

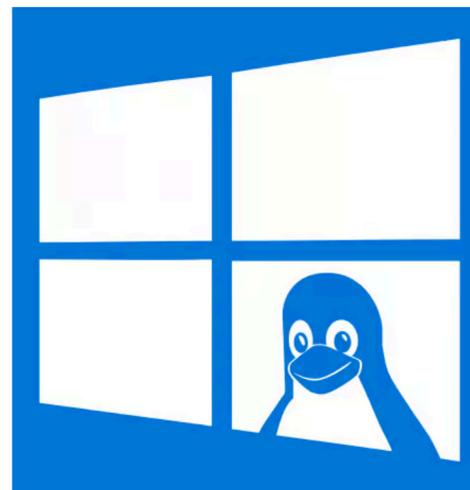
Rappels

et plus



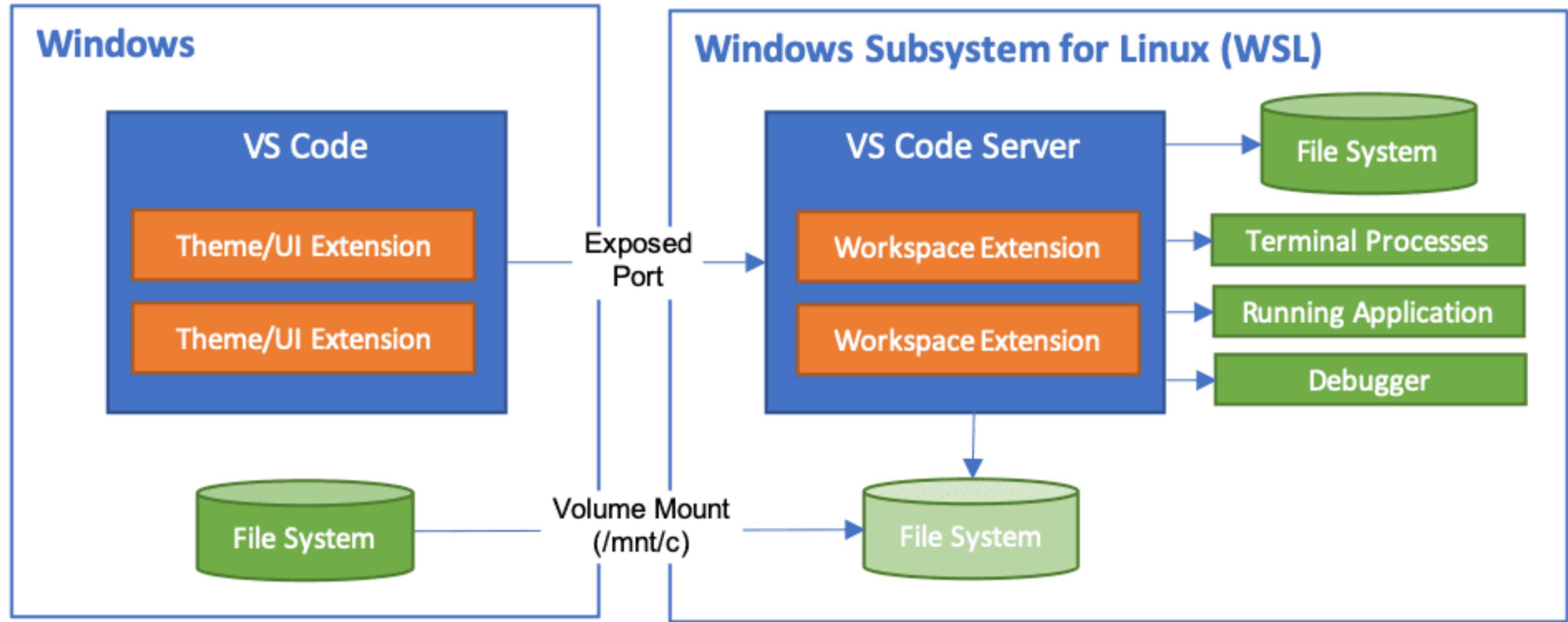
L'environnement Unix

- Dans ce cours nous utilisons des systèmes **Unix**
 - (Ubuntu) Linux
 - MacOS
 - Windows / WSL



