TP C++ — Fondamentaux du langage

Thibault Dupont

Informations générales

Durée : 4 heures
Prérequis : Programmation en C (variables, boucles, fonctions, pointeurs)
Objectifs : Découvrir les spécificités du C++ moderne par rapport au C

Ce TP est fait pour permettre de manipuler les concepts de base du C++ (références, auto, surcharge, new/delete) et comprendre les différences avec le C. Il ne se substitue aucunement au cours qui sera vu dans le prochain cours et qui théorisera d'avantage les concepts.

1 Prérequis C

> **Important :** Cette partie vérifie que vous maîtrisez les bases du langage C, nécessaires pour aborder les spécificités du C++. Si vous êtes à l'aise avec ces concepts, vous pouvez passer rapidement à la partie suivante.

Vous pouvez tester les différents codes sur godbolt.org en sélectionnant "C" comme langage et en collant les snippets ou encore sur cpp.sh. Le C++ est très largement compatible avec le C, et dans notre cas compiler avec un compilateur C++ ne devrait poser aucun problème.

Pour les anciennes normes du C (avant C99), la déclaration des variables doit se faire au début du bloc { … } (avant toute instruction). Nous nous placerons dans le cas correspondant au standard C18, qui accepte la déclaration en milieu de bloc et dans des portées de bloc et dans la porté d'une boucle : `for(int i=...)`.

1.1 Structure d'un programme

```c
#include <stdio.h>

int main(void) {
    printf("Hello world\n");
    return 0;
}
```

Le minimum étant

```c
int main(void) { return 0; }
```

Éléments essentiels :

Élément	Rôle
`#include <stdio.h>`	Inclusion d'un header via préprocesseur
`int main(void)`	Fonction d'entrée du programme
`{ ... }`	Bloc d'instructions

Élément	Rôle
`;`	Fin d'instruction
`return 0;`	Code de retour (0 = succès)

Rappels importants :

- Le **`;` termine une instruction**, pas une ligne
- Les `{ }` définissent un **bloc de portée**
- Les variables déclarées dans un bloc n'existent que dans ce bloc

1.2 Structures conditionnelles

```c
// if / else
if (x > 0) {
    printf("Positif\n");
} else if (x < 0) {
    printf("Négatif\n");
} else {
    printf("Zéro\n");
}

// switch (entiers/char uniquement)
switch (choice) {
    case 1:
        printf("Un\n");
        break;  // Important !
    case 2:
        printf("Deux\n");
        break;
    default:
        printf("Autre\n");
}
```

Attention : Sans `break`, le `switch` continue sur les cas suivants (fallthrough).

1.3 Boucles

```c
// while : condition vérifiée avant
while (i < 5) {
    printf("%d\n", i);
    i++;
}

// do-while : exécuté au moins une fois
do {
    printf("%d\n", i);
    i++;
} while (i < 5);

// for : nombre d'itérations connu
for (int i = 0; i < 5; i++) {
// le i ici n'est pas le même que le i du while - sa portée est limitée au for
// que ferait `for (i = 0; i < 5; i++) {` ?
    printf("%d\n", i);
}
```

Contrôle de flux :

- `break` : Sort de la boucle
- `continue` : Passe à l'itération suivante
- `return` : Sort de la fonction

1.4 Fonctions

```c
// Déclaration et définition
int add(int a, int b) {
    return a + b;
}

// Appel
int result = add(3, 4);
```

Types de passage de paramètres (rappel C) :

- **Par valeur** : `void f(int x)` → copie
- **Par pointeur** : `void f(int* x)` → modification via `*x`

1.5 Pointeurs

```c
int a = 10;
int* p = &a;  // p pointe vers a
printf("a = %d, *p = %d\n", a, *p);
```

Opérations sur les pointeurs :

- `&` en dehors de la déclaration : adresse de la variable
- `*` en dehors de la déclaration : déréférencement (accès à la valeur pointée)
- `*` dans la déclaration : création d'un pointeur

1.6 Variables et pointeurs constants

```c
#include <stdio.h>

int main(void) {
    int a = 5, b = 0; // variable normale
    const int x = 5;  // x ne peut pas être modifié après initialisation
    const int* p1 = &a;  // pointeur vers un int constant (valeur protégée)
    int const* p2 = &a;  // pointeur constant vers un int (adresse protégée)
    int* const p3 = &a;  // pointeur constant vers un int (adresse protégée)
    const int* const p4 = &a; // pointeur constant vers un int constant (adresse et valeur pr

    // *p1 += 1; // interdit
    // *p2 += 1; // interdit
    *p3 += 1;
    // *p4 += 1; // interdit

    p1 = &b;
    p2 = &b;
    // p3 = &b; // interdit
    // p4 = &b; // interdit
```

```c
    return 0;
}
```

Rappel mémotechnique : `const` protège ce qui est à sa gauche (ou à sa droite s'il est en début de déclaration).

