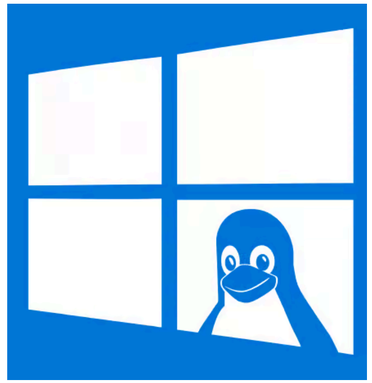
Rappels

et plus

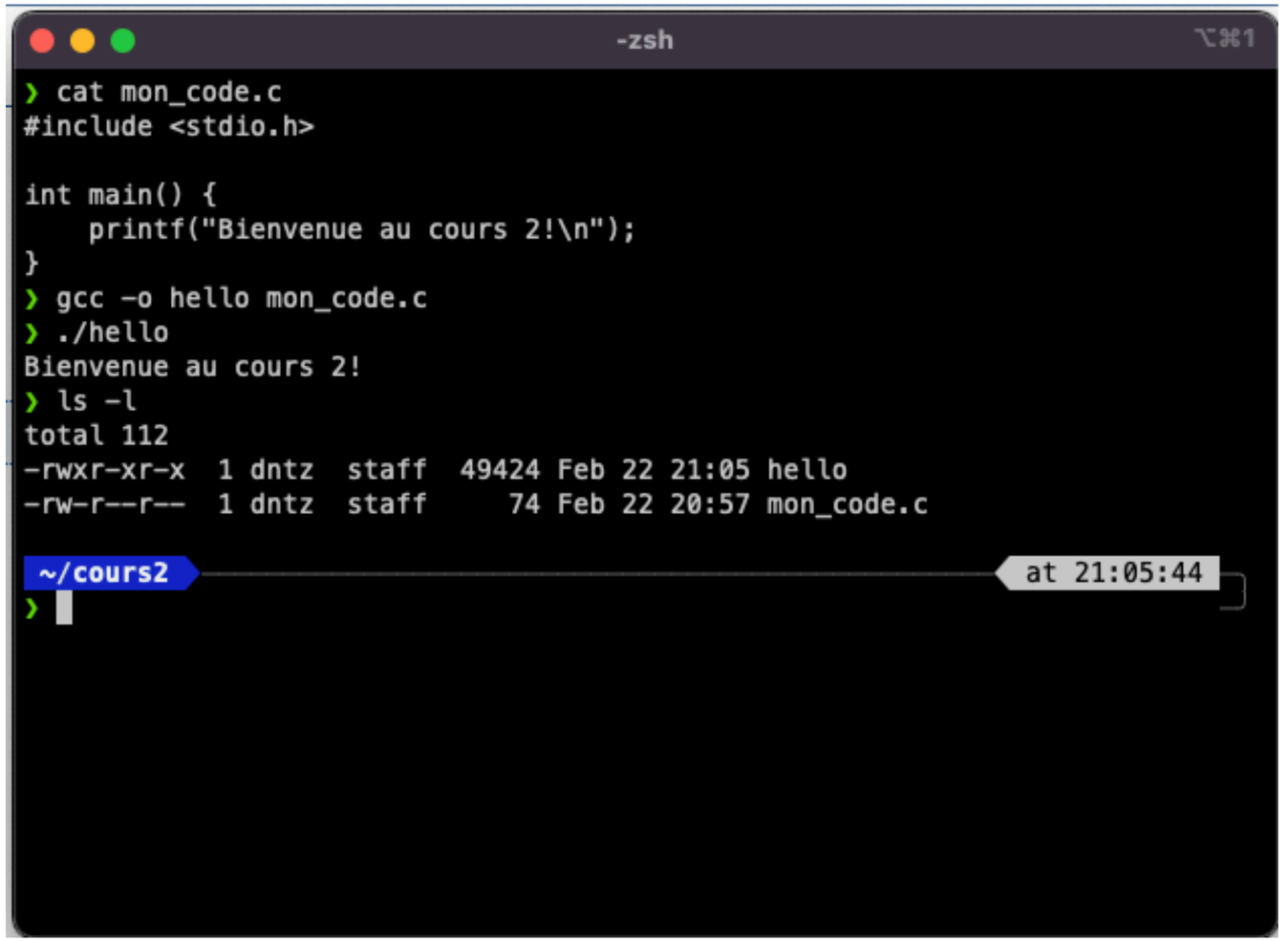
L’environment Unix

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



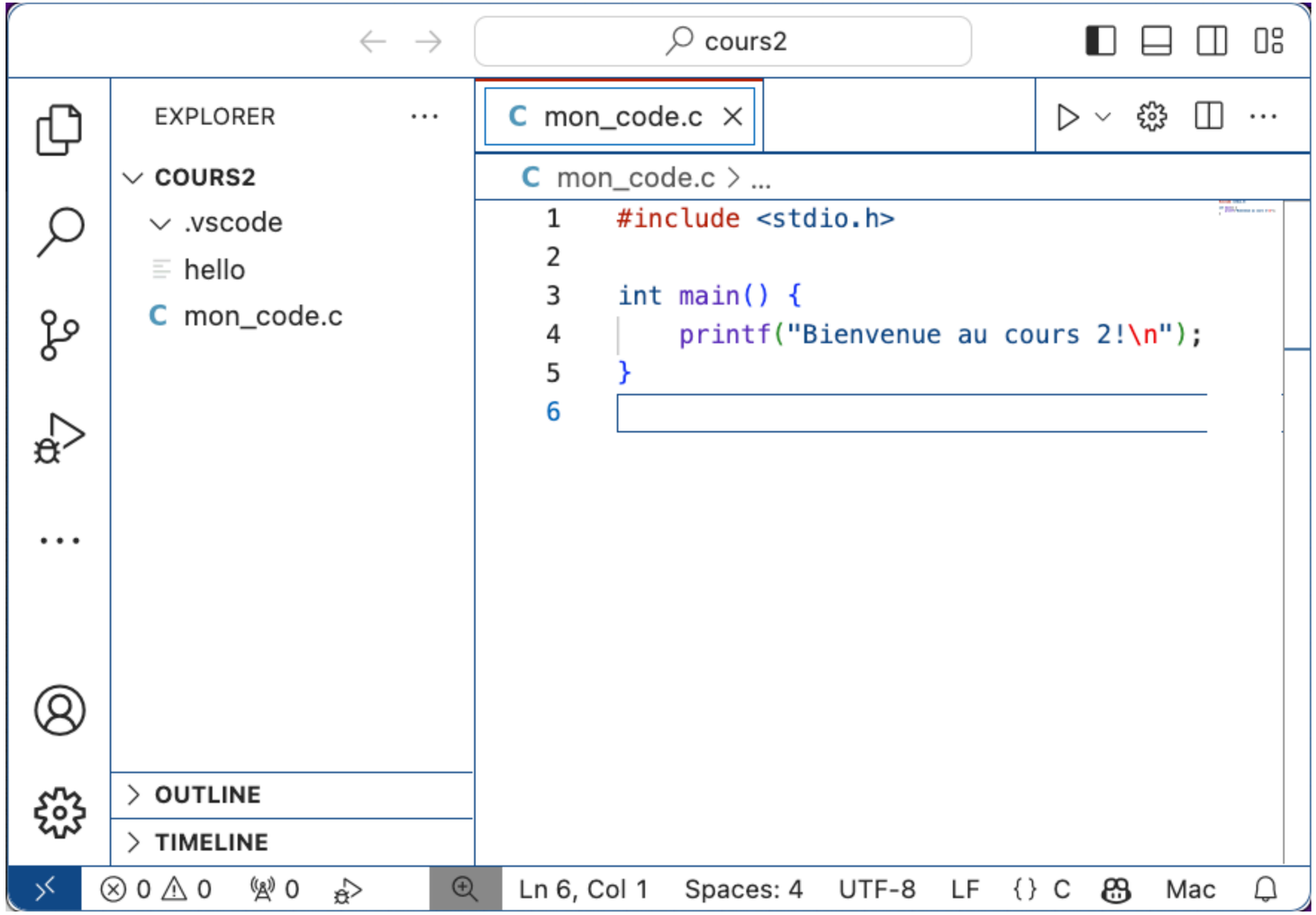
Le terminal

- 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`



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



Un programme C

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

```c
#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'execution
- Une variable peut changer de valeur pendant l'execution
- 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 (`taille_max - 1`) ⚠️