WSL

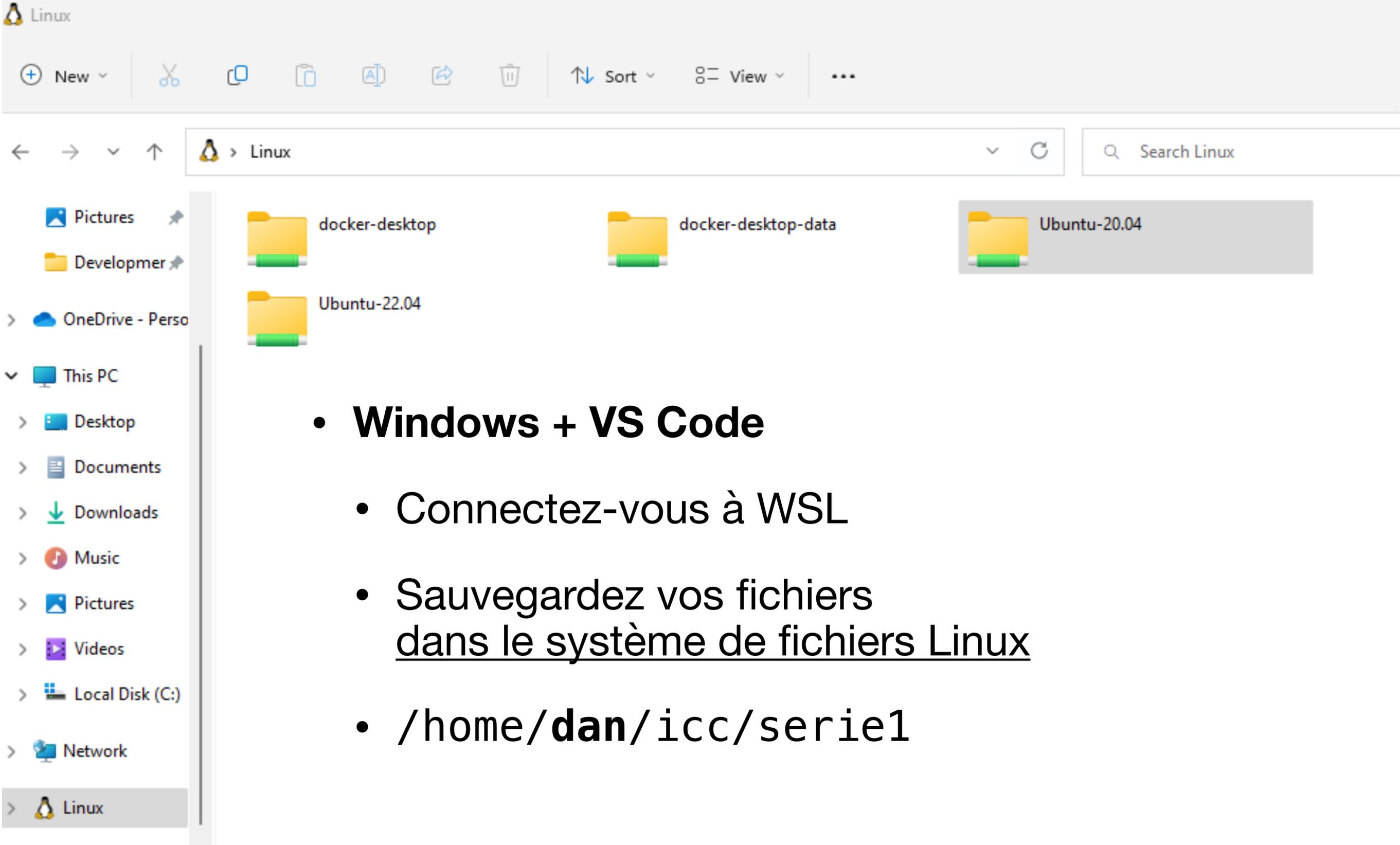
Qu'est-ce?



