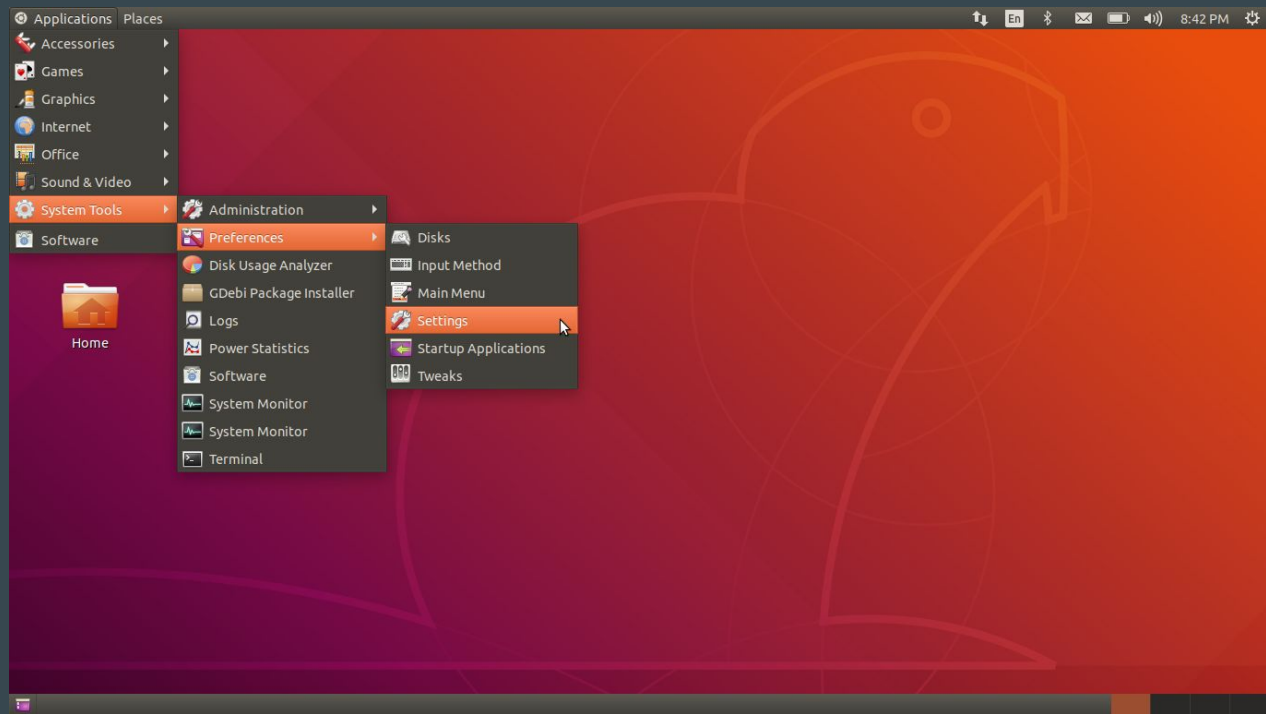
Pour récupérer les slides et exemples

git clone <https://github.com/trosh/TOI24>



**Préparer son Environnement**

# Mettez-vous à l'aise dans votre environnement



# Mettez-vous à l'aise dans votre environnement



# Éléments de l'environnement

- Choix de l'environnement de bureau
  - Gnome
  - KDE
  - Sway
  - ...
- Choix du Shell
  - Configuration du prompt (PS1)
  - Mise en place d'alias
  - Automatisation courantes
- Commandes alternatives (modernisation des outils Unix)
- Choix de l'éditeur de code
  - Un en mode texte
  - Un en graphique
- Déploiement des outils de programmation de base
- Configuration de Git
- ...



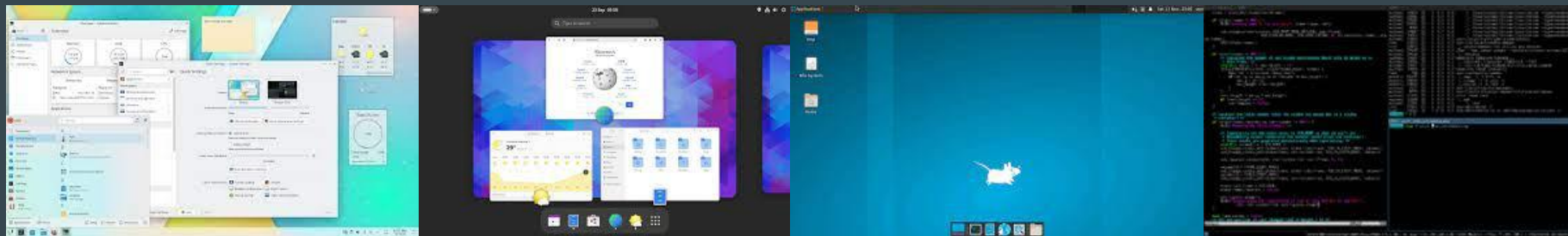
# Les raisons de soigner son environnement

- Vous allez vivre dans l'ordinateur, rendez le agréable
  - Alignez et rangez tous vos outils à portée de main
  - Rendez l'environnement lisible et léger
- Rendez votre environnement portable
  - Sauvez vos "dotfiles" dans un répertoire git
- Un environnement dont vous avez l'habitude vous rendra
  - Efficace
  - Heureux
  - Unique
- Utilisez tous les outils qui vous rendront plus productifs
- Automatisez **toutes** les tâches répétitives



# Il existe différents *desktop environments* (DEs)

- **KDE** : Un environnement de bureau complet et polyvalent, connu pour son interface utilisateur « à la Windows », mais très configurable
- **Gnome** : Un environnement fait pour être intuitif, le choix par défaut de nombreuses distributions, mais peut être plus complexe à configurer que KDE ou XFCE.
- **XFCE** : Léger et facile à comprendre pour les utilisateurs expérimentés car il ne dispose pas d'une interface graphique aussi riche que celle de Gnome ou KDE.
- Considérez les *tiling window-managers* (TWM) :
  - **i3** : Un gestionnaire dynamique de fenêtres qui est léger et configurable, mais peut être difficile à prendre en main au premier abord
  - **Sway** : Un remplacement pour i3 conçu par les développeurs de Wayland



# Le Système D'exploitation

# Anatomie d'un *OS Tree* Linux

```
$ tree -L 1 /
```

```
/
├── bin
├── boot
├── dev
├── etc
├── home
├── lib
├── lib64
├── media
├── mnt
├── opt
├── proc
├── root
├── run
├── sbin
├── sys
├── tmp
├── usr
└── var
```

Voici les répertoires à la racine « / » d'un Linux  
(Dépend de la distribution !)

métaphore : [tout est un fichier](#)