```c
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

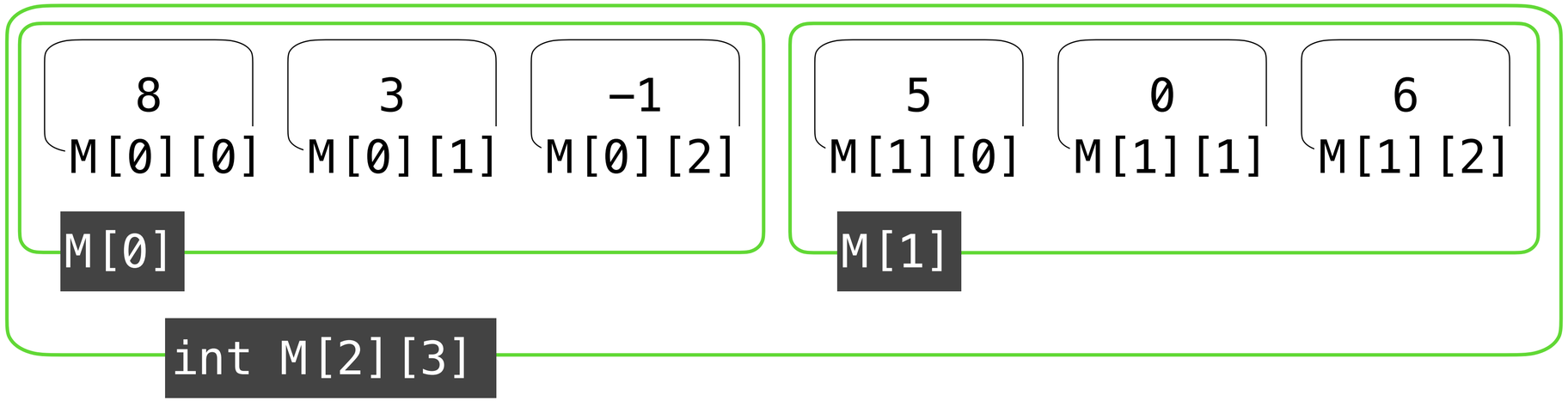
Multidimensional arrays

- On peut définir un tableau de tableaux ! 🤯
- `type nom[d1][d2]`
  - Signifie un tableau de d1 tableaux de taille d2
- `type nom[d1][d2][d3] ... [dn]`