<https://code.visualstudio.com/docs/remote/wsl>

WSL

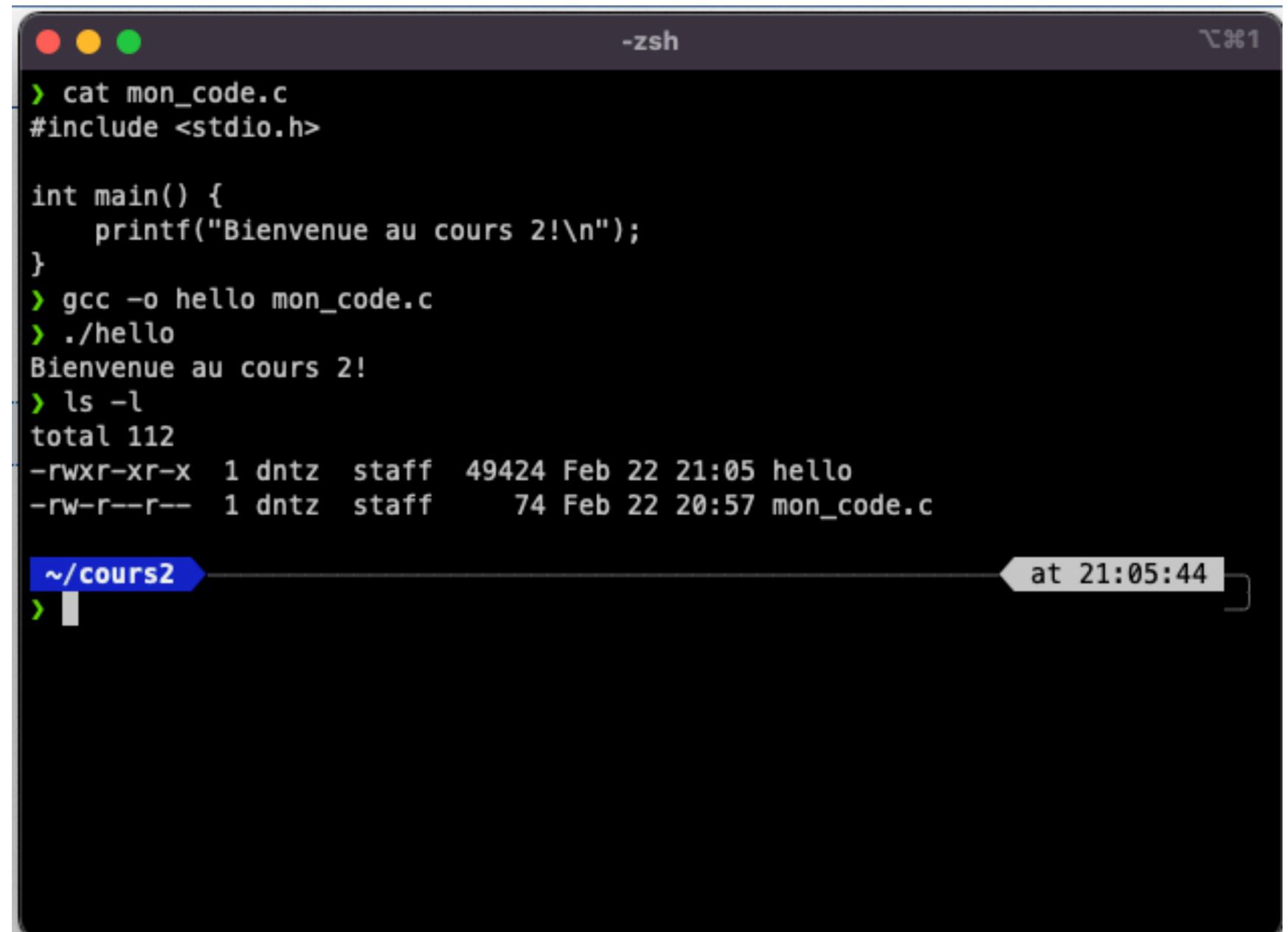
Windows Subsystem for Linux



- **Windows + VS Code**
 - Connectez-vous à WSL
 - Sauvegardez vos fichiers dans le système de fichiers Linux
 - `/home/dan/icc/serie1`

Le terminal

- Le terminal est un outil puissant pour interagir avec l'ordinateur
- On peut naviguer dans le [système de fichiers](#)
- Il permet de lancer d'autres programmes
- Par exemple, le [compilateur gcc](#)

