



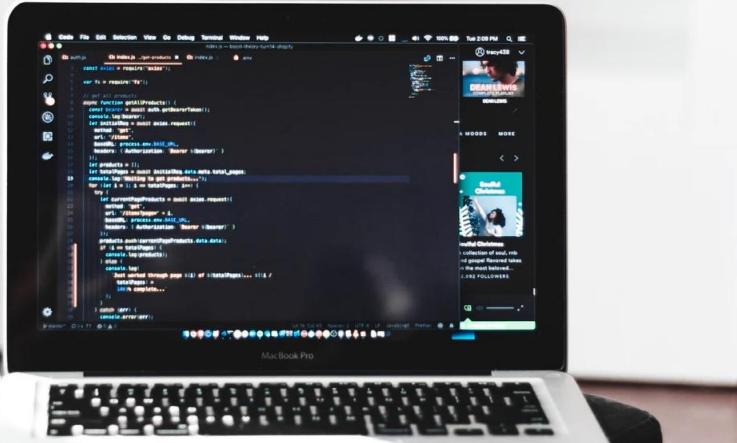
La grande école du numérique pour tous

# Débuter avec C++

## Rappels



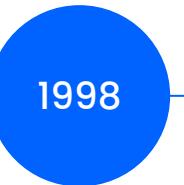
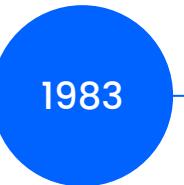
# / Un peu d'histoire



# Un peu d'histoire

## C with classes

Développé par Bjarne Stroustrup chez AT&T Bell Labs, une extension du langage C est née, puis renommée C++ en 1983.



C

Dennis Ritchie commence le développement du langage C chez AT&T Bell Labs, jusqu'en 1973.

Il est conçu pour le développement du système Unix.

## C++98

L'ISO (Organisation Internationale de normalisation) crée la première norme officielle du langage C++,

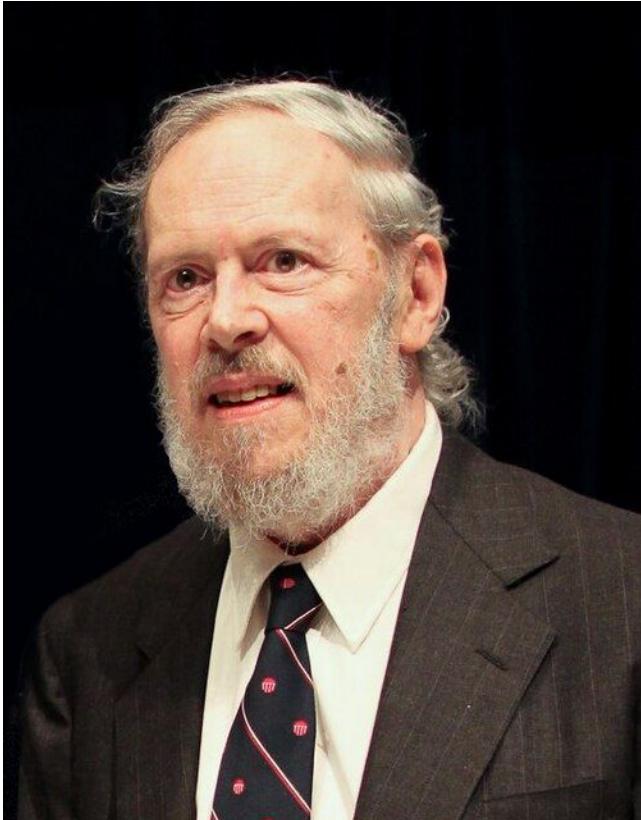
## Standard C

Le comité ANSI (American National Standards Institute) forme un comité pour standardiser le langage C. Le premier standard apparaît en **1989. (C89)**

## Normes modernes

Des versions plus modernes des deux langages continuent à être créées : C03, C++11, 17, 20, 23.

# Dennis Ritchie



**Dennis MacAlistair Ritchie**, né en 1941 aux USA et retrouvé mort en 2011 à Berkely Heights, New Jersey.

Il est un des pionniers de l'informatique moderne, inventeur du langage C et co-développeur d'Unix.

# Bjarne Stroustrup



**Bjarne Stroustrup**, né en 1950 à Aarhus, est un informaticien, écrivain et professeur de sciences informatiques danois.

Il est connu pour être l'auteur du langage de programmation C++, l'un des plus utilisés dans le monde.