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

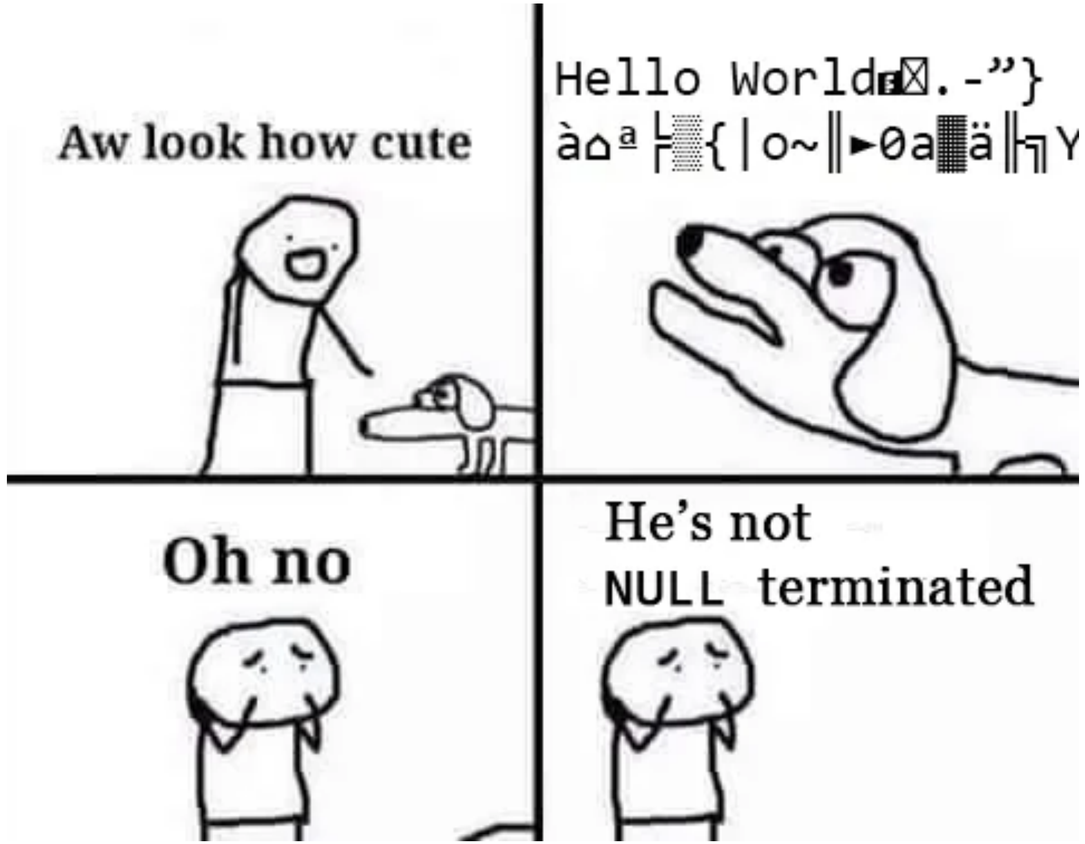
Tableaux multidimensionnels

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



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'`
- Autre caractère spécial — retour à la ligne: `'\n'`



Afficher avec printf

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

%s	string
%c	char
%g	double
%d	int
%f	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
```

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*
- `scanf` lit des valeurs depuis *stdin*
- Comme `printf`, elle prend en premier argument la *format string*
- Les arguments sont des variables qui reçoivent la valeur
- ⚠️ Elles sont précédées de l'opérateur &

```c
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`

- Déclarons une variable de type `char[100]`
- *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

```c
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`

- Pareil, déclarons une variable de type `char[100]`
- Le 1er paramètre est la variable qui reçoit le texte
  - ⚠️ Puisque c'est un tableau, ***il ne faut toujours 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

```c
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
// "
```

Premier programme interactif

```c
const double pi = 3.141592;
int diametre_cm;
char nom[100];

printf("Entrez le nom de votre pizza "
       "(sans espaces, max 99 caractères) "
       "suivi du diamètre en cm: “);

scanf("%s", nom);

scanf("%s", &diametre_cm);

double rayon_cm = diametre_cm / 2.0;

printf("La 🍕 '%s' a une surface de %g cm²\n",
       nom,
       pi * rayon_cm * rayon_cm);
```

```
❯ gcc -o pizza pizza.c
❯ ./pizza
Entrez le nom de votre pizza (sans espaces,
max 99 caractères) suivi du diamètre en cm:
Ortolana 40
La 🍕 'Ortolana' a une surface de 1256.64 cm²
```

Bon à savoir

- Quand un programme lit depuis *stdin*, il reste bloqué 😴 jusqu'à ce qu'il reçoive une entrée qui lui convient
- 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!
    - ⚠️ il faut éviter de mélanger `scanf` et `fgets`

stdin

```
Bonjour ICC 24 ↵
Ceci est une autre ligne ↵
```

stdin

“Bonjour ICC 24\nCeci est une autre ligne\n”

stdin

stdin “Bonjour ICC 24\nCeci est une autre ligne\n”

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

stdin “ICC 24\nCeci est une autre ligne\n”

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

stdin “24\nCeci est une autre ligne\n”

stdin

stdin “24\nCeci est une autre ligne\n”

```
scanf("%d", &entier); // entier vaut 24
```

stdin “\nCeci est une autre ligne\n”

```
fgets(chaine, 10, stdin); // chaine vaut “\n” !!
```

stdin “Ceci est une autre ligne\n”

Expressions

et opérations

DCT, 2023.12

Expressions

- En appliquant des opérateurs à des opérandes nous obtenons des expressions
- Exemple d'expression constante:
  `(12 + 100) / (7 * 2 + 6) - 10`
- Exemple d'expression contenant des variables
  (supposant que `double surface` et `int diametre_cm` sont des variables):

```
surface = 3.14 * (diametre_cm / 2.0) * (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	`= += -= *= /= %= >>= <<= &= ^= |=`	
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

```
9 + 5 * 4
9 + 5 * 4
9 + 20
29
```