# Anatomie d'un *OS Tree* Linux (selon la FHS)

Répertoire	Description
/bin	Binaires de base
/boot	Contient les image du noyau ( <i>kernel</i> ) et les images de <i>boot</i> ainsi que la configuration du <i>bootloader</i>
/dev	Contient les appareils ( <i>devices</i> ) matérialisés sous forme de fichiers
/etc	Contient les fichiers de configurations pour le système et les services (apache, nfs, slurm ...)
/home	Contient les répertoires utilisateurs
/lib	Contient les bibliothèques partagées principales (libc)
/mnt	C'est ici que l'on monte temporairement des systèmes de fichier (par exemple une clef USB)
/opt	Emplacement générique des logiciels commerciaux
/proc	Système de fichier virtuel contenant de nombreuses informations sur les processus et l'état de la machine
/root	« <i>home</i> » du super-utilisateur
/sbin	Binaires destinés au super-utilisateur uniquement
/sys	Configuration du kernel et matérielle (également via des fichiers)
/tmp	Répertoire temporaire généralement attaché à un ramfs
/usr	Répertoire où la majorité des programmes sont installé (aussi connu comme préfixe)
/var	Répertoire où se situent les logs (/var/log) et les données des serveurs (par exemple /var/nginx/html)

# Anatomie d'un préfixe

```
$ tree -L 1 /usr/  
/usr/  
├── bin  
├── include  
├── lib  
├── local  
├── share  
└── src
```

Répertoire	Description
/usr/bin	Binaires des programmes installés
/usr/include	Headers des bibliothèques installées
/usr/lib	Bibliothèques (partagées et statiques installées)
/usr/share	Dépendances complémentaires (images, ressources, scripts) des programmes
/usr/src	Sources des programmes installés
/usr/local/bin	Binaires des programmes installés (à partir des sources et non de paquets)
/usr/local/include	Headers des bibliothèques installées (à partir des sources et non de paquets)
/usr/local/lib	Bibliothèques (partagées et statiques installées) (à partir des sources et non de paquets)
/usr/local/share	Dépendances complémentaires (images, ressources, scripts) des programmes (à partir des sources et non de paquets)
/usr/local/src	Sources des programmes installés (à partir des sources et non de paquets)

# Le Shell

# Un shell sert à ...

- Lancer des programmes
  - Leur passer des arguments
  - Entrer des données sur leur entrée standard (*stdin*)
  - Afficher leur sorties standard (*stdout*) et d'erreur (*stderr*)
- Le shell est utilisé dans un pseudo-terminal qui est capable de gérer des séquences spéciales de caractères pour :
  - L'affichage (échappement curseur et couleur)
  - La gestion des processus (^C, ^Z)

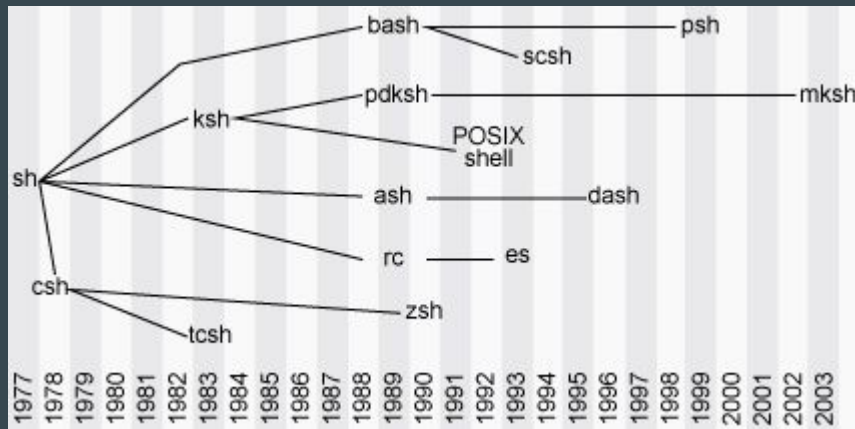


# Utiliser un shell (« invite de commande » = *command prompt*)

- Le shell est l'interface textuelle de prédilection des programmeurs
- Il est répandu sous Linux mais présent aussi sous Windows (WSL)
- Malgré son apparente simplicité il est très productif

```
[0] jbbesnard@deneb~  
% cowsay Hello There  
  
< Hello There >  
-----  
      \      ^__^  
      \      (oo)\_______  
         (_____)  )\_____  
                 ||----w |  
                 ||     ||  
  
[0] jbbesnard@deneb~  
% █
```

# Choix du Shell



C'est en général une question de goût.

Cependant, il est souvent pratique de garder la compatibilité POSIX.

Pour cette raison Bash et Zsh sont très utilisés → bashismes dans le shell OK, dans les scripts ⚠