# / Interpréte - Compilé



# Interprété / Compilé

## Langage Interprété

Code source

-Lu ligne par ligne



Interpréteur



Donnée de sortie

## Langage Compilé

Code source



Compilateur



Bytecode



Donnée de sortie



Système d'exploitation



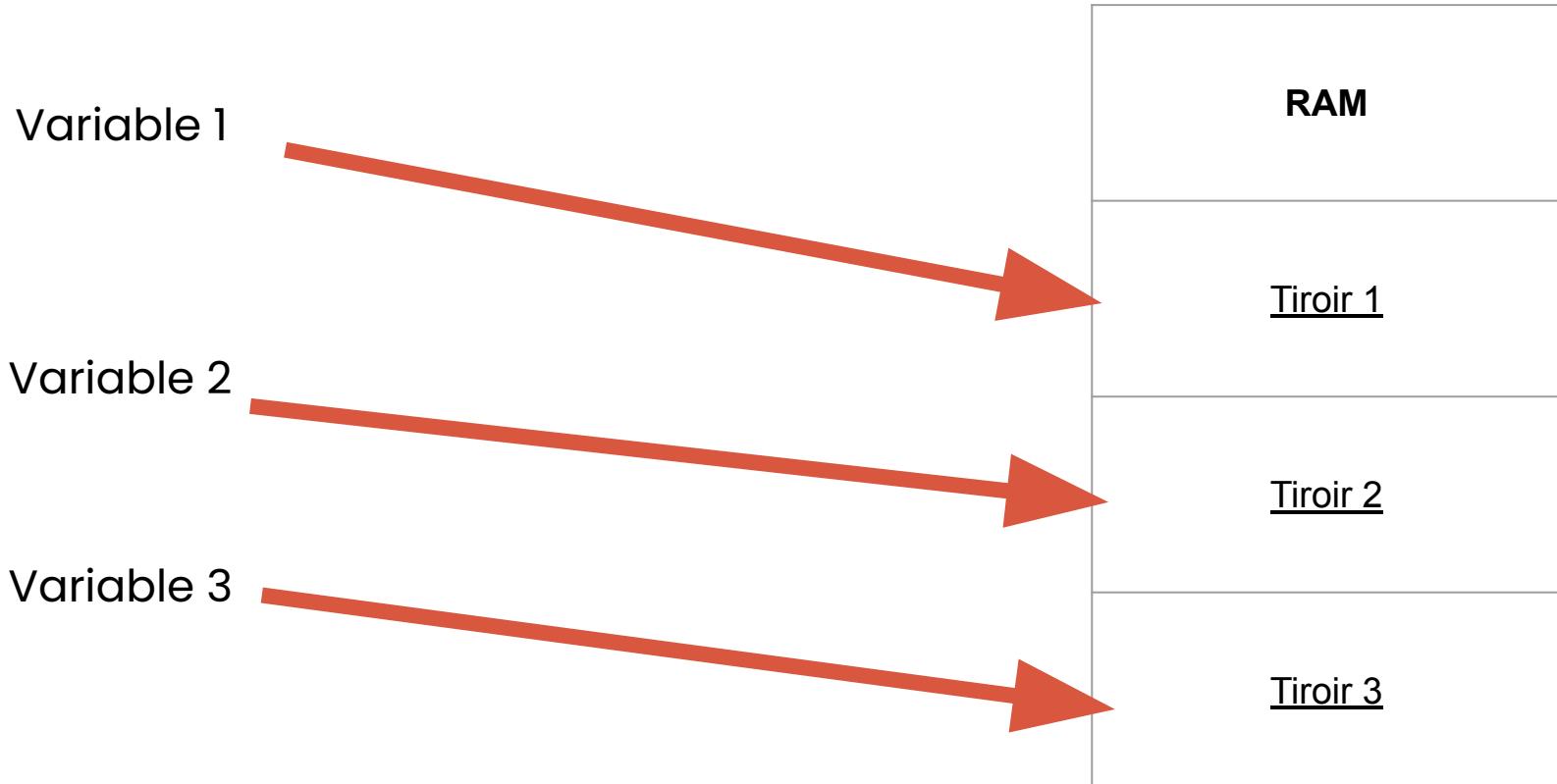
# Types de données

Type	Description	Taille
<b>char</b>	Caractère (stocke un seul caractère ASCII)	1 octet
<b>int</b>	Entier (nombre entier signé)	4 ou 16 octets (selon le compilateur)
<b>float</b>	Nombre à virgule flottante simple précision (~1 à 7 chiffres)	4 octets
<b>double</b>	Nombre à virgule flottante double précision (~15 à 16 chiffres)	8 octets
<b>array</b>	Tableau de valeurs (ex: int tab[2];)	Somme de ses membres
<b>struct</b>	Structure de données personnalisée	Somme de ses membres
<b>enum</b>	Enumération (liste de valeurs nommées)	typiquement stocké comme un int, qui fait généralement 4 octets
<b>pointers</b>	Pointeur vers un type (int *p; pour un pointeur vers un entier)	4 octets (32bits), 8 octets (64bits)
<b>bool</b>	expression booléenne	1 octet