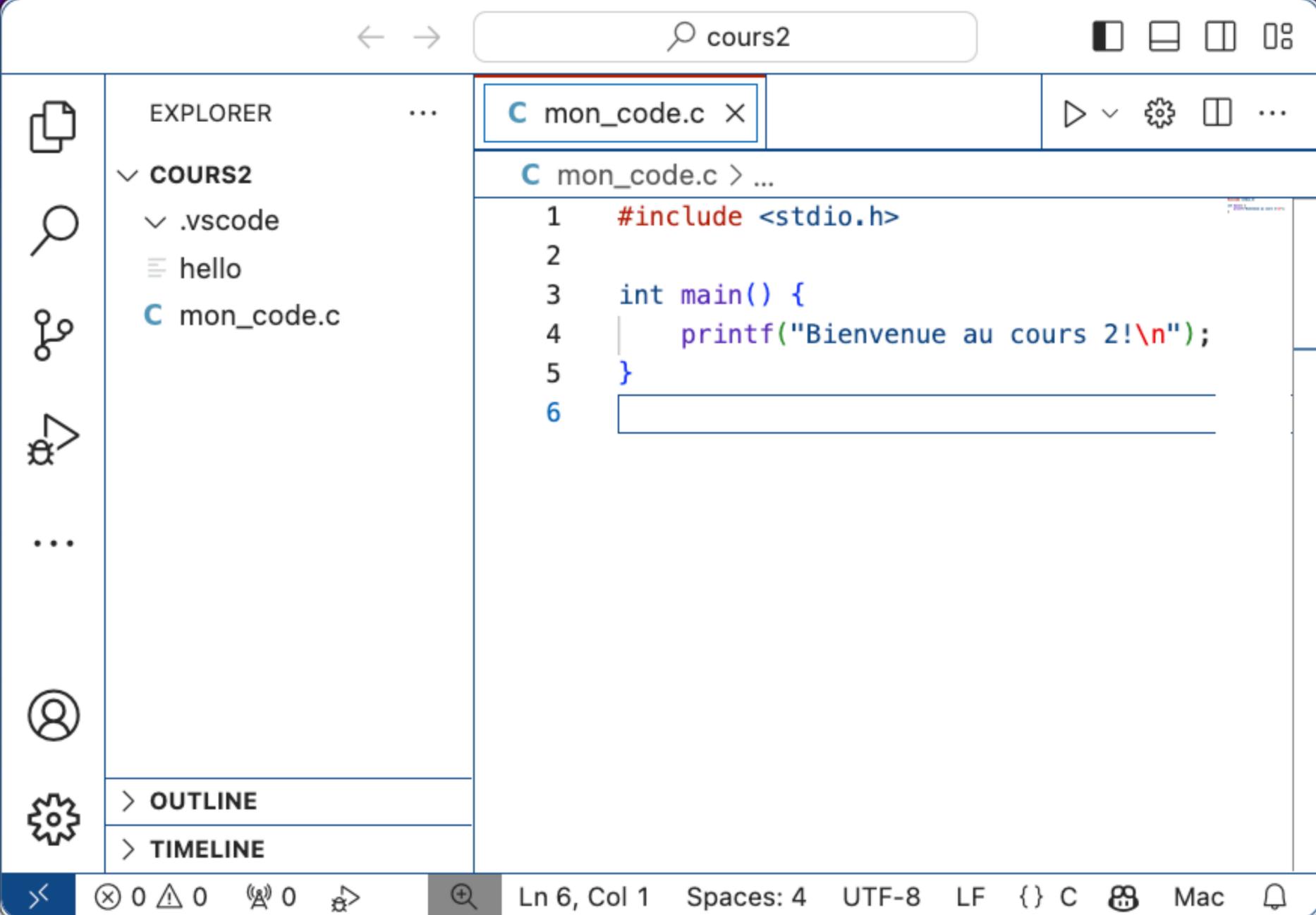


```
-zsh
> cat mon_code.c
#include <stdio.h>

int main() {
    printf("Bienvenue au cours 2!\n");
}
> gcc -o hello mon_code.c
> ./hello
Bienvenue au cours 2!
> ls -l
total 112
-rwxr-xr-x  1 dntz  staff  49424 Feb 22 21:05 hello
-rw-r--r--  1 dntz  staff    74 Feb 22 20:57 mon_code.c
~/cours2 at 21:05:44
>
```

VS Code

- IDE (*Integrated Development Environment*)
- Il comprend la **syntaxe** du C
- Il nous aide à écrire le code
- Pour l'instant on **compile** quand même dans le **terminal**



The screenshot shows the Visual Studio Code interface. The Explorer sidebar on the left displays a project named 'COURS2' with subfolders '.vscode' and 'hello', and a file 'mon_code.c'. The main editor area shows the content of 'mon_code.c', which is a C program. The code is as follows:

```
1  #include <stdio.h>
2
3  int main() {
4      printf("Bienvenue au cours 2!\n");
5  }
6
```

The status bar at the bottom indicates the current position is 'Ln 6, Col 1', the file encoding is 'UTF-8', the line ending is 'LF', and the file type is 'C'. The system tray shows 'Mac' and a notification bell.