1.7 Structures

```c
#include <stdio.h>

typedef struct {
    int x;
    int y;
} Point;

int main(void) {
    Point p1 = { 3, 4 };
    Point p2 = { .x = 5, .y = 6 };
    printf("Point p1: (%d, %d)\n", p1.x, p1.y);
    printf("Point p2: (%d, %d)\n", p2.x, p2.y);

    return 0;
}
```

Pensez au `;` à la fin de la déclaration de la structure ! La déclaration de la structure est une instruction, elle doit donc être terminée par un `;`.

En C, le type d'une structure est `struct Nom` (ex: `struct Point`), sauf si on utilise `typedef` pour créer un alias (ex: `Point`).

1.8 Mémoire dynamique

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int* arr = (int*) malloc(sizeof(int));  // allocation d'un entier
```

```c
    if (arr == NULL) {
        fprintf(stderr, "Erreur d'allocation mémoire\n");
        return 1;
    }

    *arr = 42;  // utilisation du tableau
    printf("Valeur: %d\n", *arr);

    free(arr);  // libération de la mémoire
    return 0;
}
```

1.9 Auto-évaluation

Objectif : Vérifier les bases du C avant de passer au C++.

1.9.1 Questions théoriques

1. Quelle est la différence entre `while` et `do-while` ?
2. Pourquoi met-on `break` dans un `switch` ?
3. Où vit la variable `i` déclarée dans un `for(int i=0; ...)` ?
4. Comment modifier une variable passée en paramètre (en C) ?

1.9.2 Exercices classiques (boucles + affichage)

Consigne générale : écrire des programmes en C, en utilisant des boucles et des conditions.

1.9.2.1 Exercice A — Rectangle plein

Énoncé : Écrire un programme qui lit `largeur` et `hauteur`, puis affiche un rectangle plein de `#`.

Entrée :

```
5 3
```

Sortie attendue :

```
#####
#####
#####
```

Contraintes : utiliser deux boucles imbriquées.

1.9.2.2 Exercice B — Rectangle creux

Énoncé : Même entrée, mais afficher seulement les bords (le centre est en espaces).

Entrée :

```
6 4
```

Sortie attendue :

```
######
#    #
#    #
######
```

1.9.2.3 Exercice C — Triangle rectangle (aligné à gauche)

Énoncé : Lire `n` et afficher un triangle de hauteur `n`.

Entrée :

```
4
```

Sortie attendue :

```
#
##
###
####
```

1.9.2.4 Exercice D — Liste chainée

Énoncé : Implémenter une liste chainée d'entiers avec les fonctions `add_front`, `add_back`, `print_list` et `free_list`.

La structure de la liste est la suivante :

```c
typedef struct Node {
    int value;
    struct Node* next;
} Node;
```

2 Spécificités C++

Cette section présente quelques **nouveautés du C++** -depuis ses débuts jusqu'à la norme C++22- par rapport au C. Ces concepts seront utilisés dans tous les exercices.

2.1 Entrées/sorties avec iostream

Différence fondamentale avec C :

Aspect	C (stdio.h)	C++ (iostream)
Header	`#include <stdio.h>`	`#include <iostream>`
Sortie	`printf("x=%d\n", x);`	`std::cout << "x=" << x << "\n";`
Entrée	`scanf("%d", &x);`	`std::cin >> x;`
Type safety	Non (format string)	Oui (surcharge)
Format manuel	Oui (`%d`, `%s`…)	Non (automatique)

En C :

```c
#include <stdio.h>
int x = 42;
printf("Valeur: %d\n", x);
scanf("%d", &x);
```

En C++ :