# / Les pointeurs



# Les variables en mémoire

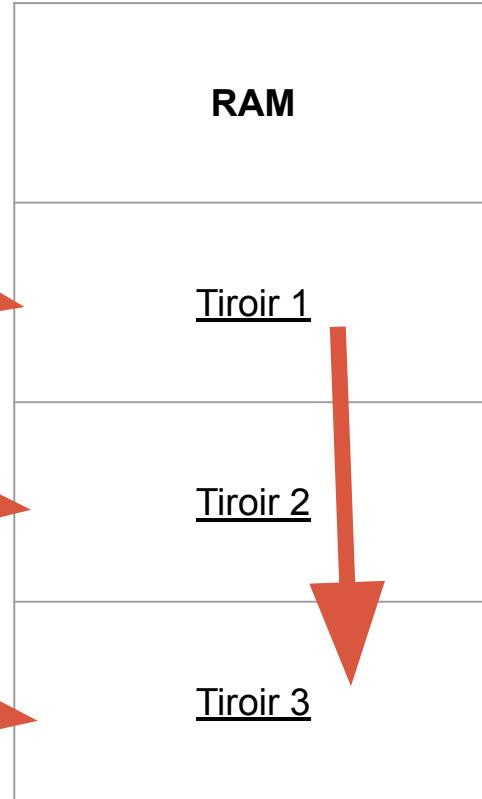


# Les pointeurs

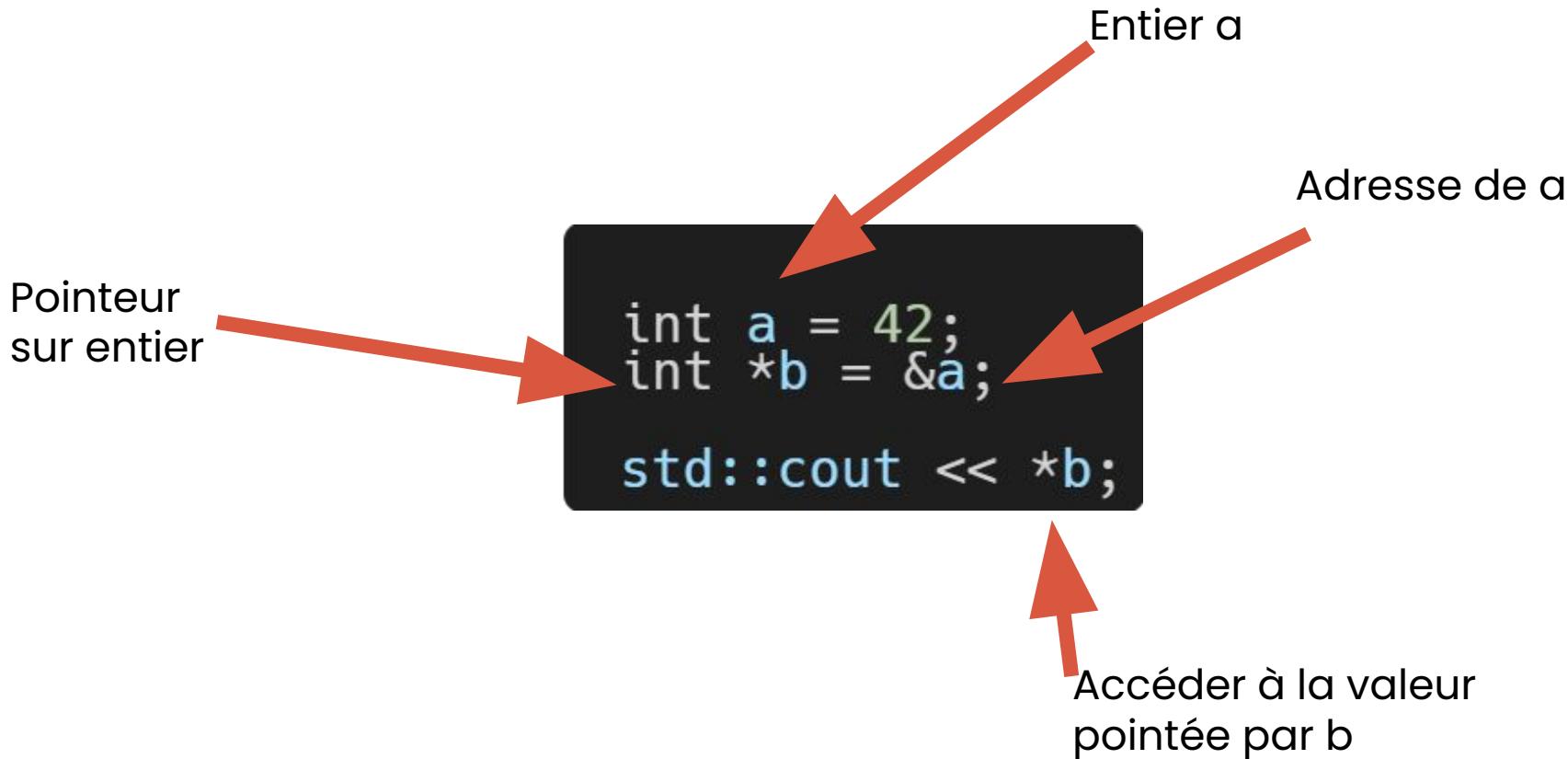
Variable 1  
pointe vers  
variable 3

Variable 2

Variable 3



# Les pointeurs - syntaxe



# Les pointeurs - syntaxe

Pointeur sur entier

Adresse de a

Pointeur sur pointeur sur entier

Adresse de b

Accéder à la valeur pointée par la valeur pointée par c.

```
int a = 42;  
int *b = &a;  
int **c = &b;  
  
std::cout << **c;
```