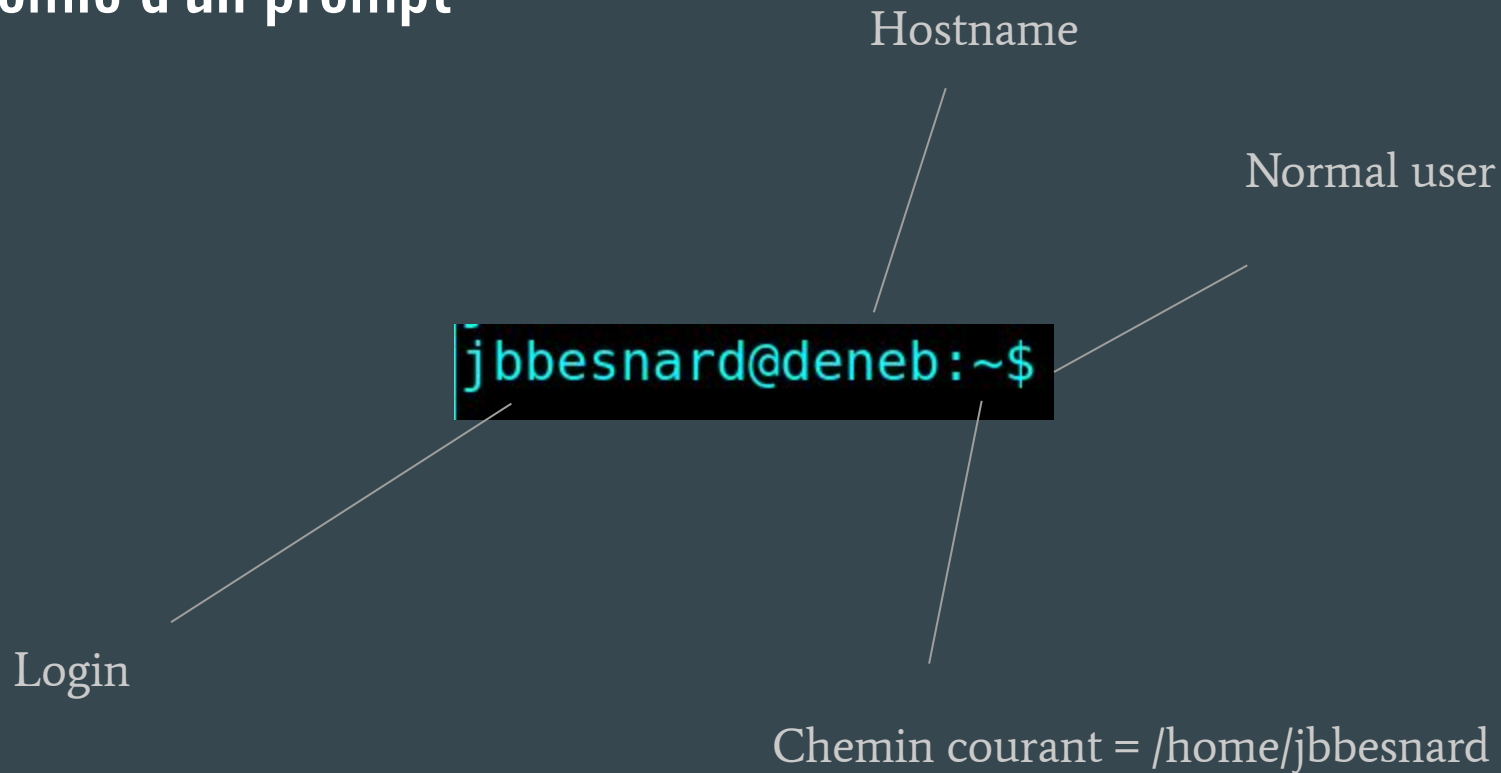


# Anatomie d'un prompt



```
jbbesnard@deneb:~$
```

# Anatomie d'un prompt



# Les Fichiers / Dossiers

```
.└─ toto
    ├── plop
    │   └─ data
    ├── tata
    └─ titi
```

2 directories, 3 files

# Commandes sur Répertoires

Répertoire	Description
<b>cd</b>	Changer de répertoire
<b>mkdir</b>	Créer un répertoire
<b>rm</b>	Supprimer un fichier ou dossier
<b>find</b>	Trouver des fichiers
<b>ls</b>	Lister les fichiers
<b>pwd</b>	Afficher le répertoire courant
<b>du</b>	Afficher la taille d'un répertoire
<b>df</b>	Afficher l'espace libre sur les points de montage
<b>grep</b>	Chercher à l'intérieur des fichiers
<b>cat</b>	Afficher le contenu d'un fichier sur la sortie standard

Notion de chemin absolu et relatif:

- un chemin absolu commence par la racine /
- Un chemin relatif l'est par rapport au répertoire courant (../lib, ./lib/)

# Commandes de base

- **pwd** affiche le dossier/répertoire courant
- **ls** liste les fichiers
- **cd** change de répertoire courant (*change directory*)
- **mkdir** crée un dossier
- **rmdir** supprime un dossier vide
- **rm** supprime un dossier (--recursive) ou un fichier
- **cd ..** (remonte d'un dossier)
- **touch x** créer le fichier x, vide

# Le manuel

- La commande “man”
  - man ls
  - man pwd
  - man man
  - man 7 signal
  - ...
- Sous linux pour avoir les pages de manuel en français:
  - manpages-fr
  - manpages-fr-dev
  - manpages-fr-extra

```
MAN(1)                                Manual pager utils                                MAN(1)

NAME
  man – an interface to the system reference manuals

SYNOPSIS
  man [man options] [[section] page ...]...
  man -k [apropos options] regexp ...
  man -K [man options] [section] term ...
  man -f [whatis options] page ...
  man -l [man options] file ...
  man -w [-W [man options] page ...]

DESCRIPTION
  man is the system's manual pager. Each page argument given to man is normally the name of a program, utility or function. The manual page associated with each of these arguments is then found and displayed. A section, if provided, will direct man to look only in that section of the manual. The default action is to search in all of the available sections following a pre-defined order (see DEFAULTS), and to show only the first page found, even if page exists in several sections.

  The table below shows the section numbers of the manual followed by the types of pages they contain.

  1   Executable programs or shell commands
  2   System calls (functions provided by the kernel)
  3   Library calls (functions within program libraries)
  4   Special files (usually found in /dev)
  5   File formats and conventions, e.g. /etc/passwd
  6   Games
  7   Miscellaneous (including macro packages and conventions), e.g. man(7), groff(7), man-pages(7)
  8   System administration commands (usually only for root)
  9   Kernel routines [ Non standard ]

  A manual page consists of several sections.

  Conventional section names include NAME, SYNOPSIS, CONFIGURATION, DESCRIPTION, OPTIONS, EXIT STATUS, RETURN VALUE, ERRORS, ENVIRONMENT, FILES, VERSIONS, CONFORMING TO, NOTES, BUGS, EXAMPLE, AUTHORS, and SEE ALSO.
```

```
LS(1)                                Commandes de l'utilisateur                                LS(1)

NOM
  ls - Afficher le contenu de répertoires

SYNOPSIS
  ls [OPTION]... [FICHIER]...

DESCRIPTION
  Afficher les informations des FICHIERs (du répertoire courant par défaut). Les entrées sont triées alphabétiquement si aucune des options -ctuvSUX ou --sort n'est indiquée.

  Les paramètres obligatoires pour les options de forme longue le sont aussi pour les options de forme courte.

  -a, --all
      inclure les entrées débutant par « . »

  -A, --almost-all
      omettre les fichiers « . » et « .. »

  --author
      avec -l, afficher l'auteur de chaque fichier

  -b, --escape
      afficher les caractères non graphiques sous la forme de caractères d'échappement de style C

  --block-size=TAILLE
      avec -l, ajuster les tailles avec TAILLE quand elles sont affichées ; par exemple « --block-size=M » ; voir le format de TAILLE ci-dessous

  -B, --ignore-backups
      ignorer les fichiers commençant par « . » et « .. »

Manual page ls(1) line 1 (press h for help or q to quit)
```



## Stopper un programme...

**CTRL + C** → **SIGINT**

**CTRL + Z** → **SIGTSTP** (*fg(1)/bg(1)*)

**CTRL + \** → **SIGQUIT** (⚠)

Voir *signal(7)*

**CTRL + S/Q** → **TTY XOFF/XON**

## Éditer la ligne de commande (readline)

CTRL + A/E → Curseur au début/fin de ligne

CTRL + W → Couper le dernier mot avant curseur

CTRL + U/K → Couper du curseur au début/fin de ligne

CTRL + Y → Coller

CTRL + L → *clear* l'écran

CTRL + R → Chercher un commande dans *history*

ALT + . → Reprendre le dernier argument

# La Variable PS1

- Permet de configurer l'apparence du prompt
- A une syntaxe souvent spécifique au shell pour les placeholders
- Peut prendre de la couleur
- Utilisez les générateurs sur internet pour gagner du temps (googlez PS1 generator + [Nom du shell])

La variable PS1 dans Bash est utilisée pour définir le prompt de commande. Voici un exemple simple :

```
export PS1="\u@\h:\w$ "
```

Cela affichera sous la forme :

```
[nom d'utilisateur]@[nom d'hôte]:[répertoire courant] $
```

En zsh:

```
export PS1="%n%@%m %~ $ "
```

# Les Alias

Les alias sont des raccourcis pour les commandes dans votre terminal qui peuvent être très utiles lorsque vous souhaitez exécuter une série de commandes fréquemment.