```cpp
#include <iostream>
int x = 42;
std::cout << "Valeur: " << x << "\n";
std::cin >> x;
```

Opérateur << (insertion) :

- C'est un **opérateur surchargé** pour chaque type

- `std::cout << x` retourne `std::cout` → permet le chaînage
- `std::cout << "a" << 1 << 3.14` fonctionne car évalué de gauche à droite

Opérateur >> (extraction) :

- Lit depuis `std::cin`
- Conversion automatique selon le type de la variable
- Saute les espaces/retours à la ligne automatiquement

Gestion d'erreurs :

```
int number;
std::cin >> number;
if (std::cin.fail()) {  // Échec de lecture (ex: texte au lieu de nombre)
    std::cin.clear();   // Réinitialiser l'état
    std::cin.ignore(1000, '\n');  // Vider le buffer jusqu'à \n
}
```

2.2 Références (nouveauté C++)

Concept clé : Une référence est un **alias** d'une variable existante.

Pointeur (C et C++) :

```
int x = 5;
int* p = &x;    // p contient l'adresse de x
*p = 10;        // Modification via déréférencement
p = nullptr;    // Peut pointer vers rien
```

Référence (C++ uniquement) :

```
int x = 5;
int& ref = x;   // ref est un ALIAS de x (même case mémoire)
ref = 10;       // Modifie directement x, pas de * nécessaire
// ref ne peut PAS être null
// ref ne peut PAS être réassigné vers une autre variable
```

Tableau comparatif :

Aspect	Pointeur	Référence
Déclaration	`int* p`	`int& ref`

Aspect	Pointeur	Référence
Initialisation obligatoire	Non	**Oui**
Peut être null	Oui	**Non**
Peut changer de cible	Oui	**Non**
Syntaxe d'accès	`*p = 5;`	`ref = 5;`
Arithmétique	Oui (`p++`)	Non

- `&` en dehors de la déclaration : adresse de la variable
- `&` dans la déclaration : création d'une référence
- `*` en dehors de la déclaration : déréférencement (accès à la valeur pointée)
- `*` dans la déclaration : création d'un pointeur

Passage par référence (simplifie le code) :

```
// En C : passage par pointeur
void swap_c(int* a, int* b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
int x = 5, y = 10;
swap_c(&x, &y);  // & requis à l'appel, syntaxe lourde

// En C++ : passage par référence
void swap_cpp(int& a, int& b) {
    int tmp = a;   // Pas de * !
    a = b;
    b = tmp;
}
int x = 5, y = 10;
swap_cpp(x, y);  // Syntaxe simple, modification garantie
```

Quand utiliser quoi ?

- **Référence** : Quand la cible existe toujours et ne change pas
- **Pointeur** : Quand peut être null, ou réassignation nécessaire

Les références sont souvent utilisées pour les paramètres de fonctions qui doivent modifier les arguments, ou -avec des références constantes- pour éviter les copies coûteuses.

```cpp
int a = 0;
const int& ref_a = a;  // Référence constante : ne peut pas modifier a via ref_a
int const& ref_b = a;  // Même chose, ordre de const ne change rien
// int& const ref_c = a;  // ERREUR
//  const ne peut pas qualifier une référencepar définition
```

2.3 auto : déduction de type (C++11)

Principe : Le compilateur **déduit automatiquement** le type à la compilation.

```cpp
auto a = 42;         // Type déduit : int
auto b = 3.14;       // Type déduit : double
auto c = "hello";    // Type déduit : const char*
auto d = 3.14f;      // Type déduit : float
auto e = 'A';        // Type déduit : char
auto f = true;       // Type déduit : bool
auto g = 3u;       // Type déduit : unsigned int
//...

int x = 5;
auto y = x;          // y est int (copie de x)
auto& z = x;         // z est int& (référence à x)
const auto& w = x;   // w est const int& (référence constante)
```

Règles importantes :

- Déduction à la **compilation** (typage statique, pas dynamique)
- Type fixé définitivement
- `auto` copie par défaut, `auto&` crée une référence

2.4 Surcharge de fonctions (overloading)

En C : Impossible, il faut des noms différents

```cpp
int add_int(int a, int b);
double add_double(double a, double b);
```

Le mangling (nom interne) est simple : `add_int`, `add_double`.

En C++ : Même nom, signatures différentes