# Les pointeurs - syntaxe

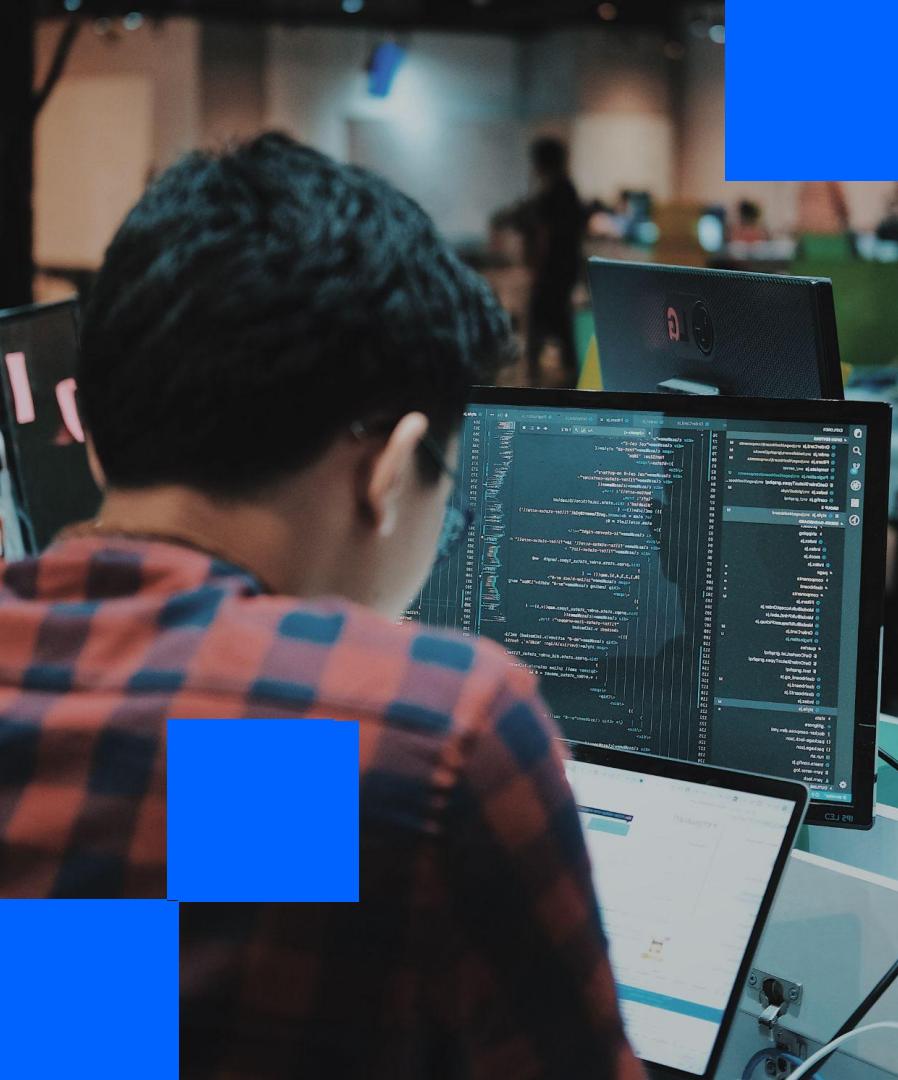


```
#include <iostream>

int main() {
    int a = 42;
    int *b = &a;
    int **c = &b;

    std::cout << "a = " << a << "\n";           // 42
    std::cout << "*b = " << *b << "\n";         // 42
    std::cout << "**c = " << **c << "\n";        // 42
    std::cout << "*c = " << *c << "\n";        // Adresse de a (même que b)
    std::cout << "c = " << c << "\n";          // Adresse de b
}
```

# / Les tableaux



# Les tableaux

Variable 1  
pointe vers  
tiroir 3

Début du  
tableau

Index 1 du  
tableau

Index 3 du  
tableau

RAM
Tiroir 1
Tiroir 2
Tiroir 3
Tiroir 4
Tiroir 5
Tiroir 6

# Les tableaux – Déclaration

Type de données du tableau

Nom du tableau

Taille du tableau



```
int my_array[10];  
int mon_tableau[5] = {1, 2, 3, 4, 5};  
int mon_tab[] = {1, 2, 3, 4, 5};
```

# Les tableaux – utilisation



```
#include <iostream>

void afficherTableau(int monTableau[], int taille) {
    for (int i = 0; i < taille; ++i) {
        std::cout << monTableau[i] << std::endl;
    }
}
```



```
#include <iostream>

void afficherTableau(int* monTableau, int taille) {
    for (int i = 0; i < taille; ++i) {
        std::cout << *monTableau << std::endl;
        monTableau++;
    }
}
```



```
int main() {
    int monTableau[] = {1, 2, 3, 4, 5};
    int taille = 5;
    afficherTableau(monTableau, taille);
    return 0;
}
```

# Matrices/Multidimensionnels

## Déclaration



```
int tableau[][] = {  
    {1, 2, 3},  
    {4, 5, 6}  
};
```

Non valide



```
int tableau[][3] = {  
    {1, 2, 3},  
    {4, 5, 6}  
};
```

Valide



```
int tableau[2][3] = {  
    {1, 2, 3},  
    {4, 5, 6}  
};
```

Valide

# Matrices/Multidimensionnels

## Utilisation

```
● ● ●  
int tableau[][][3] = {  
    {1, 2, 3},  
    {4, 5, 6}  
};
```

```
● ● ●  
void afficherTableauSimple(int tableau[][][3], int lignes) {  
    for (int i = 0; i < lignes; ++i) {  
        for (int j = 0; j < 3; ++j) {  
            std::cout << tableau[i][j] << std::endl;  
        }  
    }  
}
```

```
● ● ●  
void afficherTableauAvecPointeurs(int* tableau, int lignes) {  
    for (int i = 0; i < lignes; ++i) {  
        for (int j = 0; j < 3; ++j) {  
            std::cout << *tableau << std::endl;  
            tableau++;  
        }  
    }  
}
```