La syntaxe est:

```
alias ALIAS='CMD ARGS'
```

Voici quelques exemples :

- `alias ll='ls -lh' # Affiche les fichiers et répertoires en format long avec des tailles lisibles par l'homme`
- `alias gs='git status' # Vérifie le statut de votre dépôt git`
- `alias dc='docker-compose' # Un outil permettant de définir et d'exécuter des applications Docker multi-conteneurs.`

# La configuration du shell

- Les shell ont tous un fichier de configuration dans lequel mettre les commandes à lancer au démarrage
  - bash: ~/.bashrc
  - zsh: ~/.zshrc
  - ...
- C'est ici que l'on peut persister la configuration du shell entre les démarrages
- C'est un simple fichier lancé au début de chaque session de shell
  - Pour prendre en compte les changements en direct :
    - `$ . ~/.<shell>rc`

# Automatisation dans le shell

- Le shell est aussi un langage de script
  - Contrôle de flot (boucles, tests)
  - Définition de fonctions
  - Entrée / Sorties
- Il est possible de définir des fonctions dans votre ~/.bashrc

# Shell scripting

Les boucles sont également utiles pour automatiser des tâches répétitives :  
Boucler sur les fichiers d'un répertoire et afficher leur contenu

```
for file in /path/to/directory/*  
do  
    echo "$file:" ; cat "$file"  
done
```

Ce script parcourt tous les fichiers dans un répertoire donné, puis affiche leur contenu. Votre `/path/to/directory` sera remplacé par votre propre chemin d'accès au répertoire.

# Fonctions shell

Les fonctions sont aussi utilisables dans votre `.bashrc` et peuvent être très pratiques lorsque vous souhaitez encapsuler une série d'instructions qui doivent être exécutées régulièrement.

Voici un exemple :

Fonction pour créer et se déplacer dans un nouveau répertoire

```
mkcdir () {  
    mkdir -p "$1" && cd "$1"  
}
```

Cette fonction prend en argument le nom du dossier à créer, puis se déplace dedans. Elle peut être appelée avec la commande *mkcdir mon\_dossier* par exemple.



# Commandes alternatives



Certaines commandes ont des alternatives plus modernes:

- `ls => lsd`
- `cat => bat`
- `grep => rg`

Liste d'utilitaires CLI: <https://github.com/agarrharr/awesome-cli-apps>

# Les éditeurs

## En mode console (*TUI*)

- **Nano** : Un éditeur léger et facile à utiliser qui est fourni par défaut dans la plupart des distributions.
- **Vim** (Vi IMproved) : Un éditeur modal qui offre une grande flexibilité et une puissance dans l'édition du texte,  barrière d'apprentissage
  - Voir **Neovim**
- **Emacs** : Un éditeur non-modal à base de raccourcis (*chords*),  à configurer
  - Voir configs modales type **Evil**

± compatibles syntaxe readline

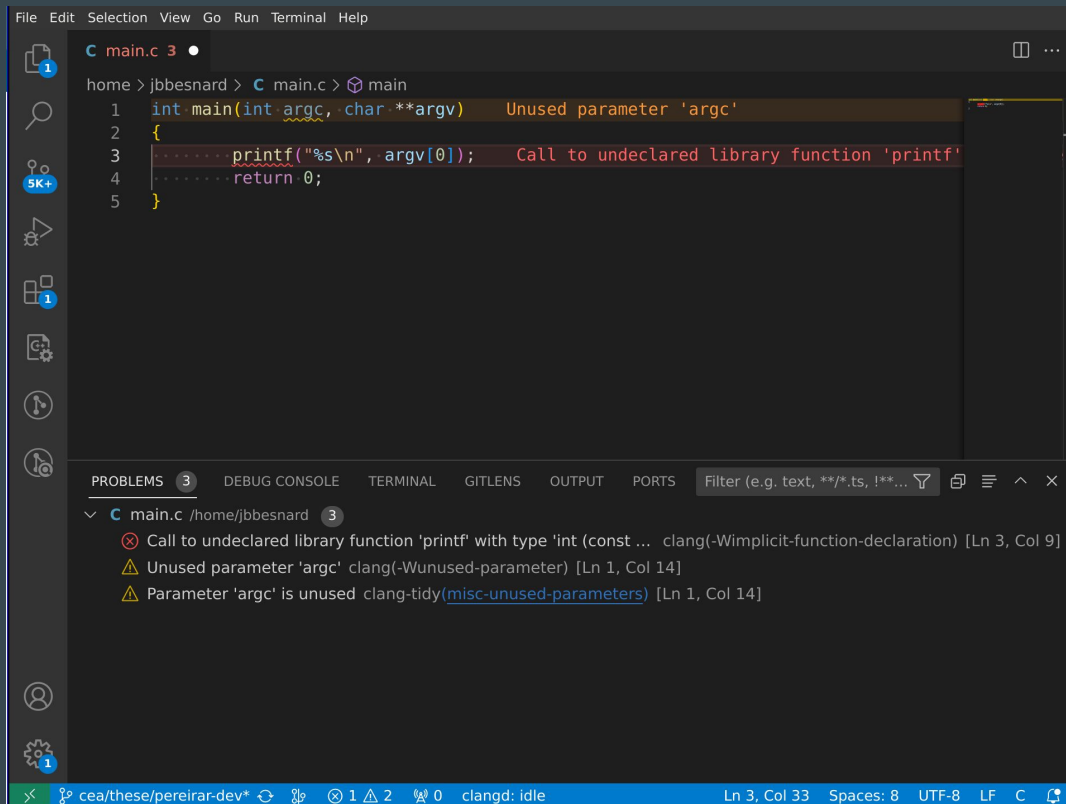
Et plein d'autres ...

## En mode graphique (*GUI*)

- Visual Studio Code (**VSCode**) : Un éditeur de code open source, gratuit et très populaire avec une grande variété d'extensions pour les langages de programmation.
- **Eclipse** : Une plateforme de développement intégrée qui permet la création, le débogage et la gestion des applications logicielles.
- **IntelliJ IDEA** : Un environnement de développement pour les langages Java, mais aussi utilisable par d'autres technologies comme Python, Ruby, etc.
- **Kate** : Une alternative à Gedit et Pluma qui est fourni par défaut dans la plupart des distributions KDE.
- **Geany** : Un éditeur de texte léger et rapide avec une interface utilisateur simple.

**Et plein d'autres .... dont Vim et Emacs**

# VSCode

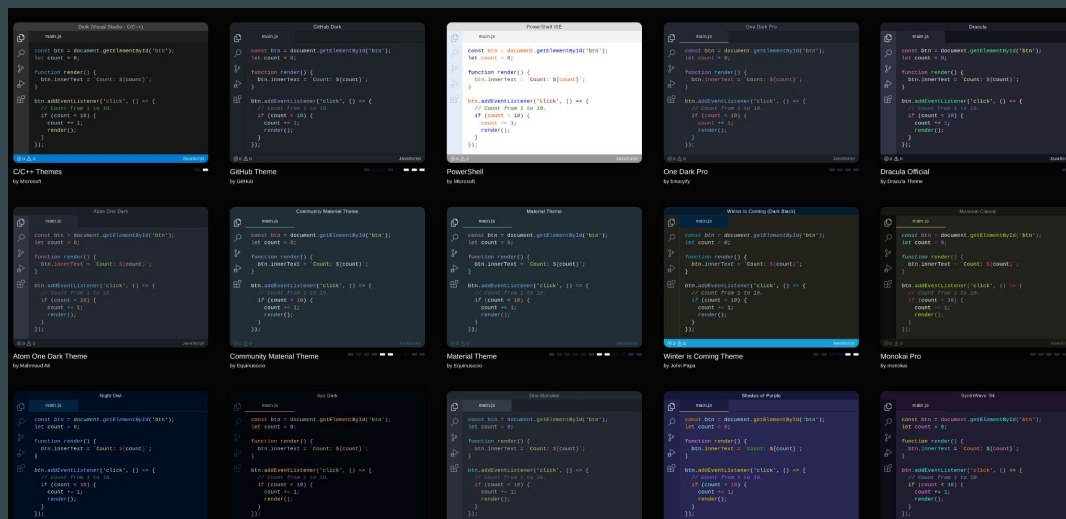


# VSCode: Installation

- Il est possible d'essayer vscode en ligne : <https://vscode.dev/>
- <https://code.visualstudio.com/Download> pour télécharger
  - Version paquets
  - Version binaires
- Windows / Linux / MacOS