Un programme C

- Fichiers texte .c
- Toutes les instructions C doivent être contenues dans `main`
- ... en tout cas dans la définition d'une `fonction`

```
#include <stdio.h>

int main()
{
    votre code ici
}
```

Types, variables et constantes

- Chaque **valeur** a un **type**
- Une **constante** a une **valeur** fixe tout au long de l'exécution
- Une **variable** peut changer de **valeur** pendant l'exécution
- On utilise l'**opérateur d'affectation** =

```
const double pi = 3.141592;  
  
int a = 10, b = 50;  
  
int c; //non initialisée  
  
c = b;  
b = a;  
a = c; //a vaut 50 et b vaut 10
```

Tableaux

Arrays

- Un tableau stocke plusieurs valeurs du même type

`type nom[taille_max] = valeur`

- Les indices commencent à 0
- ... et vont jusqu'à $(\text{taille_max} - 1)$ ⚠

```
int tableau[4] = {1, 2, 3, 4};  
  
printf("Premier element: %d\n",  
       tableau[0]);  
// Affiche  
// Premier element: 1  
  
printf("Dernier element: %d\n",  
       tableau[3]);  
// Affiche  
// Dernier element: 4
```

Tableaux multidimensionnels

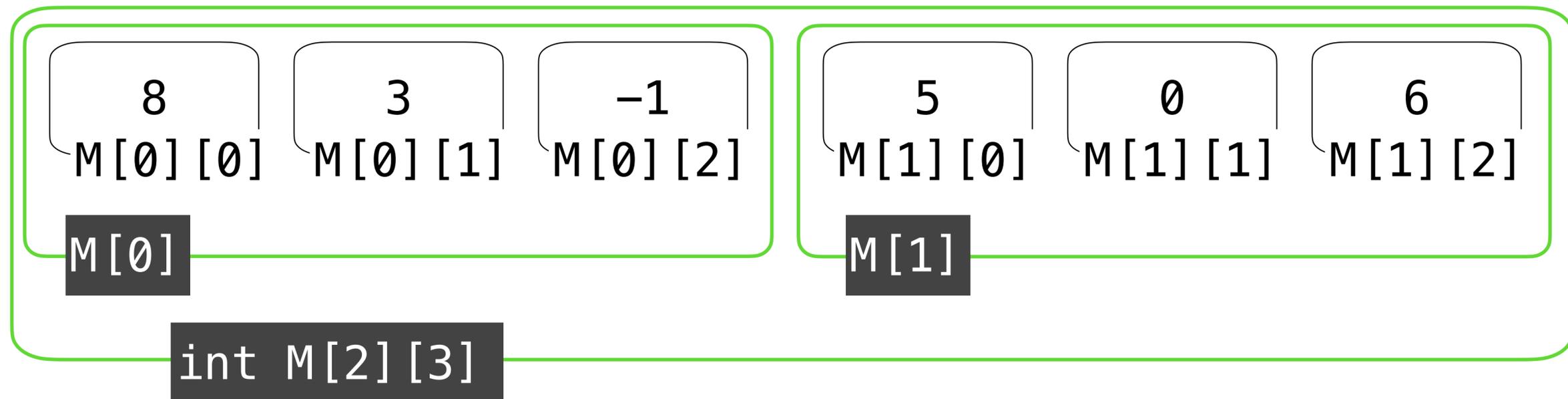
Multidimensional arrays

- On peut définir un tableau de tableaux ! 🤯
- `type nom[d1][d2]`
 - On lit de droite à gauche
 - “Un tableau de d1 tableaux de taille d2”
- `type nom[d1][d2][d3] ... [dn]`

```
int M[2][3] = {  
    {8, 3, -1},  
    {5, 0, 6},  
};
```

Tableaux multidimensionnels

```
int M[2][3] = {  
    {8, 3, -1},  
    {5, 0, 6},  
};
```