```cpp
int max(int a, int b) {
    return (a > b) ? a : b;
}

double max(double a, double b) {
    return (a > b) ? a : b;
}


// Résolution à la compilation selon les types d'arguments
max(5, 10);        // Appelle max(int, int)
max(3.14, 2.71);   // Appelle max(double, double)
```

Le mangling est plus complexe : `max__i_i`, `max__d_d` (exemples de noms internes générés).

Ambiguïté :

```cpp
max(5, 3.14);  // ERREUR : types mixtes (int, double)
               // Le compilateur ne sait pas quelle version choisir
```

On notera que les conversions implicites peuvent parfois résoudre l'ambiguïté, mais il est préférable d'éviter les cas où le compilateur doit faire des choix complexes.

2.5 Gestion mémoire : new/delete

Comparaison C vs C++ :

Aspect	C (malloc/free)	C++ (new/delete)
Allocation	`int* p = (int*)malloc(sizeof(int));`	`int* p = new int;`
Initialisation	Manuelle après malloc	`int* p = new int(42);`
Tableau	`int* a = (int*)malloc(5*sizeof(int));`	`int* a = new int[5];`
Libération	`free(p);`	`delete p;`
Libération tableau	`free(a);`	`delete[] a;`

Exemple C :

```c
int* arr = (int*)malloc(5 * sizeof(int));
if (arr == NULL) { /* erreur */ }
// Utilisation
free(arr);
```

Exemple C++ :

```cpp
// Allocation unique avec initialisation
int* p = new int(42);    // Alloue ET initialise à 42
delete p;                // Libération

// Allocation tableau
int* arr = new int[5];   // Alloue 5 entiers
delete[] arr;            // Libération tableau ([] important!)
```

RÈGLES CRITIQUES (causent des crashes si violées) :

1. `new` → `delete` (objet unique)
2. `new[]` → `delete[]` (tableau)
3. **JAMAIS** mélanger malloc/free avec new/delete
4. **TOUJOURS** libérer ce qui est alloué (sinon fuite mémoire)

Erreurs fatales courantes :

```cpp
int* arr = new int[10];
delete arr;       // FAUX : devrait être delete[]

int* p = new int(5);
delete p;
delete p;         // FAUX : double delete = crash

int* q = new int(10);
delete q;
std::cout << *q;  // FAUX : utilisation après libération = comportement indéfini
```

2.6 nullptr vs NULL vs 0 (C++11)

Évolution historique :

```cpp
// C : NULL est une macro
#define NULL 0
int* p = NULL;   // En réalité : int* p = 0;
```

```cpp
// C++ (avant C++11) : même problème
int* p = NULL;   // Hérité du C
int* q = 0;      // Équivalent
```

```cpp
// C++11+ : nullptr (type dédié)
int* p = nullptr;  // Type: std::nullptr_t
```

Pourquoi nullptr est nécessaire ?

```cpp
void function(int x) {
    std::cout << "Version int\n";
}

void function(int* p) {
    std::cout << "Version pointeur\n";
}

function(0);         // Appelle version int
function(NULL);      // Appelle version int sur certains compilateurs
function(nullptr);   // Appelle version int*
```

Règle absolue : En C++ moderne, **TOUJOURS** utiliser `nullptr`, jamais `NULL` ou `0`.

2.7 std::string vs C-strings

Problèmes des C-strings (`char*`) :

```cpp
char str[20] = "Hello";
strlen(str);           // Longueur (parcourt jusqu'à \0)
strcmp(str, "World");  // Comparaison (retourne <0, 0, >0)
strcpy(str, "Hi");     // Copie (DANGEREUX: pas de vérification taille)
strcat(str, " !");     // Concaténation (DANGEREUX: buffer overflow)
```

Dangers :

- Pas de vérification de bornes → **buffer overflow**
- Gestion manuelle du `\0` terminal
- Fonctions non sûres (strcpy, strcat…)
- Pas d'opérateurs (+, ==, <…)

Solution C++ : std::string

Le `std::` est un espace de noms (namespace) qui contient la classe `string` et d'autres éléments de la bibliothèque standard. Il est nécessaire pour éviter les conflits de noms. Il est possible d'utiliser ses propres espaces de noms pour organiser le code, mais `std` est celui de la bibliothèque standard.