# VSCode: Choix du thème

- <https://vscothemes.com/> pour chercher
- CTRL + K + T (ou via menu pour changer)
- Choix entre sombre et clair (question de goût)



# VSCode: Plugins recommandés

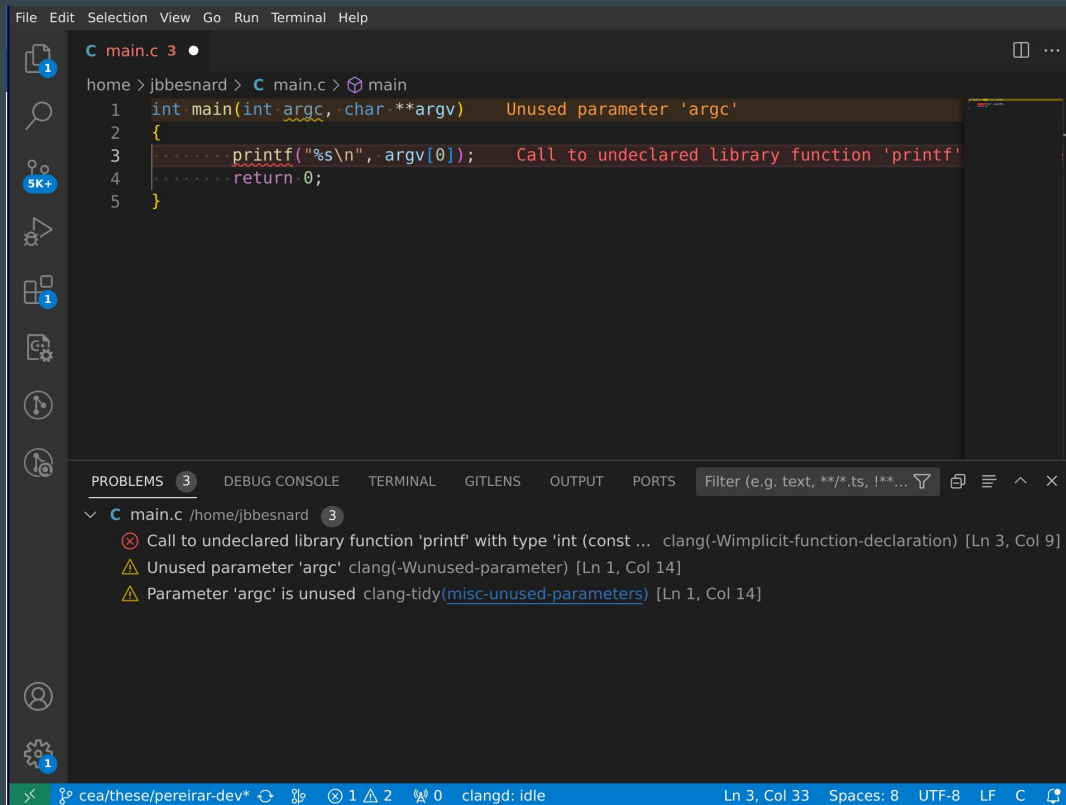
- **clangd** : analyseur de code de clang (voir plus loin le linting)
- **errorlens** : affichage des erreurs inline
- **shellcheck** : linter pour le shell
- **Python** : outils et support pour Python
- ...



# VSCode: Commandes de base

- CTRL + P : inspecteur de fichiers
- F1 : palette de commande
- CTRL + T : inspecteur de symboles
- CTRL + CLICK : naviguer au symbole
- Pensez à bind une touche pour retourner au contexte précédent/suivant
  - `workbench.action.navigateBack`
  - `workbench.action.navigateForward`

# Demo VSCode

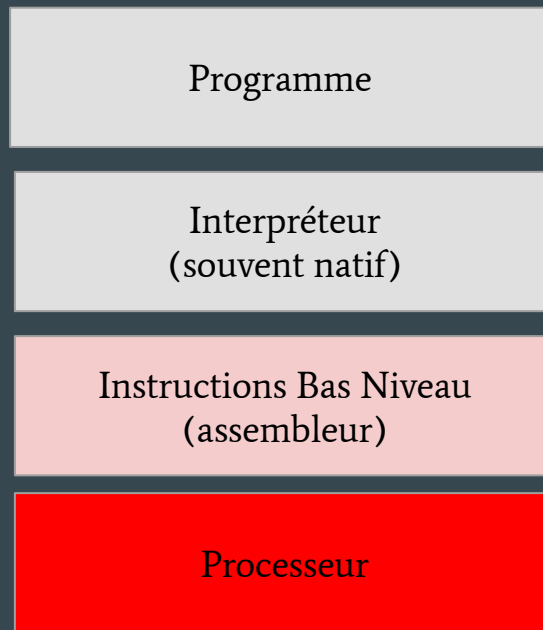


# Compiler un programme

# Langages compilés vs langages interprétés

- Langages interprétés
  - BASIC, Shell, Python, Ruby, Scilab, ...
  - Exécutés avec le code sans traduction
  - Souvent plus faciles à apprendre
- Langage compilés
  - Fortran, C, C++, Rust, Zig, ...
  - On exécute un binaire compilé
  - Souvent plus performants
- Contre-exemples
  - Compilation intermédiaire (*bytecode*, *JIT*)
  - Cython
  - Go run, TCC

# Langage interprété



# Langage compilé

Programme

Compilateur

Binaire (Natif)

Instructions Bas Niveau  
(assembleur)

Processeur



# Langages et compilateurs

Il existe de nombreux langages et compilateurs mais il y en a deux principaux, GCC et LLVM.