Afficher avec printf

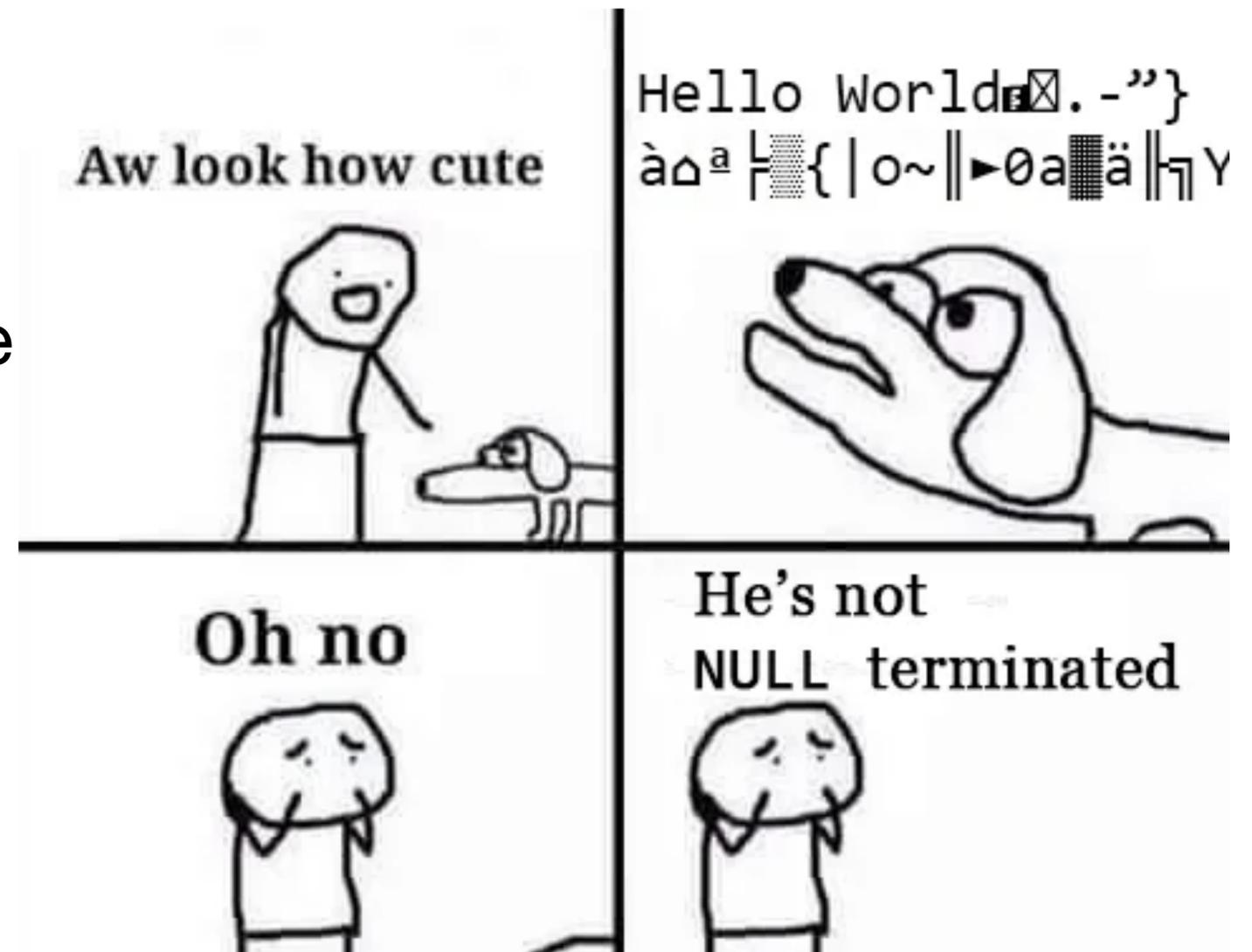
“Voici un entier %d et puis un réel %g”

| | |
|-----|--------|
| %s | string |
| %c | char |
| %g | double |
| %d | int |
| %lf | double |

```
printf("Voici un entier %d et puis un réel %g", 13, 3.14);  
// Affiche: Voici un entier 13 et puis un réel 3.14
```

Chaînes de caractères

- `char[]` = chaîne de caractères (*string*)
- Si on écrit `"Hello"` (*double-quotes*) le compilateur crée un `tableau` constant de 6 caractères
- Le 6e caractère = caractère spécial `'\0'`
- Indique la fin de la chaîne
- Si on affiche un string sans fin, on risque de voir n'importe quoi s'afficher ✨



Lire

stdin

Lecture de valeurs simples avec `scanf`

- L'entrée standard (*standard input*, *stdin*) est une source de données
- `printf` écrit des *strings* vers *stdout*

“Affiche-moi ceci à l’écran”

- `scanf` lit des *valeurs* depuis *stdin*

“L'utilisateur va entrer une valeur, stocke-la dans cette variable”

Lecture de valeurs simples avec `scanf`

- `scanf` prend en premier argument la *format string*
- Les autres arguments sont des *variables* qui reçoivent la *valeur*
- ⚠ Elles sont précédées de l'*opérateur* `&`
- ⚠ Sauf si c'est des tableaux...

```
int a, b;

printf("Entrez la premiere valeur:");

scanf("%d",
      &a);

printf("Entrez la deuxieme valeur:");
scanf("%d", &b);

printf("La somme de %d et %d est %d\n",
      a, b,
      a + b);
```

Lecture de strings avec `scanf`

- *format string* `%S`
-  Puisque c'est un tableau, ***il ne faut pas*** utiliser l'**opérateur** `&`
 - Pourquoi?
Ne ratez pas les épisodes suivants...
-  Au premier "espace" la lecture s'arrête

```
char une_chaine_sans_espace[100];  
printf("Entrez une chaîne de caractères: ");  
  
scanf("%s",  
      une_chaine_sans_espace);  
  
printf("Vous avez entré \"%s\"\n",  
      une_chaine_sans_espace);
```