```cpp
#include <string>

std::string s = "Hello";

// Opérations simples et sûres
s.length();            // Taille
s + " World";          // Concaténation avec +
s += "!";              // Ajout avec +=
s == "Hello";          // Comparaison avec ==
s < "World";           // Ordre lexicographique avec <

// Accès aux caractères
s[0];                  // Accès rapide (pas de vérification)
s.at(0);               // Accès sécurisé (exception si hors bornes)

// Manipulation
s.substr(0, 5);        // Extraction sous-chaîne
s.find("ll");          // Recherche (retourne position ou npos)

// Conversion vers C-string (si nécessaire pour API C)
const char* c_str = s.c_str();
```

Conversions numériques (C++11) :

```cpp
// String → Nombre
int i = std::stoi("42");
double d = std::stod("3.14");
long l = std::stol("123456");

// Nombre → String
```

```cpp
std::string s1 = std::to_string(42);
std::string s2 = std::to_string(3.14);
```

Règle : Utiliser `std::string` partout, `.c_str()` seulement pour interfacer avec du code C legacy.

2.8 Les structures

Les structures (`struct`) existent à la fois en C et en C++. Alors qu'en C, une structure ne définit pas un type à part entière (il faut utiliser `struct Point`), en C++ une structure est un type complet et peut être utilisée sans le mot-clé `struct`. Il n'est donc pas nécessaire de préfixer les variables de type `Point` avec `struct` ou de faire un typedef.

```cpp
// En C
struct Point {
    int x;
    int y;
};
struct Point p1;  // Nécessite struct

// ou
typedef struct {
    int x;
    int y;
} Point;
Point p2;  // Avec typedef, struct n'est plus nécessaire

// En C++
struct Point {
    int x;
    int y;
};
Point p1;  // Pas besoin de struct
```

3 Premiers pas en C++

Environnement : godbolt.org ou cpp.sh

3.1 Exercice : Hello World et I/O C++

Objectif : Maîtriser `std::cout` et `std::cin`

```
// TODO: Écrire un programme qui :
// 1. Demande le nom de l'utilisateur (std::cout pour afficher, std::cin pour lire)
// 2. Demande son âge (entier)
// 3. Affiche "Bonjour [nom], vous avez [age] ans"
//
// N'utilisez PAS printf/scanf, uniquement std::cout et std::cin
// Utilisez std::string pour le nom (pas char[])
```

Test attendu :

```
Entrée utilisateur:
Alice
25

Sortie:
Bonjour Alice, vous avez 25 ans
```

Questions de compréhension :

1. Pourquoi n'a-t-on pas besoin de `%s` ou `%d` avec `std::cout` ?
2. Testez avec un nom composé ("Jean Pierre"). Que se passe-t-il ?

Aide : `std::cin >> name` lit jusqu'au premier séparateur. Pour lire une ligne complète : `std::getline(std::cin, name)`.

4 Gestion mémoire et tableaux dynamiques

4.1 Exercice : Liste chaînée simple

Objectif : Maîtriser l'allocation dynamique en C++

Reprenez la liste chaînée implémentée en C dans le prérequis, et réécrivez-la en C++ en utilisant `new` et `delete` au lieu de `malloc` et `free`.

On utilisera :

```
struct Node {
    int value;
    Node* next;
};
```

4.2 Exercice : Tableau dynamique redimensionnable

Objectif : Implémenter une structure similaire à `std::vector` (que l'on verra plus tard) pour comprendre les mécanismes d'allocation dynamique et de redimensionnement sur des tableaux.

Quelques informations sur les std::vector.

```
#include <iostream>

struct DynamicArray {
    int* data;      // Pointeur vers les données
    int size;       // Nombre d'éléments actuels
    int capacity;   // Capacité totale allouée
};

// TODO: Implémenter les fonctions suivantes

void init(DynamicArray& arr, int initial_capacity) {
    // 1. Allouer data avec new[]
```