- C :
  - clang (LLVM)
  - gcc (GNU C)
- C++ :
  - clang++ (LLVM)
  - g++ (GNU C)

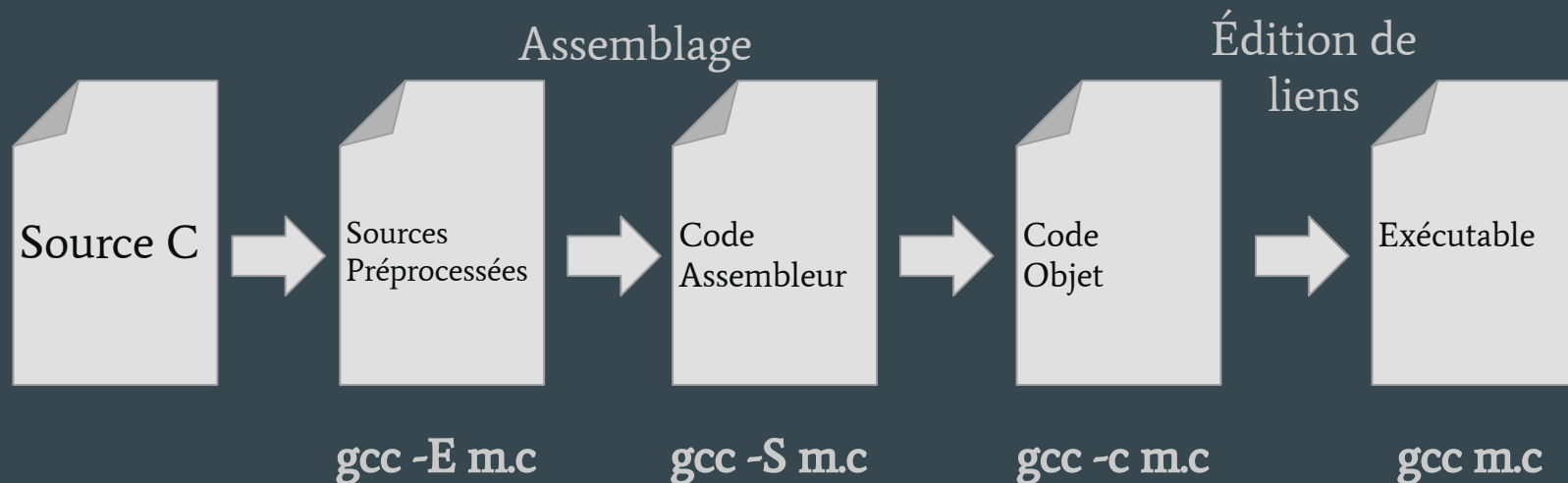
# Langages et compilateurs

GCC (GNU Compiler Collection) et LLVM (Low Level Virtual Machine) sont deux ensembles de compilateurs largement utilisés dans le développement logiciel :

- GCC :
  - Développé par le projet GNU.
  - Supporte de nombreux langages comme C, C++, Objective-C, etc.
  - Robuste, stable et compatible avec de nombreuses plateformes.
  - Distribué sous licence libre (GPL).
- LLVM :
  - Ensemble de compilateurs modulaires et infrastructure de développement.
  - Conçu pour être modulaire et flexible.
  - Utilisé comme base pour créer des compilateurs pour différents langages (par exemple Rust).
  - Architecture orientée performance et optimisation du code.
  - Distribué sous licence libre (UIUC).



# Phases de la compilation



Binaires Elf (sous Linux)

# Construire un programme (aspects bas niveau)

# Structure d'un programme C

## main.c

```
#include <stdio.h>

#include "utils.h"

// Prototype de la fonction Fibonacci définie dans
utils.c
int fibonacci(int n);

int main(void)
{
    int n = 10; // Exemple : Calculer le 10ème terme
de la suite de Fibonacci
    int result = fibonacci(n);
    printf("Le %dème terme de la suite de Fibonacci
est : %d\n", n, result);
    return 0;
}
```

## utils.h

```
#ifndef UTILS_H
#define UTILS_H

int fibonacci(int n);

#endif /* UTILS_H */
```

## utils.c

```
#include "utils.h"

// Définition de la fonction Fibonacci
int fibonacci(int n)
{
    if (n <= 1)
        return n;
    else
        return fibonacci(n-1) + fibonacci(n-2);
}
```

# Toujours éviter la double inclusion

- Lorsque vous incluez un fichier d'en-tête dans plusieurs fichiers source, il est possible que ce fichier d'en-tête soit inclus plusieurs fois dans un même fichier source. Cela peut entraîner des erreurs de compilation dues à des redéfinitions ou des conflits de symboles.
- Les directives `#ifndef`, `#define`, `#endif` sont utilisées pour envelopper tout le contenu du fichier d'en-tête. Ainsi, lorsque le fichier d'en-tête est inclus plusieurs fois dans un même fichier source, la première inclusion définira le symbole `UTILS_H`, et les suivantes seront ignorées grâce à la garde `#ifndef`.

utils.h

```
#ifndef UTILS_H
#define UTILS_H

int fibonacci(int n);

#endif /* UTILS_H */
```

# Compilation en un seul binaire

# On Compile

```
gcc ./main.c ./utils.c -o fibo  
    [ Fichiers ... .. ]      [ Nom du binaire]
```

# On Lance !

```
./fibo
```

# Notion de bibliothèque dynamique

- Principe général :
  - Une bibliothèque dynamique, aussi connue sous le nom de DLL (Dynamic Link Library) sous Windows ou de fichier .so (Shared Object) sous Unix/Linux, est un ensemble de fonctions et de routines précompilées.
  - Contrairement aux bibliothèques statiques, les bibliothèques dynamiques sont chargées en mémoire au moment de l'exécution du programme.
- Avantages des bibliothèques dynamiques :
  - Economie d'espace : Elles permettent de partager le code entre plusieurs programmes, réduisant ainsi la taille totale des programmes.
  - Mises à jour facilitées : Les mises à jour de la bibliothèque peuvent être effectuées indépendamment des programmes qui l'utilisent.
  - Chargement à la demande : Les bibliothèques dynamiques ne sont chargées en mémoire que lorsque nécessaire, ce qui peut réduire le temps de démarrage et économiser de la mémoire.

# Notion de bibliothèque dynamique

- Principe général :
  - Une bibliothèque dynamique, aussi connue sous le nom de DLL (Dynamic Link Library) sous Windows ou de fichier .so (Shared Object) sous Unix/Linux, est un ensemble de fonctions et de routines précompilées.
  - Contrairement aux bibliothèques statiques, les bibliothèques dynamiques sont chargées en mémoire au moment de l'exécution du programme.
- Avantages des bibliothèques dynamiques :
  - Economie d'espace : Elles permettent de partager le code entre plusieurs programmes, réduisant ainsi la taille totale des programmes.
  - Mises à jour facilitées : Les mises à jour de la bibliothèque peuvent être effectuées indépendamment des programmes qui l'utilisent.
  - Chargement à la demande : Les bibliothèques dynamiques ne sont chargées en mémoire que lorsque nécessaire, ce qui peut réduire le temps de démarrage et économiser de la mémoire.
- Faiblesses des bibliothèques dynamiques :
  - Résolution d'une partie du code au runtime (incompatibilité, instabilité)
  - Nécessité de maintenir les search path (LD\_LIBRARY\_PATH) (environnement plus complexe)
  - Surcoût lors du premier appel (passage par la Procédure Linkage Table PLT)