# Passage par référence

Change la  
valeur pointée

Affiche 21

Affiche 42

```
void change_int(int a)
{
    a = 42;
}

void change_ptr(int *a)
{
    *a = 42;
}

int main(void)
{
    int a = 21;
    change_int(a);
    std::cout << a;
    change_ptr(&a);
    std::cout << a;
    return (0);
}
```

# / Allocation

- Quand on crée une variable, on demande au système d'exploitation de réserver de la mémoire pour notre programme.
- Deux types d'allocation existent : Statique, et Dynamique.



# Allocation statique

**La plupart des allocations sont statiques.**

**Un entier prend 4 octets en mémoire.**

**On demande la place pour 10 entiers.**

```
int tableau[10];
```

**Le système donne donc 40 octets au programme pour stocker le tableau.**

# Allocation dynamique

Parfois, on ne connaît pas les besoins en mémoire avant l'exécution du programme. Dès lors, l'allocation doit être dynamique.

## Malloc



```
int size = 5 ;  
int* tableau = (int*)malloc(size * sizeof(int));  
free(tableau); // libération
```

- Ne déclenche pas de constructeur, donc **pas de surcharge inutile** pour les types primitifs
- Permet de **redimensionner** la mémoire allouée sans copier les données (**realloc**).

## New



```
int size = 5;  
int* tableau = new int[size];  
delete[] tableau ;
```

- Alloue la mémoire **et** appelle les constructeurs des objets (si besoins)
- Pas besoin de **cast**, contrairement à Malloc
- Utilisé avec **delete**, ce qui appelle aussi les destructeurs (si besoin).

# Allocation dynamique

RAM	<u>Tiroir 1</u>	<u>Tiroir 2</u>	<u>Tiroir 3</u>	<u>Tiroir 4</u>	<u>Tiroir 5</u>	<u>Tiroir 6</u>	<u>Tiroir 7</u>	<u>Tiroir 8</u>	<u>Tiroir 9</u>	...
-----	-----------------	-----------------	-----------------	-----------------	-----------------	-----------------	-----------------	-----------------	-----------------	-----



**Votre programme, et sa  
mémoire statique.**

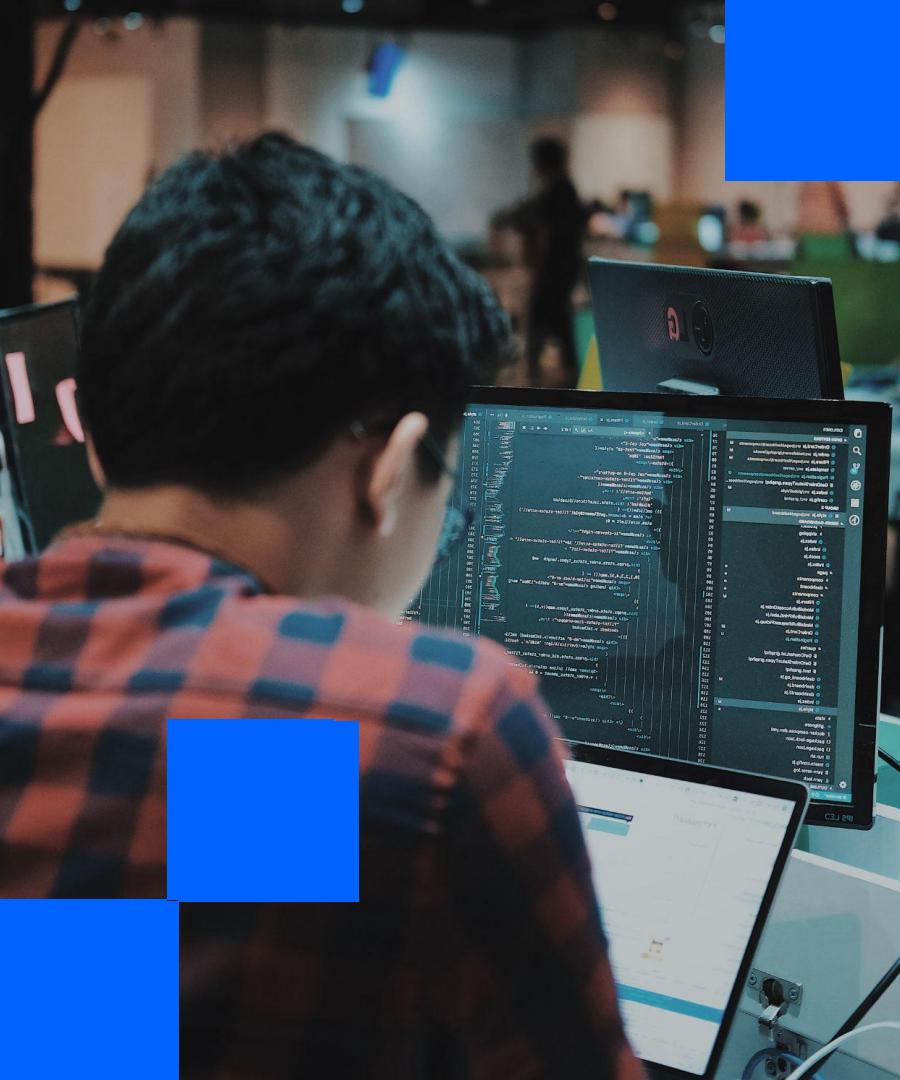
**Un autre  
programme.**

**Mémoire  
dynamique  
de votre  
programme.**

# Fuites de mémoire

```
int *array;  
  
for (int i = 0; i < 10; i++)  
{  
    array = malloc(5 * sizeof(int));  
}  
free(array);
```