Lecture de strings avec `fgets`

- Le 1er paramètre est la variable qui reçoit le texte
 -  Puisque c'est un tableau, ***il ne faut pas*** utiliser l'opérateur `&`
- Le 2è paramètre est le # max de caractères à lire
- Le 3è paramètre spécifie le flux à utiliser — `stdin`
-  Au premier "retour à la ligne" `'\n'` la lecture s'arrête
- Tous les espaces de la ligne seront lus, y compris le dernier retour à la ligne

```
char une_ligne[100];
printf("Entrez une ligne de texte: ");

fgets(une_ligne,
      100,
      stdin);

printf("Vous avez entré\n"
       "\"%s\"\n",
       une_ligne);
// Si on entre "Bonjour tout le monde"
// Affiche:
// Vous avez entré
// "Bonjour tout le monde
// "
```

Bon à savoir

- Quand un programme demande à lire depuis *stdin*, il reste bloqué 🤤
- Dans la console quand on termine une ligne en appuyant sur “entrée”:
 - on envoie **une ligne entière** au *stdin*
 - on réveille le programme qui, selon le code, en lit autant qu’il a besoin
 - ⚠️ avec *scanf* pas forcément toute la ligne d’un coup!
 - ⚠️ vaut mieux éviter de mélanger *scanf* et *fgets*

Exemple d'appels à `scanf` et `fgets`

Le code:

```
scanf("%s", mot1);  
scanf("%s", mot2);  
scanf("%d", &entier);  
fgets(ligne1, 10, stdin);  
fgets(ligne2, 10, stdin);
```

Ce qu'entre l'utilisateur:

```
Bonjour ICC 25 ↵  
Ceci est une autre ligne ↵
```

Que contiennent les 5 variables ?

stdin

Bonjour ICC 25 ↵

stdin

“Bonjour ICC 25\n”

```
scanf("%s", mot1); // mot1 vaut “Bonjour”
```

stdin

“ICC 25\n”

```
scanf("%s", mot2); // mot2 vaut “ICC”
```

stdin

“25\n”

stdin

stdin

"25\n"

```
scanf("%d", &entier); // entier vaut 25
```

stdin

"\n"

```
fgets(ligne1, 10, stdin); // ligne1 vaut "\n" !!
```

stdin

""

stdin

stdin

""



Ceci est une autre ligne ←

stdin

"Ceci est une autre ligne\n"

```
fgets(ligne2, 10, stdin); // ligne2 vaut "Ceci est une autre ligne\n"
```

stdin

""

Expressions

et opérations

DCT, 2023.12

Expressions

- En appliquant des **opérateurs** à des **opérandes** nous obtenons des **expressions**
- Expression constante:
 $(12 + 100) / (7 * 2 + 6) - 10$
- Expression contenant des variables (`int diametre_cm` est une variable):
 $3.14 * (\text{diametre_cm} / 2.0) * (\text{diametre_cm} / 2.0)$
- Chaque **expression** a ***toujours***
 - un **type** déterminé à la compilation
 - une **valeur** qui est complètement déterminée à l'exécution

Evaluer une expression

- Evaluer une expression signifie trouver son type et sa valeur
- L'expression doit être valide (types compatibles pour chaque opération)
 - Une expression invalide produit une erreur de compilation
- L'ordre des opérations est important
 - Pour une meilleure lisibilité, *utilisez les parenthèses* 🙏

L'ordre des opérations en C

| Ordre | Catégorie | Operateur | Associativité |
|-------|-----------------|-----------------------------------|--------------------|
| 1 | Postfix | () [] -> . ++ -- | De gauche à droite |
| 2 | Unaire (préfix) | + - ! ~ ++ -- (type)* & sizeof | De droite à gauche |
| 3 | Multiplicatif | * / % | De gauche à droite |
| 4 | Additif | + - | |
| 5 | Shift | << >> | |
| 6 | Relationnel | < <= > >= | |
| 7 | Egalité | == != | |
| 8 | ET par bits | & | |
| 9 | XOR par bits | ^ | |
| 10 | OU par bits | | |
| 11 | ET logique | && | |
| 12 | OU logique | | |
| 13 | Conditionnel | ?: | De droite à gauche |
| 14 | Affectation | = += -= *= /= %= >>= <<= &= ^= = | De droite à gauche |
| 15 | Virgule | , | De gauche à droite |