# Notion de Bibliothèque Dynamique

Compilation :

toto.h

libtoto.so

myprog.c

---

Exécution :

myprog

libtoto.so



# Compilation en un seul binaire

# On Compile

```
gcc ./main.c ./utils.c -o fibo
    [ Fichiers ... .. ]      [ Nom du binaire ]
```

# On Lance !

```
./fibo
```

# On regarde les bibliothèques de notre fibo (nous sommes déjà link à des libs) :

```
$ ldd fibo
linux-vdso.so.1 (0x00007ffc6d620000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fa0090c6000)
/lib64/ld-linux-x86-64.so.2 (0x00007fa0092db000)
```

# Construisons libutils.so

# On Compile

```
gcc -shared -fPIC utils.c -o libutils.so
```

- `-shared` :
  - L'option `-shared` indique à GCC de produire une bibliothèque partagée (.so) plutôt qu'un exécutable.
- `-fPIC` (Position Independent Code) :
  - L'option `-fPIC` indique à GCC de générer du code indépendant de la position, ce qui est requis pour créer des bibliothèques partagées. Le code indépendant de la position peut être chargé à n'importe quelle adresse en mémoire sans nécessiter de réadressage, ce qui est essentiel pour permettre le partage des bibliothèques entre plusieurs processus en mémoire.

# Construisons Fibo en nous “linkant” à libutils.so

# On Compile

```
gcc main.c -o main -L. -lutils
```

- `gcc` : C'est le compilateur C GNU
- `main.c` : le fichier source contenant le code principal du programme.
- `-o main` : spécifie le nom du fichier exécutable généré. Dans ce cas, l'exécutable sera nommé « main ».
- `-L.` : indique à GCC de rechercher les bibliothèques partagées dans le répertoire courant « . ».
- `-lutils` : Cet argument indique à GCC de lier le programme avec une bibliothèque appelée libutils.so. (Le préfixe « lib » et le suffixe « .so » sont automatiquement ajoutés.)

# Tentons de lancer ...

```
$ ./main
./main: error while loading shared libraries: libutils.so: cannot
open shared object file: No such file or directory
```

Le binaire (au runtime ne trouve pas libutils.so)

```
$ ldd main
    linux-vdso.so.1 (0x00007ffc101fb000)
    libutils.so => not found
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6
(0x00007fd347b9d000)
    /lib64/ld-linux-x86-64.so.2 (0x00007fd347db2000)
```

# Comment le binaire cherche ses bibliothèques

Au démarrage du programme, le **loader** doit charger les bibliothèques.

Il tente donc de les chercher aux endroits suivants :

- **rpath** : L'option **rpath** permet de spécifier un chemin de recherche pour les bibliothèques partagées au moment de l'exécution lors de la compilation.
- **LD\_LIBRARY\_PATH**: chemins de recherche dans une variable d'environnement
- Fichiers de configuration ld : le système recherche les chemins spécifiés dans les fichiers de configuration ld, tels que :
  - /usr/local/cuda/targets/x86\_64-linux/lib
  - /usr/local/cuda-12/targets/x86\_64-linux/lib
  - /usr/local/cuda-11.0/targets/x86\_64-linux/lib
  - ...
- Chemins standards : Si la bibliothèque n'est pas trouvée dans les étapes précédentes, le système recherche dans les chemins standard, tels que :
  - /lib
  - /lib64
  - /usr/lib
  - /usr/lib64

# Tentons de lancer ... avec LD\_LIBRARY\_PATH

```
$ LD_LIBRARY_PATH=$PWD:$LD_LIBRARY_PATH ./main  
Le 10ème terme de la suite de Fibonacci est : 55
```

```
$ LD_LIBRARY_PATH=$PWD:$LD_LIBRARY_PATH ldd main  
linux-vdso.so.1 (0x00007ffd33ffe000)  
libutils.so => /home/jbbesnard/repo/TOOI/fibonacci/libutils.so  
(0x00007fe451578000)  
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fe45136a000)  
/lib64/ld-linux-x86-64.so.2 (0x00007fe451584000)
```

# Tentons de lancer ... recompilons avec un -rpath

```
$ gcc main.c -L. -lutils -Wl,-rpath=$PWD -o main
```

- -Wl : C'est une option utilisée avec GCC pour transmettre des options directement à l'éditeur de liens (linker). L'option -Wl permet de passer des options spécifiques à l'éditeur de liens pendant la phase de compilation.
  - -rpath=\$PWD : L'option -rpath spécifie un chemin de recherche pour les bibliothèques partagées.
  - \$PWD est une variable d'environnement qui représente le répertoire de travail actuel (le répertoire où se trouve le fichier exécutable en cours de compilation). En utilisant \$PWD, on spécifie le chemin de recherche comme étant le répertoire de travail actuel au moment de la compilation.

# Tentons de lancer ... recompilons avec un -rpath

Maintenant le binaire contient le RPATH, plus besoin d'altérer l'environnement.

```
$ ./main
```

```
Le 10ème terme de la suite de Fibonacci est : 55
```

```
$ ldd main
```

```
linux-vdso.so.1 (0x00007ffd716db000)
```

```
libutils.so => /home/jbbesnard/repo/TOOI/fibonaci/libutils.so
```

```
(0x00007f8fe83c0000)
```

```
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f8fe81b2000)
```

```
/lib64/ld-linux-x86-64.so.2 (0x00007f8fe83cc000)
```

```
$ objdump -x ./main | grep RUNPATH
```

```
RUNPATH /home/jbbesnard/repo/TOOI/fibonaci
```



# Binaire statique (tout à l'intérieur)

# On compile

```
gcc ./main.c ./utils.c -o fibo -static
```

# On lance !

```
./fibo
```

# Fibo est maintenant un binaire statique :

```
$ ldd fibo  
n'est pas un exécutable dynamique
```

# Bibliothèques statiques

Il est aussi possible de construire des bibliothèques statiques (extension .a) ces bibliothèques sont des archives de .o et elles peuvent être passées lors de la compilation d'un programme pour une inclusion directe.