Ici `free` est appelé alors que 10 allocations ont été faites.  
Les 9 premiers pointeurs sont inaccessibles, mais leur mémoire n'a pas été libérée.



# / Préprocesseur

- Tout code qui commence par un # est une instruction dite de préprocesseur.
- Ces instructions sont lues et interprétées par le préprocesseur.
- Ces instructions disparaissent du code final.

# / Headers



# Exemple

```
#pragma once  
  
#define PI (3.14159)  
  
#define ABS(x) (x < 0 ? -x : x)  
  
typedef struct  
{  
    int x;  
    int y;  
} point;  
  
int distance(point a, point b);
```

# Garde-fou

```
#pragma once
```

**Cette instruction indique que le contenu ne doit être compilé qu'une seule fois. C'est donc une protection contre les imports circulaires.**

# / Macros



# Macro

```
#define PI (3.14159)
```

**Define déclare une constante. A la compilation,  
chaque fois que le compilateur verra "PI", il le  
remplacera par 3.14159.**

# Macro

```
#define ABS(x) (x < 0 ? -x : x)
```

**Une macro peut recevoir des arguments, et les traiter de manière adéquate.**



# / Structures

- Une structure est un type de données complexe défini par le développeur.
- Une structure peut contenir n'importe quel type de données, en n'importe quelle quantité.