```
    // 2. Initialiser size à 0 (aucun élément)
    // 3. Initialiser capacity à initial_capacity
}

void destroy(DynamicArray& arr) {
    // 1. Libérer data avec delete[]
    // 2. Mettre data à nullptr pour éviter dangling pointer ou pointeur flottant
    // 3. Mettre size et capacity à 0
}

bool is_empty(const DynamicArray& arr) {
    // Retourner true si size == 0
    return false;  // À modifier
}

bool is_full(const DynamicArray& arr) {
    // Retourner true si size == capacity
    return false;  // À modifier
}

void push_back(DynamicArray& arr, int value) {
    // Si le tableau est plein (size == capacity) :
    //   1. Calculer nouvelle capacité (doubler : capacity * 2)
    //   2. Allouer un nouveau tableau avec new[]
    //   3. Copier tous les éléments de l'ancien vers le nouveau
    //   4. Libérer l'ancien tableau avec delete[]
    //   5. Mettre à jour data et capacity
    //
    // Ajouter value à la position size
    // Incrémenter size
}

void print(const DynamicArray& arr) {
    std::cout << "[ ";
    for (int i = 0; i < arr.size; i++) {
        std::cout << arr.data[i] << " ";
    }
    std::cout << "] (size=" << arr.size
              << ", capacity=" << arr.capacity << ")\n";
}

int main() {
```

```
    DynamicArray arr;
    init(arr, 2);  // Capacité initiale : 2

    std::cout << "Ajout progressif d'éléments:\n";
    for (int i = 0; i < 10; i++) {
        push_back(arr, i * i);
        print(arr);
    }

    destroy(arr);

    return 0;
}
```

Questions :

1. Pourquoi passer `DynamicArray&` et pas `DynamicArray` ?
2. Combien de fois le tableau est-il redimensionné pour ajouter 10 éléments (capacité initiale 2) ?
3. Quel est le problème si on oublie d'appeler `destroy()` ?
4. Pourquoi doubler la capacité plutôt que l'augmenter de 1 ?

Sortie attendue :

```
Ajout progressif d'éléments:
[ 0 ] (size=1, capacity=2)
[ 0 1 ] (size=2, capacity=2)
[ 0 1 4 ] (size=3, capacity=4)
[ 0 1 4 9 ] (size=4, capacity=4)
[ 0 1 4 9 16 ] (size=5, capacity=8)
[ 0 1 4 9 16 25 ] (size=6, capacity=8)
[ 0 1 4 9 16 25 36 ] (size=7, capacity=8)
[ 0 1 4 9 16 25 36 49 ] (size=8, capacity=8)
[ 0 1 4 9 16 25 36 49 64 ] (size=9, capacity=16)
[ 0 1 4 9 16 25 36 49 64 81 ] (size=10, capacity=16)
```

5 VSCode et Makefile

5.1 Installation et configuration

Étapes :

1. **Installer VSCode** : https://code.visualstudio.com/
2. **Installer extensions C++** (dans VSCode, onglet Extensions) :
   - `C/C++` (Microsoft) - essentiel
   - `C/C++ Extension Pack` (optionnel mais recommandé)
3. **Vérifier le compilateur** (ouvrir un terminal) :

   ```
   g++ --version
   # ou
   clang++ --version
   ```

   Si pas installé :

   - **Linux Ubuntu/Debian** : `sudo apt install g++ make`
   - **Linux Fedora/RedHat** : `sudo yum install gcc-c++ make`
   - **macOS** : `xcode-select --install`
   - **Windows** : Installer MinGW-w64 ou MSYS2

5.2 Premier projet multi-fichiers

Créer la structure :

```
mkdir tp_cpp_partie3
cd tp_cpp_partie3
```

Créer les 4 fichiers suivants :

5.2.1 utils.h

```h
#ifndef UTILS_H
#define UTILS_H

// Déclarations uniquement (interface publique)
int max(int a, int b);
int min(int a, int b);
void swap(int& a, int& b);

#endif  // UTILS_H
```

5.2.2 utils.cpp

```cpp
#include "utils.h"

// TODO: Implémenter max
// Retourne le plus grand des deux nombres

// TODO: Implémenter min
// Retourne le plus petit des deux nombres

// TODO: Implémenter swap
// Échange les valeurs de a et b (utiliser une variable temporaire)
```

5.2.3 main.cpp

```cpp
#include <iostream>
#include "utils.h"

int main() {
    int a = 10, b = 20;

    std::cout << "Valeurs initiales: a = " << a << ", b = " << b << "\n";
    std::cout << "Max(a, b) = " << max(a, b) << "\n";
    std::cout << "Min(a, b) = " << min(a, b) << "\n";

    swap(a, b);
```

```
    std::cout << "Après swap: a = " << a << ", b = " << b << "\n";

    return 0;
}
```

5.2.4 `Makefile`

On ne demande pas de le comprendre pour le moment, mais voici un Makefile de base pour compiler ce projet :