Opérateurs arithmétiques

- A priori rien de spécial
- La multiplication et la division ont la priorité sur l'addition et la soustraction

$$\begin{array}{l} 9 + 5 * 4 \\ 9 + 5 * 4 \\ 9 + 20 \\ 29 \end{array}$$

Opérateurs arithmétiques

- La division / dépend du type des opérandes

9 + 5 / 4
9 + 5 / 4
9 + 1
10

- La **division entière** si les deux opérandes sont des entiers

Opérateurs arithmétiques

- Pour la division réelle

9 + 5 / 4.0

9 + 5 / 4.0

9 + 1.25

10.25

- Il suffit qu'un des opérandes soit un réel

Conversions *implicites* de type

Implicit casting

- Si on multiplie les deux constantes $100 * 4.0$ on obtient une **expression** de type **double** (et de valeur 400.0)

$100 * 4.0$

$100.0 * 4.0$

400.0

- En général, une opération arithmétique entre deux **types** numériques va déclencher une **conversion implicite** du type le plus contraignant vers celui le moins contraignant

Opérateurs arithmétiques

- Il existe aussi l'opérateur "reste de la division entière"

9 + 5 % 3

9 + 5 % 3

9 + 2

11

- Il a la priorité sur l'addition et la soustraction

Opérateurs arithmétiques

- ⚠ La division par zéro n'est pas définie
- ⚠ Elle ne produit pas forcément une erreur!
- Quand c'est *explicite*, le compilateur vous l'indiquera

```
printf("Division par zéro: 5 / 0 = %d\n",  
      5 / 0);  
// Affiche: Division par zéro: 5 / 0 = 73896
```

```
values.c:142:14: warning: division by zero is undefined [-Wdivision-by-zero]  
                5 / 0);  
                  ^ ~
```



Exemple 1

$12 + 100 / 7 * 2 + 6 - 10$

- Le **type** de chaque opérande est `int`, donc le **type** de cette expression est `int`
 - ⚠ l'opération de division est la *division entière*
- Calculons la valeur de cette **expression**
 - Les opérateurs multiplicatifs ont la priorité sur les opérateurs additifs
 - L'associativité est de gauche à droite

Exemple 1

$$12 + 100 / 7 * 2 + 6 - 10$$

$$12 + 14 * 2 + 6 - 10$$

$$12 + 14 * 2 + 6 - 10$$

$$12 + 28 + 6 - 10$$

$$12 + 28 + 6 - 10$$

$$40 + 6 - 10$$

etc.

- Conclusion: la valeur de cette expression est (int) 36

Exemple 2

$$(12 + 100) / (7 * 2 + 6) - 10$$

- Le **type** de chaque opérande est `int`, par conséquent le **type** de cette expression est aussi `int`
 -  l'opération de division est de nouveau la division entière
- Calculons la valeur de cette **expression**
 - Les opérateurs multiplicatifs ont la priorité sur les opérateurs additifs
 - L'associativité est de gauche à droite

Exemple 2

$$\begin{aligned} & (12 + 100) / (7 * 2 + 6) - 10 \\ & 112 / (7 * 2 + 6) - 10 \\ & 112 / (14 + 6) - 10 \\ & 112 / (14 + 6) - 10 \\ & 112 / 20 - 10 \\ & 112 / 20 - 10 \\ & 5 - 10 \end{aligned}$$

- Conclusion: la **valeur** de cette expression est (int) -5

Exemple 3

$$(12 + 100) / (7 * 2 + 6.0) - 10$$

- Le **type** de 6.0 est **double**, par conséquent le **type** de cette expression devient aussi **double**
 -  l'opération de division est cette fois-ci la division réelle
- Calculons l'expression équivalente avec des parenthèses:
 - Les opérateurs multiplicatifs ont la priorité sur les opérateurs additifs
 - L'associativité est de gauche à droite

Exemple 3

$$(12 + 100) / (7 * 2 + 6.0) - 10$$

$$(112 / 20.0) - 10$$

$$(112 / 20.0) - 10$$

$$5.6 - 10$$

- Conclusion: la valeur de cette expression est (double) -4.4

L'opérateur conversion explicite de type

Casting

- Pour convertir le **type** d'une expression vers un autre **type** (si possible) on utilise l'opérateur de **conversion de type** (*type casting*)
- syntaxe: `(nouveau_type) (expression)`
- Exemple:

`(double) (4 + 7) vaut 11.0`

L'opérateur conversion de type

Casting

- Convertir un nombre réel en un nombre entier par ***troncation***:

```
double x = 32.5;  
int x_int = (int) x; // 32
```

```
double x_neg = -12.5;  
int x_neg_int = (int) x; // -12
```

- ⚠ Cela ne fonctionne pas pour convertir un *string* en un nombre!

```
double x_err = (double) "32.5"; // erreur!
```

- Il faut utiliser des fonctions spéciales — voir chapitre sur les *strings*