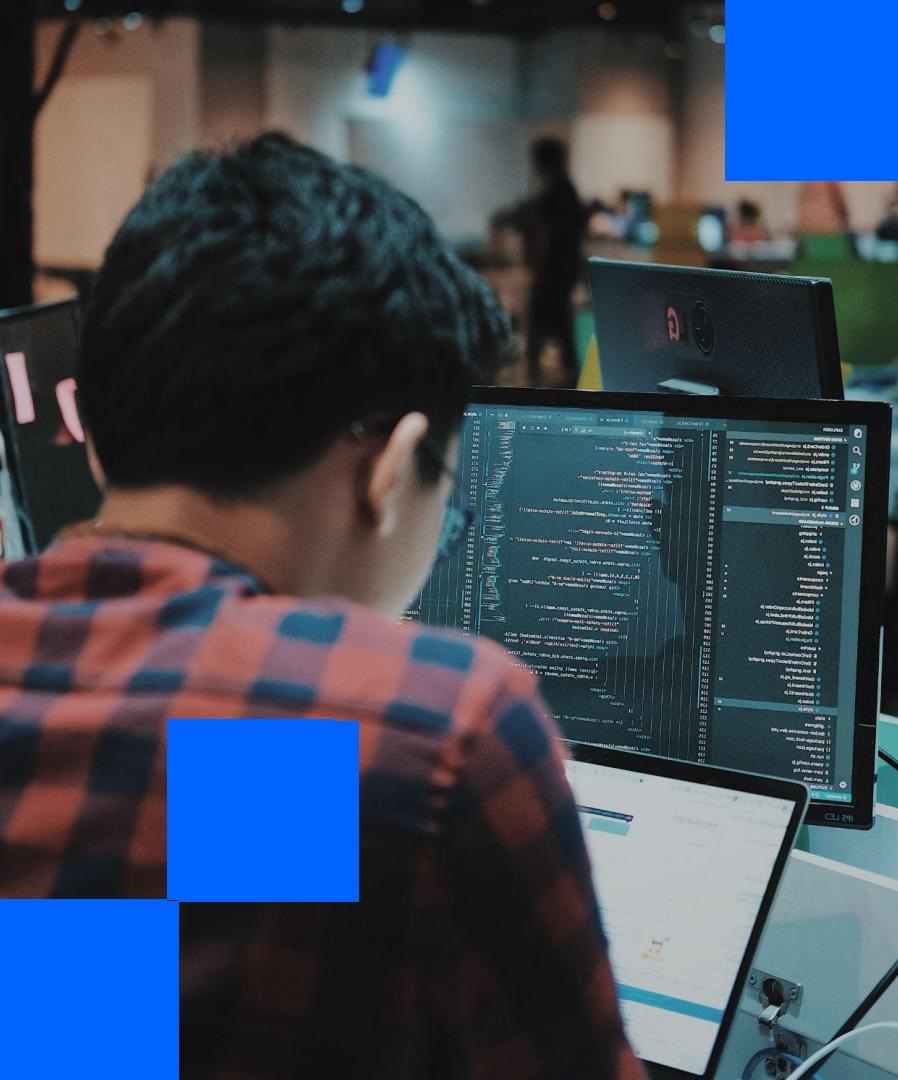
# Structures

**Le type point pourra être initialisé.**

```
typedef struct
{
    int x;
    int y;
} point;
```

**Deux entiers, x et y, composent la struct.**

# / Prototypes

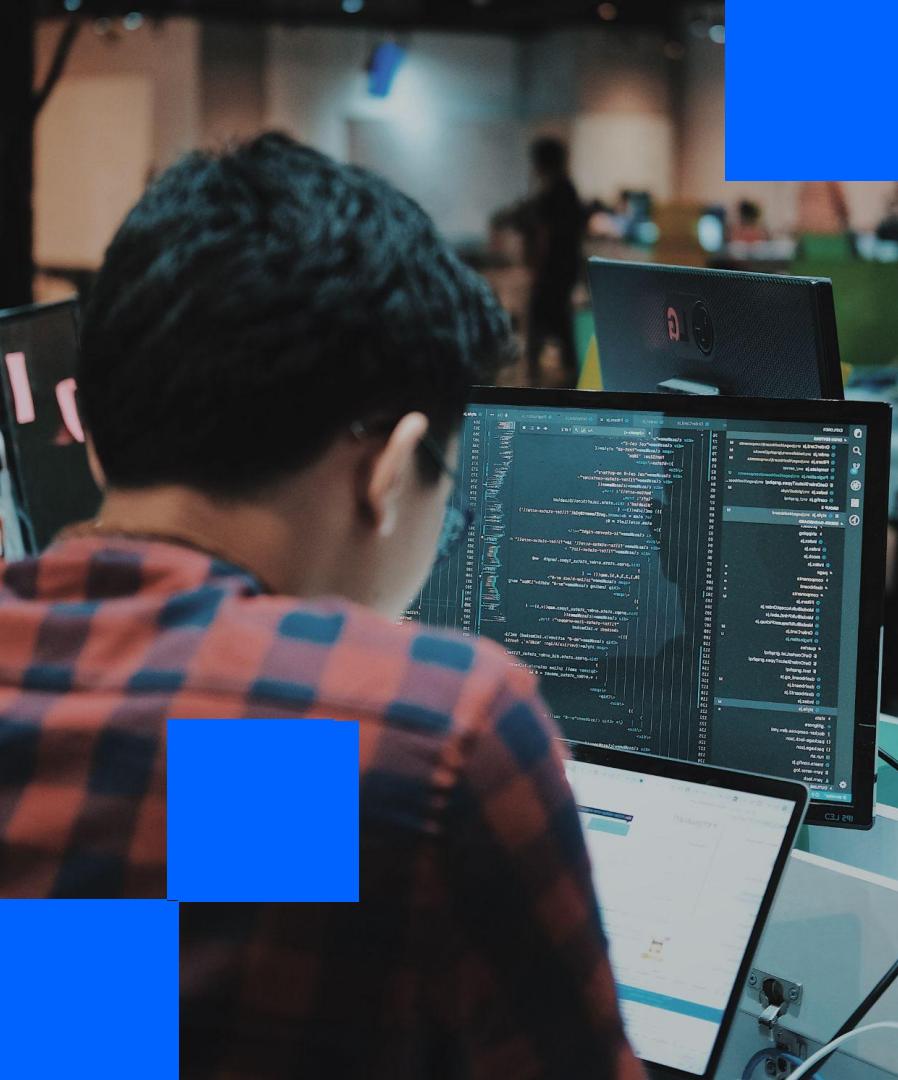


# Prototypes

```
int distance(point a, point b);
```

**Un prototype représente une fonction présente dans un fichier cpp. Il est nécessaire pour appeler une fonction qui est présente dans un autre fichier.**

# / Utilisation



# Utilisation – distance.hpp

```
#pragma once  
  
#define PI (3.14159)  
  
#define ABS(x) (x < 0 ? -x : x)  
  
typedef struct  
{  
    int x;  
    int y;  
} point;  
  
int distance(point a, point b);
```

# Utilisation – main.cpp

```
#include "distance.h"

int main(void)
{
    point a;
    point b;
    int dist;
    a.x = 15;
    a.y = 10;
    b.x = 32;
    b.y = 4;
    dist = distance(a, b);
    return (0);
}
```

**Pour accéder à un attribut d'une structure, on utilise le point.**

**Une structure peut être passée en argument comme n'importe quelle variable.**

# Distance.cpp

```
#include "distance.h"

int distance(point a, point b)
{
    return (ABS(a.x - b.x) + ABS(a.y - b.y));
```

# / Les pointeurs intelligents



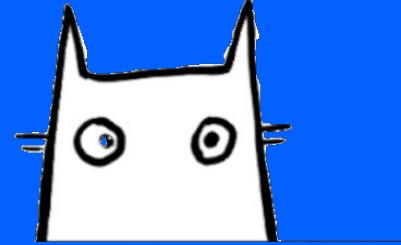


# Contexte et motivation

Pourquoi parle-t-on de "pointeurs intelligents" ?

- En C++, on gère souvent la mémoire manuellement avec new / delete.
- Cette gestion est **complexe et risquée**
  - Oublier un **delete** → fuite mémoire.
  - Doubler un **delete** → crash ou corruption mémoire.
  - Utiliser un pointeur déjà supprimé → comportement indéfini.
  - Solution moderne : **RAll + pointeurs intelligents (Smart Pointers)**.