```
# Variables de configuration
CXX = g++
CXXFLAGS = -std=c++20 -Wall -Wextra -g

# Nom de l'exécutable
TARGET = program

# Fichiers source et objets
SRCS = main.cpp utils.cpp
OBJS = $(SRCS:.cpp=.o)

# Règle par défaut (exécutée par "make" sans argument)
all: $(TARGET)

# Création de l'exécutable à partir des .o
$(TARGET): $(OBJS)
    $(CXX) $(CXXFLAGS) -o $(TARGET) $(OBJS)

# Règle générique : .cpp → .o
%.o: %.cpp
    $(CXX) $(CXXFLAGS) -c $< -o $@

# Nettoyage des fichiers générés
clean:
    rm -f $(OBJS) $(TARGET)

# Recompilation complète
rebuild: clean all

# Déclaration des cibles "phony" (pas de fichiers réels)
.PHONY: all clean rebuild
```

Compilation et exécution :

```
make          # Compile le projet
./program     # Exécute
make clean    # Nettoie les .o et l'exécutable
make rebuild  # Nettoie puis recompile tout
```

Sortie attendue :

```
Valeurs initiales: a = 10, b = 20
Max(a, b) = 20
Min(a, b) = 10
Après swap: a = 20, b = 10
```

Questions (pour l'impatient) :

1. À quoi servent les fichiers `.o` ?
2. Pourquoi séparer `.h` et `.cpp` ?
3. Que fait `#ifndef UTILS_H` ?
4. Si vous modifiez seulement `utils.cpp`, quels fichiers sont recompilés ?
5. Quelle différence entre `make` et `make rebuild` ?

6 Chaînes de caractères

6.1 Exercice : std::string

Créer un nouveau projet :

```
mkdir string_test
cd string_test
```

Créer `string_test.cpp` :

```
#include <iostream>
#include <string>

int main() {
    std::string cpp_str1 = "Hello";
    std::string cpp_str2 = "World";

    // TODO: Afficher la longueur avec .length() ou .size()
    std::cout << "Longueur: " << "\n";

    // TODO: Comparer avec == et
    if (true) {
        std::cout << "cpp_str1 est égal à 'Hello'\n";
    }

    if (true) {
        std::cout << cpp_str1 << " < " << cpp_str2 << " (ordre lexico)\n";
    }

    // TODO: Concaténer avec + et +=
    std::string cpp_result ;
    std::cout << "Concat C++: " << cpp_result << "\n";

    // Ajouter "!"
    std::cout << "Après +=: " << cpp_str1 << "\n";
```

```
    std::cout << "\n========================================\n";
    std::cout << "=== PARTIE : Conversions ===\n";
    std::cout << "========================================\n\n";

    // std::string → C-string
    const char* from_cpp ;
    std::cout << "C-string depuis std::string: " << from_cpp << "\n";

    // C-string → std::string
    std::string from_c;
    std::cout << "std::string depuis C-string: " << from_c << "\n";


    std::cout << "\n========================================\n";
    std::cout << "=== PARTIE : Conversions numériques ===\n";
    std::cout << "========================================\n\n";

    // TODO: Convertir string → int avec std::stoi
    std::string num_str = "42";
    int num ;
    std::cout << "String '42' → int: " << num << "\n";

    // TODO: Convertir int → string avec std::to_string
    int value = 123;
    std::string value_str;
    std::cout << "int 123 → string: '" << value_str << "'\n";

    return 0;
}
```

Créer un Makefile :

```
CXX = g++
CXXFLAGS = -std=c++20 -Wall -Wextra

string_test: string_test.cpp
    $(CXX) $(CXXFLAGS) -o string_test string_test.cpp

clean:
    rm -f string_test
```

```
.PHONY: clean
```

Compiler et exécuter :

```
make
./string_test
```

Questions :

1. Quels sont les risques avec `strcpy` et `strcat` ?
2. Pourquoi `std::string` est-il plus sûr ?
3. Que retourne `.c_str()` ? Peut-on modifier le résultat ?
4. Que se passe-t-il si on fait `std::stoi("abc")` ?