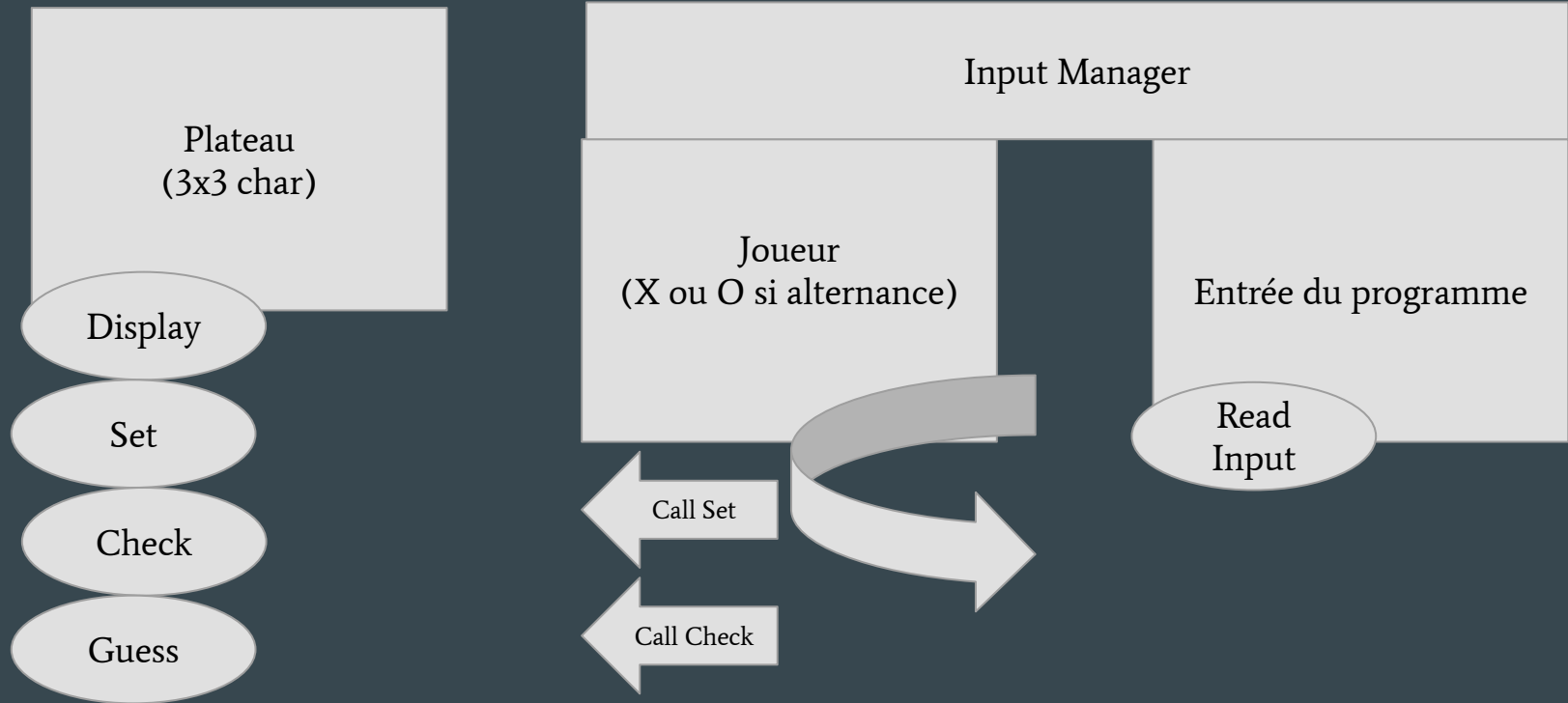
- Compilation des fichiers source : Compilez les fichiers source en utilisant le compilateur C de votre choix (par exemple, GCC). Assurez-vous de compiler avec l'option `-c` pour produire des fichiers d'objets (extension .o) plutôt que des exécutables. Par exemple :
  - `gcc -c fichier1.c fichier2.c ...`
- Archivage des fichiers objets : Utilisez l'utilitaire d'archivage (souvent `ar` sous Unix/Linux) pour créer une archive contenant les fichiers objets. Par exemple :
  - `ar rcs libmabibliotheque.a fichier1.o fichier2.o ...`
    - `r` : Remplace ou ajoute des fichiers à l'archive.
    - `c` : Crée l'archive si elle n'existe pas.
    - `s` : Met à jour l'index de l'archive.
- Pour utiliser la bibliothèque statique `libmabibliotheque.a` dans un programme, vous pouvez compiler comme suit :
  - `gcc main.c -o main -L. -lmabibliotheque`
    - `-L.` spécifie au compilateur de rechercher la bibliothèque dans le répertoire courant.
    - `-lmabibliotheque` indique au compilateur de lier le programme avec la bibliothèque `libmabibliotheque.a`.

# De l'importance de la modularité

# La programmation est affaire d'abstraction

- Ne répétez jamais le code (*DRY*) : créez des fonctions
- Si un code doit être partagé par plusieurs programmes créez des bibliothèques
- Dans vos programmes :
  - Effectuez un découpage fonctionnel
  - Identifier les scénarios d'usage
  - Construisez un schéma bloc correspondant
  - A partir de chaque bloc créez un fichier (.c .h) avec ses interfaces
  - Adopter les principes d'encapsulation et d'abstraction des langages orientés objets (même si vous êtes dans un langage qui n'a pas ce support)
  - Ne modifiez jamais un état interne à un objet hors d'une de ses méthodes
  - Évitez les variables globales
  - Écrivez du code lisible et ne rognez pas sur les noms de variables

# Petit diagramme d'un morpion



# Passons à un diagramme UML

Exemple d'éditeur en Ligne: <https://mermaid.live>

Lien vers diagramme [ICI](#).

Ceci donne trois fichiers:

- main.c :
  - Analyse des arguments
  - instantiation de InputManager
  - Appel de mainloop
- input.c :
  - Implémentation logique de jeu (1 et 2 joueurs)
- morpion.c :
  - Gestion de la grille
  - Calculs sur grille



**Implémentation pendant le TP**

# Automatiser le Build, le Makefile



# Principe d'un Makefile

- Fichier de configuration de l'outil Make
- Facilite la compilation et le link de programmes
- Le nom du fichier est toujours "Makefile" (sauf explicite)
- Vue du shell, on compile avec la commande make dans le répertoire où se trouve le Makefile. Cela uniformise la compilation de tout programme (et donc la nôtre et la vôtre !)
- Reproductibilité de compilation (il est facile d'oublier un flag de compilation)
- Principaux arguments à la commande make :
  - -f myMakefile : changer le nom du fichier
  - -C <chemin\_du\_projet> : pour compiler dans un autre répertoire

Tout argument non-Make est ensuite utilisé comme nom de cible (voir slide suivant)

# Principe d'un Makefile

- Une cible définit un fichier à construire ou une action
- Une règle est l'ensemble des commandes à exécuter pour réaliser cette cible (lancées dans un shell différent).  
Chaque règle commence par une tabulation
- La première cible est celle exécutée par défaut
- Une cible peut avoir des dépendances : des cibles à résoudre avant.
- La résolution de dépendances fonctionne par horodatage. Permet par exemple de recompiler les .o pour lesquels le fichier .c a été modifié en amont

<code>\$@</code>	Nom de la cible
<code>\$&lt;</code>	Nom de la 1ere dépendance
<code>\$\$</code>	Nom de toutes les dépendances
<code>\$?</code>	Nom des dépendances plus récentes que la cible
<code>\$*</code>	Nom du fichier sans suffixe (voir <b>.SUFFIXES</b> )

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

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

```
main.bin: main.o
    gcc -o main.bin main.o
```

```
main.o: main.c
    gcc -c $$< -o $$@
```

```
main.bin: main.o
    gcc -o $$@ $$<
```

```
%.o: %.c
    gcc -c $$< -o $$@
```

```
main.bin: main.o
    gcc -o $$@ $$<
```

```
$ make main.bin
```

# Makefile de base

```
CC=gcc  
CFLAGS=-O3
```

```
.PHONY: all clean
```

```
all: main
```

```
main: main.c
```

```
|——> $(CC) $(CFLAGS) $^ -o $@
```

```
clean:
```

```
|——> rm -f main
```

TOUJOURS INDENTER  
LE MAKEFILE AVEC  
DES TABULATIONS.