# Les principaux pointeurs intelligents



# unique\_ptr



```
● ● ●

#include <memory>
#include <iostream>

void example() {
    std::unique_ptr<int> ptr = std::make_unique<int>(42);
    std::cout << *ptr << std::endl;
} // La mémoire est libérée ici automatiquement
```

- Possède **seul** un objet (pas de copie possible).
- Mémoire libérée quand le pointeur sort du scope.
- Léger, sûr, performant.

# shared\_ptr



```
std::shared_ptr<int> ptr1 = std::make_shared<int>(100);
std::shared_ptr<int> ptr2 = ptr1;

std::cout << *ptr1 << std::endl;
std::cout << ptr1.use_count() << std::endl;
```

- Plusieurs pointeurs peuvent **partager** la même ressource.
- Un **compteur de référence** est maintenu.
- La mémoire est libérée quand **tous** les pointeurs sont détruits.

# weak\_ptr



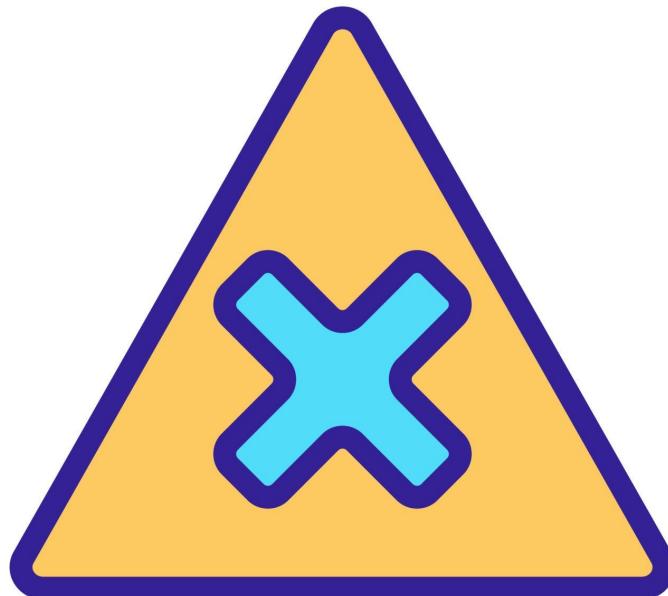
```
● ● ●  
std::shared_ptr<int> sp = std::make_shared<int>(99);  
std::weak_ptr<int> wp = sp;  
  
if (auto tmp = wp.lock()) {  
    std::cout << *tmp << std::endl;  
}
```

- **N'observe** qu'un shared\_ptr sans en augmenter le compteur.
- Empêche les **références circulaires** (shared\_ptr ↔ shared\_ptr).
- Doit être utilisé avec **.lock()** pour accéder à la ressource.

# Cas d'utilisation concrets

Situation	Recommandation
Objet à <b>un seul</b> propriétaire	<b>unique_ptr</b>
Objet partagé entre <b>plusieurs</b> modules	<b>shared_ptr</b>
Besoin d'observer <b>sans posséder</b>	<b>weak_ptr</b>
Cycles d'objets (A ↔ B)	<b>shared_ptr + weak_ptr</b>
Ressource externe (fichier, DB, etc.)	<b>unique_ptr</b> avec custom deleter

# Les erreurs à éviter



- Utiliser **shared\_ptr** partout "juste au cas où"
- **Oublier** le std::move avec unique\_ptr
- Utiliser shared\_ptr **entre** objets circulaires sans weak\_ptr
- Penser que shared\_ptr est **gratuit** → coût CPU et mémoire (réf. count)



# Bonnes pratiques modernes

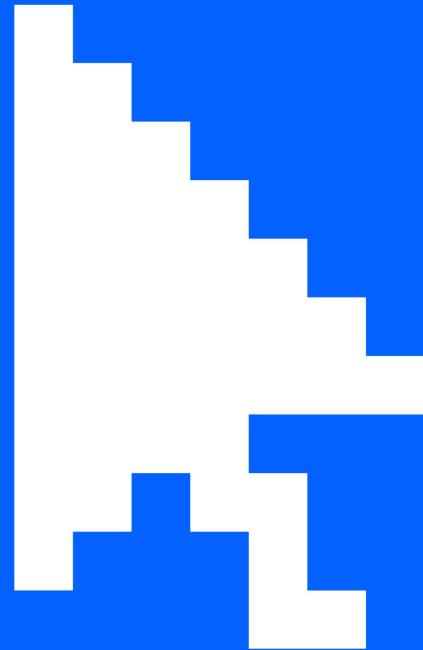
- Utiliser **std::make\_unique / std::make\_shared** (sécurité + performance).
- Préférer **unique\_ptr** si possible.
- Réserver **shared\_ptr** aux cas où plusieurs objets doivent accéder à la ressource.
- Penser **RAII** : toute ressource doit avoir un propriétaire clair.

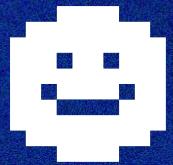


# Conclusion

- Les **pointeurs intelligents** simplifient la gestion mémoire.
- Ils sont essentiels pour écrire du **C++ moderne, sûr et propre**.
- Utiliser le bon type selon le contexte est crucial.
- L'automatisation de la libération mémoire évite de nombreux bugs critiques.

# Des Questions ?





La Plateforme