# Développement natif - Démonstration 1 : développer un programme *natif* en C, fiche de suivi

Dans cette démonstration, nous allons compiler du code source C vers un binaire executable par l'OS. Nous allons voir en détail les différentes étapes de la compilation.



## Objectifs

- Comprendre le processus de compilation
- Comprendre que la compilation vise une plateforme spécifique

## Pré-requis

### Sous Gnu/Linux (Debian, Ubuntu)

Installer le compilateur gcc

```
sudo apt install gcc
```

### Sous Windows

Installer WSL 2 pour accéder à un environnement GNU/Linux (Ubuntu par défaut) sur votre machine.

Pour télécharger et installer WSL 2, vous pouvez vous rendre sur la page officielle de Microsoft et suivre les instructions.

Ouvrez une Invite de commandes en mode administrateur en cliquant avec le bouton droit et en sélectionnant `"Exécuter en tant qu'administrateur"`

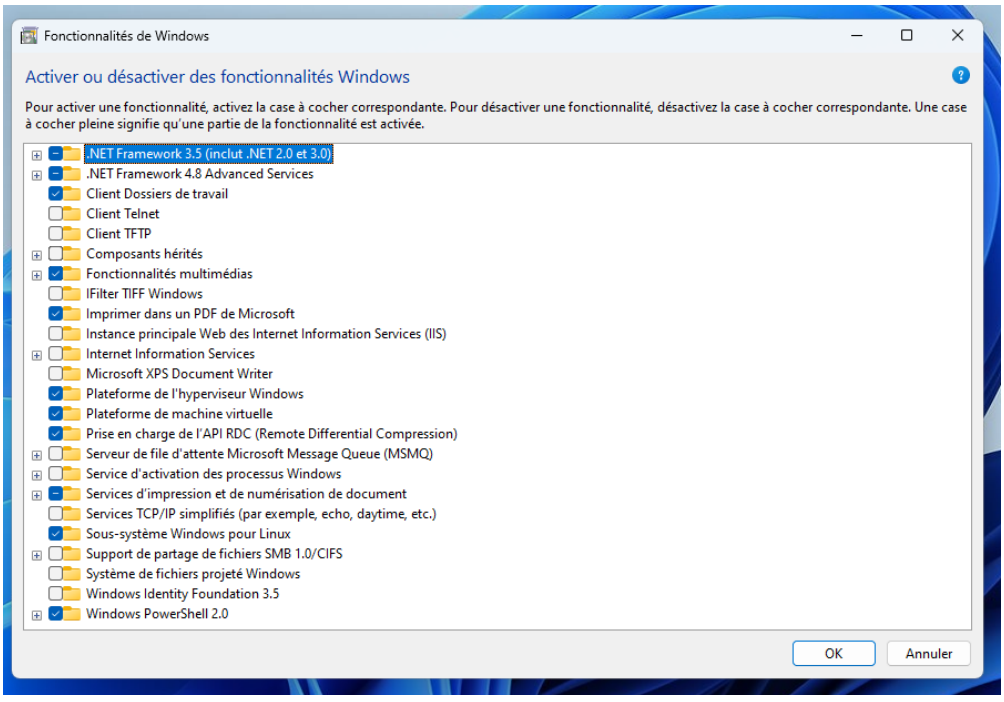
```
wsl --install
wsl --update
```

# Erreurs rencontrées et configuration de Windows

En cas de problèmes avec wsl, veuillez consulter cette page de problèmes connus avant de demander de l'aide. Vous pouvez également consulter la FAQ pour en savoir plus.

Pour exécuter WSL 2, vous devez au préalable *activer la plateforme de machine virtuelle Windows* (appelée Hyper-V). Si ce n'est pas le cas, vous allez obtenir un message comme celui-ci `"Please enable the virtual Machine Platform WIndows feature and ensure virtualization is enabled in the BIOS"`.

Pour activer la plateforme de virtualisation, redémarrer votre machine, appuyez sur `F2` ou la touche indiquée indiquée par votre carte-mère pour accéder au BIOS. Vous devez également vous assurer que les fonctionnalités Plateforme de l'hyperviseur Windows et Plateforme de machine virtuelle sont bien activées. Tapez `"fonctionnalités windows"` dans votre barre de recherche, puis activez-les. Redémarrez votre machine.




Vérifier l'installation

```
systeminfo
```

La dernière entrée du rapport généré vous indiquera la configuration requise pour Hyper-V. Vérifier que tous les prérequis sont bien remplis.