Opérateurs arithmétiques

- Sauf… 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
```

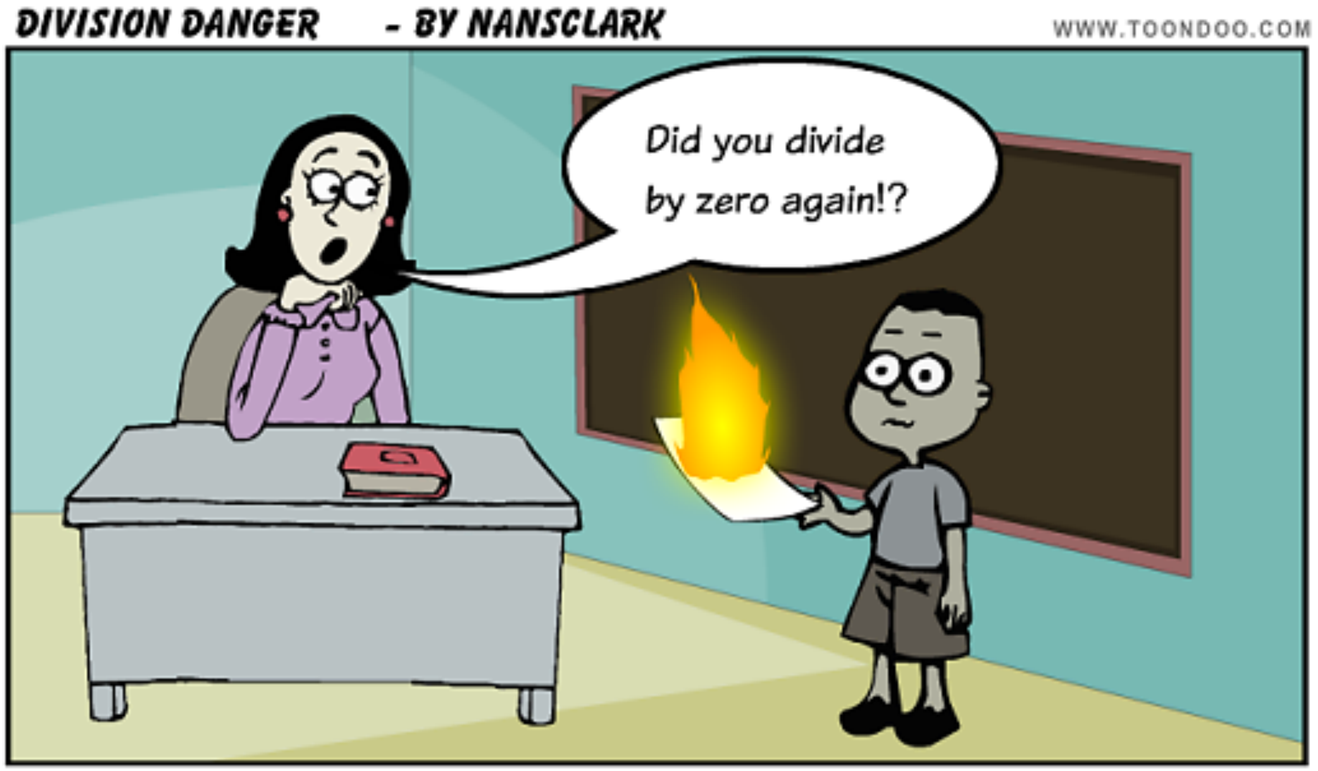
- Il a la priorité sur l'addition et la soustraction

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

```c
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);
                                            ^ ~
```


DIVISION DANGER - BY NANSCLARK
WWW.TOONDOO.COM
Did you divide by zero again!?

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

```
(12 + 100) / (7 * 2 + 6) - 10
112 / (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
```

- 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*

L’opérateur affectation

Assignment operator

```
int a, b, c, d;
a = 10;
b = a;
```

- L’opérande gauche est une variable, plus généralement une L-valeur (*L-value*)

- Permet de combiner plusieurs affectations en une même expression:

```
b = c = d = a;
b = (c = (d = a)); // La même chose
                   // associativité de droite à gauche
```

L-valeurs

- Peut être à gauche de l'opérateur d'affectation
  - Variables
  - Eléments de tableaux
- On ne peut pas écrire

```
(b = c) = a; // Erreur! (b = c) n'est pas une L-value
```

  - L'expression (`b = a`) a le type `int` et la valeur `10` (comme la variable a), mais ce n'est pas une *L-value*!

Affectation combinée

- On peut combiner des opérations et des affectations

`+=, *=, -=, /=, %=`

- L’expression `a += 2` est équivalente à
  `a = a + 2`
- Sa valeur est `a + 2` après évaluation
- Pareil pour les autres opérateurs

Encore des conversions implicites de type

Implicit type casting

- A gauche du “=” il y a une *L-value* avec un type bien déterminé
- Une conversion implicite aura lieu vers le type de la *L-value*

```
int exemple5;
exemple5 = (12 + 100) / (7 * 2 + 6.0) - 10;
```

- Exemple 3: que le résultat de l’évaluation du côté droit de l’affectation est un réel `(double) -4.4`
- Par contre, la *L-value* à gauche est un `int`
- La variable `exemple5` vaudra `(int) -4.4`, donc `(int) -4`