Pour lister les distributions installées

```
wsl -l -v
```

Ouvrir la WSL. Choisissez un nom d'utilisateur et un mot de passe pour le système GNU/Linux. Cet utilisateur est l'administrateur du système avec la capacité d'exécuter des commandes d'administration (`sudo`). WSL va vous ouvrir shell sur votre instance GNU/Linux. Executer les commandes suivantes :

```
#Mettre à jour la liste des paquets
sudo apt update
#Mettre à jour les paquets
sudo apt upgrade
#Installer gcc
sudo apt install gcc
# Tester
gcc --version
```

Vous avez à présent accès à une distribution GNU/Linux via la WSL et installé le compilateur `gcc`.

# Compiler du code source C vers du langage machine (binaire)

**Créer un dossier** `demo-compilation`.

**Créer un fichier source** `main.c`.

```
#include<stdio.h>

int main(){
    printf("Hello world !");
    return 0;
}
```

**Construire l'executable** à partir du code source pour la plateforme GNU/Linux en passant vers les différentes étapes de la "compilation" (*build*) :

1. Compilation
2. Assemblage
3. Linkage

> Regardez les différentes options de `gcc` avec l'option `--help`. Par défaut, gcc effectue toutes les étapes de la compilation en une fois. `gcc main.c` compile, assemble et link pour produire l'executable `a.out`

### Compilation

```
gcc -S main.c
```

Cela crée un fichier en langage assembleur `main.s`

> Inspecter le fichier assembleur `main.s`

### Assemblage

```
gcc -c main.s
```

Cela crée un fichier objet (executable au format ELF) `main.o`.

**Inspecter** le contenu du fichier `main.o` (`cat main.o`). Qu'est ce qui s'affiche ? Pourquoi ?

> Sous le capot, `gcc` utilise l'assembleur `as`

### Linkage

**Linker** pour créer le programme (link vers l'implémentation de `printf` de la librairie standard)

```
gcc main.o -o say-hi
```

> L'option `-o` permet de contrôler le nom de fichier de sortie
>
> Sous le capot, `gcc` utilise le linker `ld`

**Executer** le binaire sur votre OS, via le shell

```
./say-hi
```

> Si on essaie d'exécuter le programme `say-hi` sur Windows, cela ne fonctionnera pas car l'executable généré est spécifique à l'OS GNU/Linux (format ELF) ! Windows et Linux ne manipulent pas les mêmes formats de binaire. Également, lors de l'appel à `printf` (*dynamic linking*), il sera impossible de trouver (path) le binaire correspondant sur le système.

`say-hi` est un programme *natif* à la plateforme GNU/Linux. Il a été *compilé pour cette plateforme* uniquement.

## Bonus : scripter le processus de compilation avec `make`

`make` est un programme qui permet de maintenir des programmes. Il permet d'automatiser la compilation de programmes à partir des fichiers sources. `make` fonctionne sur la base de *règles* à écrire.

Essayez `man make`. Lisez

**Créer** un fichier `Makefile`. Voici le template d'une règle `make` :

```
cible: dependance1 dependance2
    commande 1
    commande 2
```

où `dependance1` et `dependance2` sont d'autres cibles dont `cible` dépend. Ces règles seront donc exécutées par `make` en amont.

Pour executer une règle, dans le terminal

```
make cible
```

Par défaut, `make` execute la première règle si aucune règle n'est spécifiée.

1. **Écrire** un `Makefile` qui permet de réaliser chaque étape du *build* (compilation, assemblage et linkage) *indépendamment*. Chaque règle doit pouvoir être exécutée directement. Par exemple, on doit pouvoir procéder au linkage sans *explicitement* passer par les phases de compilation et d'assemblage.

`make` permet de déclarer des variables sous forme de clef/valeur. Voici la syntaxe :

```
VARIABLE=VALEUR
```

Pour déférencer cette variable (extraire sa valeur) dans le `Makefile`

```
#Ceci est un commentaire
#Afficher sur la sortie standard le contenu de la variable VARIABLE
@echo $(VARIABLE)
```

2. **Déclarer** une variable qui contient le nom du binaire à produire (`say-hi`). **Mettre à jour** le `Makefile` en conséquence. On souhaite que l'instruction `make` fabrique le binaire `say-hi` et affiche à la fin `"Le programme say-hi a été compilé avec succès !"`

   Indice : pour afficher un message sur la sortie standard, utiliser la commande `@echo "Mon message"`.

# Conclusion de cette démo

- Ce qu'on appelle *compilation* de manière abusive comprend en fait plusieurs étapes : compilation, assemblage, linkage
- Chaque OS (ou langage) fournit dans son SDK des libraires utilisables pour le développement (`stdlib.h` fait partie du SDK du langage C)
- Un *programme natif* est un programme compilé *vers une plateforme cible* (ici via `gcc`). Il est natif *à la plateforme*. Ici, il est exécuté *directement* par l'OS (code machine ou binaire)

- Il existe des outils comme `make` pour automatiser les processus liées à la compilation, notamment dans le cas de projets réels ou le nombre de fichiers sources et de libraires est important